

# Rational Rhapsody Property Definitions

# Table of Contents

Activity .....	18
General .....	18
Activity_diagram .....	19
AcceptEventAction .....	19
AcceptTimeEvent .....	20
Action .....	20
ActionBlock .....	21
ActionPin .....	22
ActivityParameter .....	23
AutoPopulate .....	26
ButtonArray .....	27
CallBehavior .....	28
CallOperation .....	29
Comment .....	30
Complete .....	35
ConnectorText .....	35
Constraint .....	36
ControlFlow .....	40
DiagramFrame .....	41
DecisionNode .....	42
DefaultTransition .....	43
DependentText .....	46
Depends .....	46
DiagramConnector .....	50
DiagramFrame .....	50
DigitalDisplay .....	51
Gauge .....	51
General .....	52
HistoryConnector .....	54
interruptibleRegion .....	54
JunctionConnector .....	55
Knob .....	55
Labels .....	56
Led .....	56
LevelIndicator .....	57
LoopTransition .....	57
MatrixDisplay .....	58
Meter .....	59
Names .....	59
Note .....	59
ObjectFlow .....	60
ObjectFlowState .....	61
ObjectNode .....	62
OnOffSwitch .....	63
Partition .....	64
PartitionFrame .....	65

PushButton .....	65
ReferenceActivity .....	66
Requirement .....	69
Requirement .....	73
SelectorConnector .....	74
SendAction .....	75
Slider.....	76
State .....	76
StateDiagram.....	77
SubActivityState.....	77
Swimlane .....	81
TerminationConnector .....	82
TextBox.....	82
Transition .....	83
Ada_CG .....	86
Argument .....	86
Attribute .....	89
Class.....	99
Component .....	121
Configuration .....	123
Dependency.....	132
File .....	136
ModelElement.....	144
Package.....	144
Port .....	152
Type.....	153
Operation .....	163
Relation.....	177
Requirement .....	189
Statechart .....	190
Framework.....	192
GNAT .....	208
GNATVxWorks .....	248
INTEGRITY.....	265
INTEGRITY5.....	274
MultiWin32 .....	284
Multi4Win32 .....	293
OBJECTADA .....	302
RAVEN_PPC .....	308
SPARK.....	314
Event.....	320
Transition .....	323
Generalization.....	324
ADA_Roundtrip .....	326
General .....	326
CPP_Roundtrip::Type.....	328
Update .....	329
Ada2005Containers .....	330
General .....	330
StaticArray .....	331

Bounded .....	335
Unbounded .....	338
BoundedQualified .....	341
UnboundedQualified .....	344
<b>Animation .....</b>	<b>347</b>
Activity .....	347
ClassifierRole .....	347
General .....	348
Message .....	350
TargetMonitoring .....	350
<b>ATL .....</b>	<b>355</b>
Class .....	355
Configuration .....	356
Macro .....	361
Operation .....	366
<b>AutomaticTestGenerator .....</b>	<b>368</b>
Settings .....	368
<b>AUTOSAR .....</b>	<b>370</b>
ARXML .....	370
CodeGeneration .....	370
RTE_API .....	371
<b>Browser .....</b>	<b>373</b>
Operation .....	373
Settings .....	373
<b>CG .....</b>	<b>378</b>
Argument .....	378
Attribute .....	379
CGGeneral .....	383
Class .....	384
Component .....	392
Configuration .....	395
Dependency .....	402
Event .....	404
File .....	408
Framework .....	412
General .....	413
Generalization .....	416
Operation .....	416
Package .....	419
Relation .....	423
Statechart .....	433
Type .....	436
<b>COM .....</b>	<b>440</b>
Argument .....	440
Attribute .....	442
Class .....	445
coclass .....	446
Configuration .....	448
IDL .....	450
Interface .....	452

Library .....	454
Operation .....	456
Relation .....	458
<b>Communication_Diagram .....</b>	<b>460</b>
AssociationRole .....	460
AutoPopulate .....	461
Classifier .....	461
ClassifierActor .....	462
CommunicationDiagramGE .....	462
CollMessage .....	462
Comment .....	463
Complete .....	464
Constraint .....	464
Depends .....	465
DiagramFrame .....	466
General .....	467
Messages .....	467
MultiObj .....	468
Note .....	468
Requirement .....	469
ReverseCollMessage .....	469
<b>ComponentDiagram .....</b>	<b>471</b>
AutoPopulate .....	471
Comment .....	471
Complete .....	472
Component .....	472
ComponentDiagramGE .....	473
CompRealization .....	473
Constraint .....	475
Depends .....	475
DiagramFrame .....	476
FileComponent .....	478
Flow .....	479
FolderComponent .....	480
General .....	481
Interface .....	481
InterfaceComponent .....	483
Note .....	484
Requirement .....	484
<b>ConfigurationManagement .....</b>	<b>486</b>
ClearCase .....	486
General .....	504
PVCS .....	510
RationalTeamConcert .....	522
SCC .....	524
SourceIntegrity .....	533
Synergy .....	542
<b>CORBA .....</b>	<b>545</b>
Attribute .....	545
C++Mapping_CORBABasic .....	545

C++Mapping_CORBAEnum .....	546
C++Mapping_CORBAFixedArray .....	547
C++Mapping_CORBAFixedSequence .....	548
C++Mapping_CORBAFixedStruct .....	549
C++Mapping_CORBAFixedUnion .....	550
C++Mapping_CORBAInterfaceReference .....	551
C++Mapping_CORBAInterfaceVariable .....	552
C++Mapping_CORBASequence .....	553
C++Mapping_CORBAVariableArray .....	554
C++Mapping_CORBAVariableStruct .....	555
C++Mapping_CORBAVariableUnion .....	556
Class .....	557
Configuration .....	559
Operation .....	561
Package .....	561
TAO .....	561
Type .....	568
UserDefinedORB .....	571
<b>CPP_CG .....</b>	<b>578</b>
Activity .....	578
Argument .....	578
Attribute .....	581
CallOperation .....	593
Class .....	595
Configuration .....	616
Cygwin .....	628
DDS .....	645
Dependency .....	645
EnumerationLiteral .....	651
Event .....	652
File .....	656
flowPort .....	665
Framework .....	666
General .....	681
Generalization .....	681
INTEGRITY .....	682
INTEGRITY5 .....	693
Integrity5ESTL .....	710
Link .....	728
IntegrityESTL .....	728
Linux .....	745
Microsoft .....	757
MicrosoftDLL .....	771
MicrosoftWinCE600 .....	785
ModelElement .....	798
MSStandardLibrary .....	803
MSVC .....	816
MSVCDLL .....	836
MSVCStandardLibrary .....	852
Multi4Linux .....	867

Multi4Win32 .....	877
MultiWin32 .....	892
NucleusPLUS-PPC .....	903
Operation .....	916
OsePPCDiab .....	930
OseSfk .....	944
OSPL .....	958
Package .....	958
Port .....	966
QNXNeutrinoMomentics .....	970
QNXNeutrinoGCC .....	984
Relation .....	995
Requirement .....	1007
RTI .....	1009
Solaris2 .....	1009
Solaris2GNU .....	1021
State .....	1032
Statechart .....	1034
Transition .....	1036
Type .....	1037
VxWorks .....	1046
VxWorks6diab .....	1059
VxWorks6diab_RTP .....	1073
VxWorks6gnu .....	1086
VxWorks6gnu_RTP .....	1100
WorkbenchManaged .....	1114
WorkbenchManaged653 .....	1126
WorkbenchManaged_RTP .....	1142
<b>CPP_ReverseEngineering .....</b>	<b>1155</b>
Filtering .....	1155
ImplementationTrait .....	1156
Main .....	1166
MFC .....	1168
MSVC60 .....	1168
Parser .....	1169
Promotions .....	1171
Update .....	1172
<b>CPP_Roundtrip .....</b>	<b>1175</b>
General .....	1175
Update .....	1179
<b>C.CG .....</b>	<b>1182</b>
Argument .....	1182
Attribute .....	1184
AutosarRTE3x .....	1199
CallOperation .....	1210
Class .....	1212
Configuration .....	1237
Cygwin .....	1255
Dependency .....	1272
EnumerationLiteral .....	1276

Event.....	1277
File .....	1282
flowPort.....	1291
Framework.....	1292
Generalization.....	1309
INTEGRITY.....	1310
INTEGRITY5.....	1320
Interface.....	1336
Link .....	1336
Linux .....	1337
MainLoopCygwin .....	1351
MainLoopLinux .....	1351
MainLoopMicrosoft .....	1351
MainLoopMSVC9.....	1352
MainLoopS12.....	1353
MainLoopSTM32 .....	1354
Microsoft .....	1355
MicrosoftIDF .....	1368
ModelElement.....	1380
MSVC .....	1385
MSVCDLL.....	1402
Multi4Win32 .....	1420
NucleusPLUS-PPC.....	1438
Operation .....	1451
OptimizedTopDownStatechart.....	1467
OSEK21NT .....	1471
OSEK21HC12.....	1471
Package.....	1471
Port .....	1478
QNXNeutrinoMomentics .....	1481
Relation.....	1482
Requirement .....	1493
Solaris2.....	1495
Solaris2GNU .....	1507
State .....	1519
Statechart .....	1521
Transition .....	1523
Type.....	1524
VxWorks .....	1534
VxWorks6diab.....	1548
VxWorks6diab_RTP .....	1562
VxWorks6gnu .....	1576
VxWorks6gnu_RTP .....	1590
WorkbenchManaged.....	1605
WorkbenchManaged_RTP .....	1618
C_ReverseEngineering .....	1632
Filtering .....	1632
ImplementationTrait .....	1633
Main .....	1642
MFC .....	1644



MSVC60 .....	1644
Parser .....	1645
Promotions .....	1647
Update .....	1648
<b>C_Roundtrip .....</b>	<b>1651</b>
General .....	1651
Update .....	1654
<b>DDS .....</b>	<b>1656</b>
Type .....	1656
Attribute .....	1657
<b>DeploymentDiagram .....</b>	<b>1658</b>
AutoPopulate .....	1658
Comment .....	1658
Communication_Path .....	1659
Complete .....	1659
ComponentInstance .....	1660
Constraint .....	1661
Depends .....	1661
DeploymentDiagramGE .....	1663
DiagramFrame .....	1663
Flow .....	1664
General .....	1665
NodeProcessor .....	1665
Note .....	1666
Requirement .....	1667
<b>DiagramPrintSettings .....</b>	<b>1668</b>
General .....	1668
<b>Dialog .....</b>	<b>1670</b>
All .....	1670
Attribute .....	1671
Class .....	1672
ClassifierRole .....	1674
Component .....	1674
Configuration .....	1674
Dependency .....	1675
Diagrams .....	1675
Event .....	1676
File .....	1676
General .....	1677
ModelElement .....	1677
ObjectModelDiagram .....	1679
Operation .....	1679
Package .....	1680
Project .....	1681
Relation .....	1681
SequenceDiagram .....	1682
Stereotype .....	1682
TimingDiagram .....	1683
Type .....	1683
UseCaseDiagram .....	1684

<b>Eclipse</b> .....	<b>1685</b>
Configuration .....	1685
DefaultEnvironments .....	1686
Export .....	1687
Project.....	1687
<b>General</b> .....	<b>1689</b>
Attribute .....	1689
Graphics .....	1689
HTTPServer.....	1714
Message .....	1714
Model .....	1715
ModelLibraries .....	1744
Operation .....	1744
Profile.....	1744
Project.....	1745
Relations .....	1746
Report .....	1746
ReporterPLUS .....	1747
RPE .....	1749
TableView .....	1750
Visibility .....	1751
Workspace.....	1751
<b>IntelliVisor</b> .....	<b>1754</b>
General .....	1754
PredefineMacros.....	1755
PredefineMacrosTooltip.....	1756
<b>Java(1.1)Containers</b> .....	<b>1757</b>
BoundedOrdered .....	1757
BoundedUnordered .....	1763
Fixed .....	1770
General .....	1776
Qualified.....	1777
Scalar.....	1783
StaticArray .....	1786
UnboundedOrdered .....	1792
UnboundedUnordered .....	1796
User .....	1801
<b>Java(1.2)Containers</b> .....	<b>1807</b>
BoundedOrdered .....	1807
BoundedUnordered .....	1815
Fixed .....	1822
General .....	1830
Qualified.....	1830
Scalar.....	1838
StaticArray .....	1846
UnboundedOrdered .....	1854
UnboundedUnordered .....	1861
User .....	1869
<b>Java(1.5)Containers</b> .....	<b>1878</b>
BoundedOrdered .....	1878

BoundedUnordered .....	1886
Fixed .....	1893
General .....	1901
Qualified .....	1901
Scalar .....	1909
StaticArray .....	1917
UnboundedOrdered .....	1925
UnboundedUnordered .....	1932
User .....	1940
<b>JAVA_CG .....</b>	<b>1949</b>
AnimInstrumentation .....	1949
Argument .....	1949
Attribute .....	1951
Class .....	1959
Component .....	1974
Configuration .....	1975
Dependency .....	1983
Event .....	1986
File .....	1988
Framework .....	1996
Generalization .....	2009
JDK .....	2009
ModelElement .....	2022
Operation .....	2022
Package .....	2030
Port .....	2036
Relation .....	2038
Requirement .....	2048
Statechart .....	2050
Type .....	2050
<b>JAVA_ReverseEngineering .....</b>	<b>2054</b>
Filtering .....	2054
ImplementationTrait .....	2055
Main .....	2062
MFC .....	2063
MSVC60 .....	2064
Parser .....	2064
Promotions .....	2066
<b>JAVA_Roundtrip .....</b>	<b>2067</b>
General .....	2067
Type .....	2070
Update .....	2070
<b>Model .....</b>	<b>2072</b>
ARBMT .....	2072
AR_BMT .....	2073
Attribute .....	2074
Class .....	2076
Comment .....	2078
ControlledFile .....	2078
General .....	2080

Importer .....	2081
MatrixLayout .....	2082
MatrixView .....	2082
Package.....	2084
Profile.....	2084
Query .....	2087
Relation.....	2088
Stereotype .....	2088
TableLayout.....	2108
TableView .....	2109
Type.....	2109
ModelBasedTesting .....	2111
Settings.....	2111
ObjectModelGe .....	2114
Actor .....	2114
Aggregation .....	2115
Association .....	2118
Attribute .....	2121
AutoPopulate .....	2123
Class.....	2124
ClassDiagram .....	2127
Comment .....	2128
Complete .....	2131
Composition.....	2131
Constraint .....	2134
ContainArrow .....	2137
Depends .....	2138
DiagramFrame .....	2139
Flow .....	2140
flowPort.....	2141
fullPort.....	2143
General .....	2144
Inheritance .....	2145
InstanceSpecification.....	2146
Interface.....	2147
Link .....	2150
Note .....	2152
Object .....	2153
Operation .....	2156
Package.....	2158
Port .....	2159
proxyPort .....	2160
Realization .....	2162
Requirement .....	2163
standardPort .....	2166
Stereotype .....	2167
Tag .....	2168
Type.....	2170
UseCase.....	2172
OMContainers .....	2174

BoundedOrdered .....	2174
BoundedUnordered .....	2180
EmbeddedFixed.....	2187
EmbeddedScalar .....	2194
Fixed .....	2200
General .....	2207
Qualified.....	2207
Scalar.....	2215
StaticArray .....	2221
UnboundedOrdered .....	2228
UnboundedUnordered .....	2235
User .....	2242
<b>OMCorba2CorbaContainers .....</b>	<b>2250</b>
BoundedOrdered .....	2250
BoundedUnordered .....	2258
EmbeddedFixed.....	2265
EmbeddedScalar .....	2273
Fixed .....	2281
General .....	2288
Qualified.....	2289
Scalar.....	2296
StaticArray .....	2304
UnboundedOrdered .....	2312
UnboundedUnordered .....	2319
User .....	2327
<b>OMCpp2CorbaContainers.....</b>	<b>2336</b>
BoundedOrdered .....	2336
BoundedUnordered .....	2344
EmbeddedFixed.....	2351
EmbeddedScalar .....	2359
Fixed .....	2367
General .....	2374
Qualified.....	2375
Scalar.....	2382
StaticArray .....	2390
UnboundedOrdered .....	2398
UnboundedUnordered .....	2406
User .....	2413
<b>OMCppOfCorbaContainers .....</b>	<b>2422</b>
BoundedOrdered .....	2422
BoundedUnordered .....	2430
EmbeddedFixed.....	2437
EmbeddedScalar .....	2445
Fixed .....	2453
General .....	2460
Qualified.....	2461
Scalar.....	2468
StaticArray .....	2476
UnboundedOrdered .....	2484
UnboundedUnordered .....	2492

User .....	2499
<b>OMUContainers .....</b>	<b>2508</b>
BoundedOrdered .....	2508
BoundedUnordered .....	2515
EmbeddedFixed.....	2522
EmbeddedScalar .....	2530
Fixed .....	2537
General .....	2544
Qualified.....	2545
Scalar.....	2552
StaticArray .....	2559
UnboundedOrdered .....	2567
UnboundedUnordered .....	2574
User .....	2581
<b>PanelDiagram .....</b>	<b>2590</b>
ButtonArray.....	2590
DiagramFrame.....	2591
DigitalDisplay .....	2592
Gauge .....	2592
General .....	2593
Knob .....	2594
Led.....	2594
LevelIndicator .....	2595
MatrixDisplay .....	2595
Meter.....	2595
OnOffSwitch.....	2596
PushButton .....	2597
Slider.....	2597
TextBox.....	2598
<b>QoS .....</b>	<b>2599</b>
Class.....	2599
Operation .....	2601
Resource .....	2601
<b>ReverseEngineering.....</b>	<b>2603</b>
Main .....	2603
Progress .....	2604
Update .....	2606
<b>RiCContainers.....</b>	<b>2608</b>
BoundedOrdered .....	2608
BoundedUnordered .....	2616
EmbeddedFixed.....	2623
EmbeddedScalar .....	2631
Fixed .....	2639
General .....	2646
Qualified.....	2647
Scalar.....	2655
StaticArray .....	2662
UnboundedOrdered .....	2670
UnboundedUnordered .....	2678
User .....	2685

<b>RTInterface</b> .....	<b>2694</b>
DOORS.....	2694
ExportOptions .....	2695
<b>SequenceDiagram</b> .....	<b>2704</b>
Animation.....	2704
CancelledTimeout.....	2704
Comment .....	2704
Condition_Mark.....	2707
Constraint .....	2708
CreateMessage .....	2710
DataFlow.....	2711
Depends .....	2711
DestroyMessage.....	2713
DestructionEvent .....	2713
DiagramFrame.....	2714
EventMessage .....	2715
FoundMessage .....	2715
General .....	2716
InstanceLine .....	2721
InteractionOperator.....	2721
LostMessage .....	2722
Message .....	2722
Note .....	2723
ReplyMessage .....	2724
Requirement .....	2724
TimeInterval.....	2727
SequenceDiagram .....	2728
Timeout.....	2728
<b>SPARK</b> .....	<b>2729</b>
Class.....	2729
Package.....	2729
<b>StatechartDiagram</b> .....	<b>2731</b>
AcceptEventAction.....	2731
AcceptTimeEvent.....	2731
AutoPopulate .....	2732
ButtonArray.....	2733
Comment .....	2733
Complete .....	2736
CompState.....	2736
Constraint .....	2737
DefaultTransition.....	2740
Depends .....	2741
DiagramFrame.....	2742
DigitalDisplay .....	2743
Gauge .....	2744
General .....	2744
HistoryConnector .....	2745
Knob .....	2746
Led.....	2746
LevelIndicator .....	2747

MatrixDisplay .....	2747
Meter.....	2748
Note .....	2748
OnOffSwitch.....	2748
PushButton .....	2749
Requirement .....	2750
SendAction .....	2753
Slider.....	2754
State .....	2754
StateDiagram.....	2756
TextBox.....	2757
Transition .....	2758
<b>STLContainers .....</b>	<b>2760</b>
BoundedOrdered .....	2760
BoundedUnordered .....	2766
EmbeddedFixed.....	2772
EmbeddedScalar .....	2779
Fixed .....	2782
General .....	2789
Qualified.....	2790
Scalar.....	2796
StaticArray .....	2802
UnboundedOrdered .....	2809
UnboundedUnordered .....	2815
User .....	2822
<b>TestConductor.....</b>	<b>2828</b>
SDInstance .....	2828
SequenceDiagram .....	2828
Settings.....	2829
TestCase .....	2833
TestContext .....	2837
<b>TimingDiagram.....</b>	<b>2839</b>
CancelledTimeout.....	2839
Comment .....	2839
Constraint .....	2842
CreateMessage .....	2845
Depends .....	2845
DestroyMessage.....	2846
DestructionEvent .....	2847
DiagramFrame .....	2848
EventMessage .....	2849
General .....	2849
Message .....	2850
Note .....	2851
ReplyMessage .....	2852
Requirement .....	2852
TimeAxis .....	2855
Timeout.....	2856
TimingInstanceLine.....	2857
<b>UseCaseExtensions .....</b>	<b>2858</b>



Dependency.....	2858
UseCaseGe.....	2859
Actor .....	2859
Association .....	2860
AutoPopulate .....	2863
Comment .....	2863
Complete .....	2866
Constraint .....	2866
Depends .....	2869
DiagramFrame.....	2870
Flow .....	2871
General .....	2872
Inheritance .....	2873
Note .....	2874
Package.....	2875
Requirement .....	2876
SystemBox.....	2879
UseCase .....	2879
UseCaseDiagram .....	2881
VisualStudio .....	2882
DefaultEnvironments .....	2882
WebComponents .....	2884
Attribute .....	2884
Class.....	2884
Configuration .....	2885
Event.....	2886
File .....	2887
Operation .....	2887
WebFramework .....	2887
WSDL.....	2889
Package.....	2889
XSD.....	2890
Type.....	2890

# Activity

---

Contains properties relating to activities.

## General

Contains properties relating to activities.

### SimulationMode

The SimulationMode property is used to determine the simulation mode that Rational Rhapsody should use when simulating activity diagrams. Set the value to TokenOriented to get the simulation behavior introduced in version 7.5.3. Set the value to StateOriented to get the statechart-based simulation behavior that was included in versions prior to 7.5.3.

*Default = TokenOriented*

### AutoSelectControlOrObjectFlow

When the property AutoSelectControlOrObjectFlow is set to True, Rational Rhapsody will automatically replace a control flow with an object flow, or vice versa, if you have used the wrong drawing tool to connect elements in an Activity.

*Default = True*

# Activity\_diagram

---

The Activity\_diagram subject contains metaclasses that contain properties for controlling the activity diagram editor. Some of the metaclasses and their properties are for backwards compatibility with previous versions of Rational Rhapsody.

## AcceptEventAction

The AcceptEventAction metaclass contains properties that affect the appearance of accept event actions in activity diagrams.

### ShowNotation

The ShowNotation property determines what text is opened on Accept Event Action elements in an activity diagram. The possible values are:

- Name - The name of the element is displayed.
- Label - The label of the element is displayed.
- Event - The name of the event selected is displayed.
- FullNotation - In addition to the name of the event selected, Rational Rhapsody displays the name of the target selected and the argument values you provided.

Note: This property can only be set at the diagram level or higher and not at the level of individual Accept Event Action elements.

When you change the value of this property, the display of any new Accept Event Action elements are affected, but the display of Accept Event Action elements already on the diagram remains as is.

*(Default = FullNotation)*

### ShowStereotype

The ShowStereotype property determines if, and how, an stereotypes of the element are displayed in a diagram. The possible values are:

- Label - The stereotypes of the element are displayed as a text label.
- Bitmap - The bitmap image that is associated with the stereotype of the element is displayed.
- None - The stereotypes of the element are not displayed.

Note that this property can only be set at the diagram level or higher (not at the level of individual Accept Event Action elements).

When you change the value of this property, the display of any new Accept Event Action elements are affected, but the display of Accept Event Action elements already on the diagram remains as is.

*(Default = None)*

## AcceptTimeEvent

This metaclass contains properties that affect the appearance of accept time events in activity diagrams.

### ShowName

The ShowName property specifies how the name of a time event should be displayed.

The possible values are:

- Name - Display the name of the event.
- Duration - Display the length of time for the event.
- Label - Display the label of the time event.
- None - Do not display the name, label, or duration of the time event.

*(Default = Duration)*

## Action

The Action metaclass contains properties to control the appearance of actions in activity diagrams.

### ShowAction

The ShowAction property controls what is displayed inside the action block. The possible values are:

- Action - Display the name of the action
- Description - Display the description of the action
- Label - Display the label of the action

*(Default = Action)*

### ShowName

The ShowName property specifies how the name of an object should be displayed.

The possible values are:

- Name - Display the name of the action.

- Label - Display the label of the action.
- None - Display neither the name nor label for the action.

*(Default = None)*

## ShowStereotype

The ShowStereotype property determines if, and how, a stereotype of the element are displayed in a diagram. The possible values are:

- Label - The stereotype of the element are displayed as a text label.
- Bitmap - The bitmap image that is associated with the stereotype of the element are displayed.
- None - The stereotype of the element are not displayed.

The default value for this property varies for the different elements. For the following subject::metaclass combinations, the default value is None:

- Statechart::DefaultTransition
- Activity\_diagram::Action
- Activity\_diagram::ActionBlock
- Activity\_diagram::ActionPin
- Activity\_diagram::DefaultTransition
- Activity\_diagram::ReferenceActivity

For the following subject::metaclass combinations, the default value is Label:

- Statechart::Requirement
- Statechart::Depends
- ObjectModelGe::Depends
- ObjectModelGe::Inheritance
- ObjectModelGe::Actor
- UseCaseGe::Depends
- UseCaseGe::Inheritance
- Activity\_diagram::Depends
- Activity\_diagram::SubActivityState

## ActionBlock

The ActionBlock metaclass contains a property to control the appearance of action blocks in activity diagrams.

## ShowName

The ShowName property specifies how the name of an object should be displayed.

The possible values are:

- Name - Display the name of the action.
- Label - Display the label of the action.
- None - Display neither the name nor label for the action.

*(Default = None)*

## ShowStereotype

The ShowStereotype property determines if, and how, a stereotype of the element are displayed in a diagram. The possible values are:

- Label - The stereotype of the element are displayed as a text label.
- Bitmap - The bitmap image that is associated with the stereotype of the element are displayed.
- None - The stereotype of the element are not displayed.

The default value for this property varies for the different elements. For the following subject::metaclass combinations, the default value is None:

- Statechart::DefaultTransition
- Activity\_diagram::Action
- Activity\_diagram::ActionBlock
- Activity\_diagram::DefaultTransition
- Activity\_diagram::ReferenceActivity

For the following subject::metaclass combinations, the default value is Label:

- Statechart::Requirement
- Statechart::Depends
- ObjectModelGe::Depends
- ObjectModelGe::Inheritance
- ObjectModelGe::Actor
- UseCaseGe::Depends
- UseCaseGe::Inheritance
- Activity\_diagram::Depends
- Activity\_diagram::SubActivityState

## ActionPin

The ActionPin metaclass contains properties that control the appearance of action pins in activity

diagrams.

## ShowName

The ShowName property determines what text is opened alongside Action Pin elements. The possible values are:

- Name - The name of the element is displayed.
- NameAndType - Both the name and the type (for example, int) are displayed.
- Type - The type (for example, int) is displayed.
- Label - The label of the element is displayed.

Note that this property can only be set at the diagram level or higher (not at the level of individual Action Pin elements).

When you change the value of this property, the display of any new Action Pin elements are affected, but the display of Action Pin elements already on the diagram remains as is.

*Default = Name*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = None*

# ActivityParameter

The ActivityParameter metaclass contains properties determine how to display the name and stereotype.

## ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values for this property varies for the different elements, as does the default value used. The different values used are:

- Description - the content of the description field; relevant for elements such as comments

- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class are displayed with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

The various combinations of possible values are:

- Name\_only, Label (default is Name\_only)
- Used for following subject::metaclass combinations:
  - Statechart::State
- Name, Label, None (default is Name)
- Used for following subject::metaclass combinations:
  - Statechart::Transition
  - Statechart::DefaultTransition
  - ObjectModelGe::Aggregation
  - ObjectModelGe::Composition
  - ObjectModelGe::Association
  - ObjectModelGe::Link
  - UseCaseGe::Association
  - Activity\_diagram::Transition
  - Activity\_diagram::DefaultTransition
  - Activity\_diagram::ReferenceActivity
  - Activity\_diagram::SubActivityState
  - Activity\_diagram::ActivityParameter
- Name, Label, None (default is None)
- Used for following subject::metaclass combinations:
  - ObjectModelGe::Inheritance
  - ObjectModelGe::Depends
  - UseCaseGe::Depends
  - UseCaseGe::Inheritance
  - Activity\_diagram::ActivityParameter
  - Activity\_diagram::Depends
  - Statechart::Depends
- Full\_path, Relative, Name\_only, Label (default is Relative)



- Used for following subject::metaclass combinations:
- Statechart::Requirement
- ObjectModelGe::Class
- ObjectModelGe::Object
- ObjectModelGe::UseCase
- ObjectModelGe::Actor
- ObjectModelGe::Requirement
- UseCaseGe::Actor
- UseCaseGe::Association
- UseCaseGe::Requirement
- UseCaseGe::UseCase
- Activity\_diagram::Requirement
- Name, Description, Label (default is Description)
- Used for following subject::metaclass combinations:
- Statechart::Comment
- ObjectModelGe::Comment
- UseCaseGe::Comment
- Activity\_diagram::Comment
- Name, Specification, Description, Label (default is Description)
- Used for following subject::metaclass combinations:
- Statechart::Constraint
- ObjectModelGe::Constraint
- UseCaseGe::Constraint
- Activity\_diagram::Constraint
- Full\_path, Relative, Name\_only, Label (default is Name\_only)
- Used for following subject::metaclass combinations:
- ObjectModelGe::Package
- ObjectModelGe::PrimitiveOperation
- ObjectModelGe::Attribute
- ObjectModelGe::Type
- Name, Label (default is Name)
- Used for following subject::metaclass combinations:
- Activity\_diagram::Swimlane

## ShowStereotype

The ShowStereotype property determines if, and how, a stereotype of the element are displayed in a diagram. The possible values are:

- Label - The stereotype of the element are displayed as a text label.
- Bitmap - The bitmap image that is associated with the stereotype of the element are displayed.
- None - The stereotype of the element are not displayed.

The default value for this property varies for the different elements. For the following subject::metaclass combinations, the default value is None:

- Statechart::DefaultTransition
- Activity\_diagram::Action
- Activity\_diagram::ActionBlock
- Activity\_diagram::DefaultTransition
- Activity\_diagram::ReferenceActivity

For the following subject::metaclass combinations, the default value is Label:

- Statechart::Requirement
- Statechart::Depends
- ObjectModelGe::Depends
- ObjectModelGe::Inheritance
- ObjectModelGe::Actor
- UseCaseGe::Depends
- UseCaseGe::Inheritance
- Activity\_diagram::Depends
- Activity\_diagram::SubActivityState

## AutoPopulate

The AutoPopulate metaclass contains properties that can be used to control the appearance of diagrams that are drawn automatically by Rational Rhapsody.

### ArrowDirection

The ArrowDirection property is used when Rational Rhapsody automatically generates a diagram, and it determines whether the flow of connectors in the diagram runs from top to bottom or bottom to top.

There are two situations where Rational Rhapsody automatically generates diagrams:

- If you have selected the Populate Diagrams option for Reverse Engineering (for those diagrams where this feature is supported).
- If you double-click a diagram in the browser that was generated by using the Rational Rhapsody API.

*Default = Top-Bottom*

## LayoutStyle

The LayoutStyle property is used when Rational Rhapsody automatically generates a diagram, and it determines the general appearance of the diagram - hierarchical or orthogonal.

- Hierarchical - diagram layout reflects a hierarchy, appropriate for relationships such as inheritance
- Orthogonal - diagram layout resembles a grid, appropriate where there are no clear hierarchical relationships between the elements in the diagram

There are two situations where Rational Rhapsody automatically generates diagrams:

- If you have selected the Populate Diagrams option for Reverse Engineering (for those diagrams where this feature is supported).
- If you double-click in the browser a diagram that was generated by using the Rational Rhapsody API.

*Default = Hierarchical*

## ButtonArray

The ButtonArray metaclass contains properties that determine the appearance and behavior of button array controls on activity diagrams.

### ButtonFont

The ButtonFont property lets you select the font to use for the text on the face of a push button control.

To change the value of the property, click the "..." button in the box next to the property value to open the Font window. The value of the property affects both buttons already on the activity diagram and new buttons added to the diagram. (The display of buttons already on the diagram changes only after you refresh the diagram.)

*Default = Arial 10 NoBold NoItalic*

### Direction

The Direction property determines whether the button array controls are used to input data, display data, or both. The possible values are:

- In - The button arrays are only used to input data for the attribute to which it is bound.
- Out - The button arrays are only used to display data for the attribute to which it is bound.
- InOut - The button arrays are used to input data and display data.

*Default = InOut*

## ShowName

The ShowName property determines whether or not a caption is displayed for button array elements, and if so, what text should be displayed. The possible values are:

- **BindedElementFullPath** - The full path of the attribute that is bound to the button array.
- **BindedElement** - The name of the attribute that is bound to the button array.
- **Name** - The name of the button array element.
- **None** - No text is displayed.

*Default = Name*

## CallBehavior

The CallBehavior metaclass represents a call to an operation of a classifier. This behavior is only used in modeling and does not generate code for the call operation.

## ShowDescription

The ShowDescription property specifies whether descriptions for the call behavior in the activity diagram should be displayed.

*Default = Cleared*

## ShowName

The ShowName property determines what text is opened at the top of Call Behavior elements. The possible values are:

- **Name** - The name of the element is displayed.
- **Label** - The label of the element is displayed.
- **None** - Nothing is displayed at the top of the element.

*Default = Name*

## ShowStereotype

The ShowStereotype property determines if, and how, an stereotypes of the element are displayed in a diagram. The possible values are:

- **Label** - The stereotypes of the element are displayed as a text label.
- **Bitmap** - The bitmap image that is associated with the stereotype of the element is displayed.
- **None** - The stereotypes of the element are not displayed.

*Default = None*

## CallOperation

The CallOperation metaclass contains properties that relate to Call Operation elements in activity diagrams.

### ShowAction

The ShowAction property determines what is opened on Call Operation elements in an activity diagram. The possible values are:

- Action - The code entered in the Action text box is displayed.
- Description - The description entered for the element is displayed.
- Label - The label entered for the element is displayed.

Note that this property can only be set at the diagram level or higher (not at the level of individual Call Operation elements). When you change the value of this property, the display of any new Call Operation elements are affected, but the display of Call Operation elements already on the diagram remains as is.

*Default = Action*

### ShowName

The ShowName property determines what text is opened at the top of Call Operation elements. The possible values are:

- Name - The name of the element is displayed.
- Label - The label of the element is displayed.
- Operation - The operation that is being called is displayed.
- FullNotation - In addition to the operation name, the arguments and return value of the operation are displayed.
- None - Nothing is displayed at the top of the element.

Note that this property can only be set at the diagram level or higher (not at the level of individual Call Operation elements). When you change the value of this property, the display of any new Call Operation elements are affected, but the display of Call Operation elements already on the diagram remains as is.

*Default = Operation*

### ShowStereotype

The ShowStereotype property determines if, and how, an stereotypes of the element are displayed in a diagram. The possible values are:

- **Label** - The stereotypes of the element are displayed as a text label.
- **Bitmap** - The bitmap image that is associated with the stereotype of the element is displayed.
- **None** - The stereotypes of the element are not displayed.

Note that this property can only be set at the diagram level or higher (not at the level of individual Call Operation elements). When you change the value of this property, the display of any new Call Operation elements are affected, but the display of Call Operation elements already on the diagram remains as is.

*Default = Label*

## Comment

The Comment metaclass contains properties that control the appearance of comments in activity diagrams.

### CommentNotation

The CommentNotation property determines how Comment elements are displayed. This property can be set to one of two styles:

- **Note\_Style**
- **Box\_Style**

If CommentNotation is set to Note\_Style, then the appearance of the element is determined by the value selected for the property Comment::ShowForm: Note, Plain, or PushPin.

If CommentNotation is set to Box\_Style, then the element resembles a class-box, and you can control the display of compartments for the element.

*Default = Note\_Style*

### Compartments

The Compartments property determines which of the available compartments are displayed by default for the various types of elements. The value for this property is a comma-delimited string containing the names of the compartments that should be visible.

Since the available compartments vary from element to element, it is recommended that you do not try to set the value of this property by using the Properties window or directly in the .prp file. Rather, you should use an element's Display Options to set which compartments are visible, and then use the Make Default option to apply these settings at the diagram or project level for new elements of this type.

*Default = Blank*

### color

The color property specifies the default color of the border of a graphical item, such as an object box.

*(Default = 128,128,0)*

## **Fillcolor**

The Fillcolor property specifies the default fill color for the object.

*(Default = 0,255,255)*

## **line\_width**

The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines).

*(Default = 1)*

## **name\_color**

The name\_color property specifies the default color of names of graphical items.

*(Default = 0,0,0)*

## **QuickNavigationCategories**

Graphic elements in diagrams can include a list of shortcuts that are accessible from the title bar of the element. You can use the property QuickNavigationCategories to specify the categories of items that should be included in this shortcut list and the order in which the categories should appear in the list. The value of the property should be a comma-separated list of categories. The categories that can be included are:

- Activities
- DiagramReferences
- Diagrams
- Hyperlinks
- InternalBlockDiagrams
- MainBehavior
- MainDiagram
- References
- StateCharts

If you want greater control over the content of the navigation list (such as showing/hiding specific elements within a given category, or changing the order of elements within a specific category), use the Compartments list, which can be found under "Display Options".

*Default = Hyperlinks*

## ShowAnnotationContents

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the text displayed on the element in the diagram determined by the value that was set for the property ShowAnnotationContents. The property can take one of the following values:

- Name - the name of the element
- Label - the label assigned to the element
- Description - the content of the Description field
- Specification (for Requirements only) - the content of the Specification field

*Default = Description*

## ShowForm

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the appearance of the element is determined by the value that was set for the property ShowForm.

The possible values for this property vary for the different elements, as does the default value used. The values used are:

- Plain - No color background behind text
- Note - Color background behind text
- Pushpin - Color background plus pin icon

*Default = Note*

## ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values for this property varies for the different elements, as does the default value used. The different values used are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information



that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class are displayed with the class name only if the class is not positioned inside the package in the diagram.

- Specification - the content of the specification field; relevant for elements such as constraints

The various combinations of possible values are:

- Name\_only, Label (default is Name\_only)
- Used for following subject::metaclass combinations:
  - Statechart::State
- Name, Label, None (default is Name)
- Used for following subject::metaclass combinations:
  - Statechart::Transition
  - Statechart::DefaultTransition
  - ObjectModelGe::Aggregation
  - ObjectModelGe::Composition
  - ObjectModelGe::Association
  - ObjectModelGe::Link
  - UseCaseGe::Association
  - Activity\_diagram::Transition
  - Activity\_diagram::DefaultTransition
  - Activity\_diagram::ReferenceActivity
  - Activity\_diagram::SubActivityState
  - Activity\_diagram::ActivityParameter
- Name, Label, None (default is None)
- Used for following subject::metaclass combinations:
  - ObjectModelGe::Inheritance
  - ObjectModelGe::Depends
  - UseCaseGe::Depends
  - UseCaseGe::Inheritance
  - Activity\_diagram::ActivityParameter
  - Activity\_diagram::Depends
  - Statechart::Depends
- Full\_path, Relative, Name\_only, Label (default is Relative)
- Used for following subject::metaclass combinations:
  - Statechart::Requirement
  - ObjectModelGe::Class
  - ObjectModelGe::Object
  - ObjectModelGe::UseCase
  - ObjectModelGe::Actor

- ObjectModelGe::Requirement
- UseCaseGe::Actor
- UseCaseGe::Association
- UseCaseGe::Requirement
- UseCaseGe::UseCase
- Activity\_diagram::Requirement
- Name, Description, Label (default is Description)
- Used for following subject::metaclass combinations:
- Statechart::Comment
- ObjectModelGe::Comment
- UseCaseGe::Comment
- Activity\_diagram::Comment
- Name, Specification, Description, Label (default is Description)
- Used for following subject::metaclass combinations:
- Statechart::Constraint
- ObjectModelGe::Constraint
- UseCaseGe::Constraint
- Activity\_diagram::Constraint
- Full\_path, Relative, Name\_only, Label (default is Name\_only)
- Used for following subject::metaclass combinations:
- ObjectModelGe::Package
- ObjectModelGe::PrimitiveOperation
- ObjectModelGe::Attribute
- ObjectModelGe::Type
- Name, Label (default is Name)
- Used for following subject::metaclass combinations:
- Activity\_diagram::Swimlane

## ShowStereotype

The ShowStereotype property determines if, and how, a stereotype of the element are displayed in a diagram. The possible values are:

- Label - The stereotype of the element are displayed as a text label.
- Bitmap - The bitmap image that is associated with the stereotype of the element are displayed.
- None - The stereotype of the element are not displayed.

The default value for this property varies for the different elements. For the following subject::metaclass combinations, the default value is None:

- Statechart::DefaultTransition

- Activity\_diagram::Action
- Activity\_diagram::ActionBlock
- Activity\_diagram::DefaultTransition
- Activity\_diagram::ReferenceActivity

For the following subject::metaclass combinations, the default value is Label:

- Statechart::Requirement
- Statechart::Depends
- ObjectModelGe::Depends
- ObjectModelGe::Inheritance
- ObjectModelGe::Actor
- UseCaseGe::Depends
- UseCaseGe::Inheritance
- Activity\_diagram::Depends
- Activity\_diagram::SubActivityState

## Complete

The metaclass Complete contains properties that determine whether or not Rational Rhapsody automatically draws the relations that exist between an element added to a diagram and elements already on the diagram.

### Complete\_Relation

The Complete\_Relation property is used to specify that when an element is added to a diagram, Rational Rhapsody should automatically draw the relations that exist between the element and elements already on the diagram.

*Default = Cleared*

## ConnectorText

The ConnectorText metaclass contains properties that control the appearance of text inside connectors in activity diagrams.

### color

The color property specifies the default color of the border of a graphical item, such as an object box.

*(Default = 0,0,0)*

## Constraint

The Constraint metaclass contains properties that control the appearance of constraints in activity diagrams.

### Compartments

The Compartments property determines which of the available compartments are displayed by default for the various types of elements. The value for this property is a comma-delimited string containing the names of the compartments that should be visible.

Since the available compartments vary from element to element, it is recommended that you do not try to set the value of this property by using the Properties window or directly in the .prp file. Rather, you should use an element's Display Options to set which compartments are visible, and then use the Make Default option to apply these settings at the diagram or project level for new elements of this type.

*Default = Specification*

### ConstraintNotation

The ConstraintNotation property determines how Constraint elements are displayed. This property can be set to one of two styles:

- Note\_Style
- Box\_Style

If ConstraintNotation is set to Note\_Style, then the appearance of the element is determined by the value selected for the property Constraint::ShowForm: Note, Plain, or PushPin.

If ConstraintNotation is set to Box\_Style, then the element resembles a class-box, and you can control the display of compartments for the element.

*Default = Note\_Style*

### QuickNavigationCategories

Graphic elements in diagrams can include a list of shortcuts that are accessible from the title bar of the element. You can use the property QuickNavigationCategories to specify the categories of items that should be included in this shortcut list and the order in which the categories should appear in the list. The value of the property should be a comma-separated list of categories. The categories that can be included are:

- Activities

- DiagramReferences
- Diagrams
- Hyperlinks
- InternalBlockDiagrams
- MainBehavior
- MainDiagram
- References
- StateCharts

If you want greater control over the content of the navigation list (such as showing/hiding specific elements within a given category, or changing the order of elements within a specific category), use the Compartments list, which can be found under "Display Options".

*Default = Hyperlinks*

## ShowAnnotationContents

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the text displayed on the element in the diagram determined by the value that was set for the property ShowAnnotationContents. The property can take one of the following values:

- Name - the name of the element
- Label - the label assigned to the element
- Description - the content of the Description field
- Specification (for Requirements only) - the content of the Specification field

*Default = Description*

## ShowForm

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the appearance of the element is determined by the value that was set for the property ShowForm.

The possible values for this property vary for the different elements, as does the default value used. The values used are:

- Plain - No color background behind text
- Note - Color background behind text
- Pushpin - Color background plus pin icon

*Default = Note*

## ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a

diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values for this property varies for the different elements, as does the default value used. The different values used are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class are displayed with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

The various combinations of possible values are:

- Name\_only, Label (default is Name\_only)
- Used for following subject::metaclass combinations:
  - Statechart::State
- Name, Label, None (default is Name)
- Used for following subject::metaclass combinations:
  - Statechart::Transition
  - Statechart::DefaultTransition
  - ObjectModelGe::Aggregation
  - ObjectModelGe::Composition
  - ObjectModelGe::Association
  - ObjectModelGe::Link
  - UseCaseGe::Association
  - Activity\_diagram::Transition
  - Activity\_diagram::DefaultTransition
  - Activity\_diagram::ReferenceActivity
  - Activity\_diagram::SubActivityState
  - Activity\_diagram::ActivityParameter
- Name, Label, None (default is None)
- Used for following subject::metaclass combinations:
  - ObjectModelGe::Inheritance
  - ObjectModelGe::Depends

- UseCaseGe::Depends
- UseCaseGe::Inheritance
- Activity\_diagram::ActivityParameter
- Activity\_diagram::Depends
- Statechart::Depends
- Full\_path, Relative, Name\_only, Label (default is Relative)
- Used for following subject::metaclass combinations:
- Statechart::Requirement
- ObjectModelGe::Class
- ObjectModelGe::Object
- ObjectModelGe::UseCase
- ObjectModelGe::Actor
- ObjectModelGe::Requirement
- UseCaseGe::Actor
- UseCaseGe::Association
- UseCaseGe::Requirement
- UseCaseGe::UseCase
- Activity\_diagram::Requirement
- Name, Description, Label (default is Description)
- Used for following subject::metaclass combinations:
- Statechart::Comment
- ObjectModelGe::Comment
- UseCaseGe::Comment
- Activity\_diagram::Comment
- Name, Specification, Description, Label (default is Description)
- Used for following subject::metaclass combinations:
- Statechart::Constraint
- ObjectModelGe::Constraint
- UseCaseGe::Constraint
- Activity\_diagram::Constraint
- Full\_path, Relative, Name\_only, Label (default is Name\_only)
- Used for following subject::metaclass combinations:
- ObjectModelGe::Package
- ObjectModelGe::PrimitiveOperation
- ObjectModelGe::Attribute
- ObjectModelGe::Type
- Name, Label (default is Name)
- Used for following subject::metaclass combinations:

- Activity\_diagram::Swimlane

## ShowStereotype

The ShowStereotype property determines if, and how, a stereotype of the element are displayed in a diagram. The possible values are:

- Label - The stereotype of the element are displayed as a text label.
- Bitmap - The bitmap image that is associated with the stereotype of the element are displayed.
- None - The stereotype of the element are not displayed.

The default value for this property varies for the different elements. For the following subject::metaclass combinations, the default value is None:

- Statechart::DefaultTransition
- Activity\_diagram::Action
- Activity\_diagram::ActionBlock
- Activity\_diagram::DefaultTransition
- Activity\_diagram::ReferenceActivity

For the following subject::metaclass combinations, the default value is Label:

- Statechart::Requirement
- Statechart::Depends
- ObjectModelGe::Depends
- ObjectModelGe::Inheritance
- ObjectModelGe::Actor
- UseCaseGe::Depends
- UseCaseGe::Inheritance
- Activity\_diagram::Depends
- Activity\_diagram::SubActivityState

## ControlFlow

The ControlFlow metaclass contains properties that control the appearance of control flows in activity diagrams.

### line\_style

The line\_style property specifies the default line style for a graphical item.

The possible values are:



- `straight_arrows` - straight line.
- `rectilinear_arrows` - rectilinear line with right-angled corners placed at appropriate locations, depending on the starting and ending points of the line
- `spline_arrows` - curved line without corners
- `rounded_rectilinear_arrows` - similar to ordinary rectilinear lines but use rounded bends rather than straight ninety-degree angles

*Default = `spline_arrows`*

*Default = `spline_arrows`*

## ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values are:

- Name - the name of the element
- Label - the label provided for the element
- None - nothing should be displayed

*Default = `Name`*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams.

The possible values are:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = `None`*

# DiagramFrame

The DiagramFrame metaclass contains properties that control the appearance of a diagram frame.

## DefaultSize

The DefaultSize property specifies default values for diagram frame dimensions.

*Default = 20, 20, 590, 500*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## DecisionNode

The DecisionNode metaclass contains a property that controls the appearance of condition connectors in activity diagrams.

## CaptionBehavior

The CaptionBehavior property provides you with a certain degree of flexibility in terms of displaying text for a conditional connector.

By default, the value of the property is set to Fixed. This means that the amount of text displayed depends upon the size of the connector. If you have a lot of text and want it all to be displayed, you have to enlarge the connector in order to enlarge the text display area.

You can get around this limitation by setting the value of the property to Floating. This provides you with separate controls that can be used to enlarge the text display area without affecting the size of the connector itself.

In addition to the different behavior in terms of the size of the text display area, the value of the property also affects the position of the text. When set to Fixed, the text is always displayed at the center of the connector. When set to Floating, you can move the text anywhere you want relative to the position of the connector.

*Default = Fixed*

## show\_name

The show\_name property specifies how to label condition connectors in activity diagrams. The possible values are Name, Label, and None.

*(Default = Label)*

# DefaultTransition

The DefaultTransition metaclass controls the appearance of a default transition in an activity diagram.

## line\_style

The line\_style property specifies the default line style for a graphical item.

The possible values are:

- straight\_arrows - straight line.
- rectilinear\_arrows - rectilinear line with right-angled corners placed at appropriate locations, depending on the starting and ending points of the line
- spline\_arrows - curved line without corners
- rounded\_rectilinear\_arrows - similar to ordinary rectilinear lines but use rounded bends rather than straight ninety-degree angles

*Default = spline\_arrows*

*Default = straight\_arrows*

## ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values for this property varies for the different elements, as does the default value used. The different values used are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class are displayed with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

The various combinations of possible values are:

- Name\_only, Label (default is Name\_only)
- Used for following subject::metaclass combinations:
- Statechart::State
- Name, Label, None (default is Name)
- Used for following subject::metaclass combinations:
- Statechart::Transition
- Statechart::DefaultTransition
- ObjectModelGe::Aggregation
- ObjectModelGe::Composition
- ObjectModelGe::Association
- ObjectModelGe::Link
- UseCaseGe::Association
- Activity\_diagram::Transition
- Activity\_diagram::DefaultTransition
- Activity\_diagram::ReferenceActivity
- Activity\_diagram::SubActivityState
- Activity\_diagram::ActivityParameter
- Name, Label, None (default is None)
- Used for following subject::metaclass combinations:
- ObjectModelGe::Inheritance
- ObjectModelGe::Depends
- UseCaseGe::Depends
- UseCaseGe::Inheritance
- Activity\_diagram::ActivityParameter
- Activity\_diagram::Depends
- Statechart::Depends
- Full\_path, Relative, Name\_only, Label (default is Relative)
- Used for following subject::metaclass combinations:
- Statechart::Requirement
- ObjectModelGe::Class
- ObjectModelGe::Object
- ObjectModelGe::UseCase
- ObjectModelGe::Actor
- ObjectModelGe::Requirement
- UseCaseGe::Actor
- UseCaseGe::Association
- UseCaseGe::Requirement
- UseCaseGe::UseCase

- Activity\_diagram::Requirement
- Name, Description, Label (default is Description)
- Used for following subject::metaclass combinations:
- Statechart::Comment
- ObjectModelGe::Comment
- UseCaseGe::Comment
- Activity\_diagram::Comment
- Name, Specification, Description, Label (default is Description)
- Used for following subject::metaclass combinations:
- Statechart::Constraint
- ObjectModelGe::Constraint
- UseCaseGe::Constraint
- Activity\_diagram::Constraint
- Full\_path, Relative, Name\_only, Label (default is Name\_only)
- Used for following subject::metaclass combinations:
- ObjectModelGe::Package
- ObjectModelGe::PrimitiveOperation
- ObjectModelGe::Attribute
- ObjectModelGe::Type
- Name, Label (default is Name)
- Used for following subject::metaclass combinations:
- Activity\_diagram::Swimlane

## ShowStereotype

The ShowStereotype property determines if, and how, a stereotype of the element are displayed in a diagram. The possible values are:

- Label - The stereotype of the element are displayed as a text label.
- Bitmap - The bitmap image that is associated with the stereotype of the element are displayed.
- None - The stereotype of the element are not displayed.

The default value for this property varies for the different elements. For the following subject::metaclass combinations, the default value is None:

- Statechart::DefaultTransition
- Activity\_diagram::Action
- Activity\_diagram::ActionBlock
- Activity\_diagram::DefaultTransition
- Activity\_diagram::ReferenceActivity

For the following subject::metaclass combinations, the default value is Label:

- Statechart::Requirement
- Statechart::Depends
- ObjectModelGe::Depends
- ObjectModelGe::Inheritance
- ObjectModelGe::Actor
- UseCaseGe::Depends
- UseCaseGe::Inheritance
- Activity\_diagram::Depends
- Activity\_diagram::SubActivityState

## DependentText

The DependentText metaclass contains a property that controls the appearance of dependent text in activity diagrams.

### color

The color property specifies the default color of the border of a graphical item, such as an object box.

*(Default = 0,0,255)*

## Depends

The Depends metaclass contains properties that control the appearance of dependency relation lines in collaboration diagrams.

### line\_style

The line\_style property specifies the default line style for a graphical item.

The possible values are:

- straight\_arrows - straight line.
- rectilinear\_arrows - rectilinear line with right-angled corners placed at appropriate locations, depending on the starting and ending points of the line
- spline\_arrows - curved line without corners
- rounded\_rectilinear\_arrows - similar to ordinary rectilinear lines but use rounded bends rather than straight ninety-degree angles

*Default = spline\_arrows*

*Default = straight\_arrows*

## ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values for this property varies for the different elements, as does the default value used. The different values used are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class are displayed with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

The various combinations of possible values are:

- Name\_only, Label (default is Name\_only)
- Used for following subject::metaclass combinations:
  - Statechart::State
- Name, Label, None (default is Name)
- Used for following subject::metaclass combinations:
  - Statechart::Transition
  - Statechart::DefaultTransition
  - ObjectModelGe::Aggregation
  - ObjectModelGe::Composition
  - ObjectModelGe::Association
  - ObjectModelGe::Link
  - UseCaseGe::Association
  - Activity\_diagram::Transition
  - Activity\_diagram::DefaultTransition
  - Activity\_diagram::ReferenceActivity
  - Activity\_diagram::SubActivityState
  - Activity\_diagram::ActivityParameter

- Name, Label, None (default is None)
- Used for following subject::metaclass combinations:
- ObjectModelGe::Inheritance
- ObjectModelGe::Depends
- UseCaseGe::Depends
- UseCaseGe::Inheritance
- Activity\_diagram::ActivityParameter
- Activity\_diagram::Depends
- Statechart::Depends
- Full\_path, Relative, Name\_only, Label (default is Relative)
- Used for following subject::metaclass combinations:
- Statechart::Requirement
- ObjectModelGe::Class
- ObjectModelGe::Object
- ObjectModelGe::UseCase
- ObjectModelGe::Actor
- ObjectModelGe::Requirement
- UseCaseGe::Actor
- UseCaseGe::Association
- UseCaseGe::Requirement
- UseCaseGe::UseCase
- Activity\_diagram::Requirement
- Name, Description, Label (default is Description)
- Used for following subject::metaclass combinations:
- Statechart::Comment
- ObjectModelGe::Comment
- UseCaseGe::Comment
- Activity\_diagram::Comment
- Name, Specification, Description, Label (default is Description)
- Used for following subject::metaclass combinations:
- Statechart::Constraint
- ObjectModelGe::Constraint
- UseCaseGe::Constraint
- Activity\_diagram::Constraint
- Full\_path, Relative, Name\_only, Label (default is Name\_only)
- Used for following subject::metaclass combinations:
- ObjectModelGe::Package
- ObjectModelGe::PrimitiveOperation



- ObjectModelGe::Attribute
- ObjectModelGe::Type
- Name, Label (default is Name)
- Used for following subject::metaclass combinations:
- Activity\_diagram::Swimlane

## ShowStereotype

The ShowStereotype property determines if, and how, a stereotype of the element are displayed in a diagram. The possible values are:

- Label - The stereotype of the element are displayed as a text label.
- Bitmap - The bitmap image that is associated with the stereotype of the element are displayed.
- None - The stereotype of the element are not displayed.

The default value for this property varies for the different elements. For the following subject::metaclass combinations, the default value is None:

- Statechart::DefaultTransition
- Activity\_diagram::Action
- Activity\_diagram::ActionBlock
- Activity\_diagram::DefaultTransition
- Activity\_diagram::ReferenceActivity

For the following subject::metaclass combinations, the default value is Label:

- Statechart::Requirement
- Statechart::Depends
- ObjectModelGe::Depends
- ObjectModelGe::Inheritance
- ObjectModelGe::Actor
- UseCaseGe::Depends
- UseCaseGe::Inheritance
- Activity\_diagram::Depends
- Activity\_diagram::SubActivityState

## color

The color property specifies the default color of the border of a graphical item, such as an object box.

*(Default = 0,128,64)*

## UseFillColor

The UseFillColor property is a Boolean value that specifies whether to use the fill color specified for that type of connector or port.

*(Default = False)*

## DiagramConnector

The DiagramConnector metaclass contains properties that control the appearance of diagram connectors in activity diagrams.

### color

The color property specifies the default color of the border of a graphical item, such as an object box.

*(Default = 0,128,255)*

### text\_color

The text\_color property specifies the default text color.

*(Default = 255,255,255)*

## DiagramFrame

Contains properties relating to the appearance of the diagram frame.

### ShowName

The ShowName property determines the text to display next to a graphic element in a diagram. For most elements, you can provide a name and a label. Your descriptive label can be useful in cases where the name itself might not be sufficient due to various constraints, such as the inability to use spaces if the name of the element is to appear in the code.

The possible values for this property vary for the different elements, as does the default value used. The possible values are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element

- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is displayed with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

Note that for some elements, such as destruction elements in sequence diagrams and timing diagrams, the property is a boolean property that hides/shows the name of the element.

*Default = Name*

## DigitalDisplay

The DigitalDisplay metaclass contains properties that determine the appearance and behavior of digital display controls on activity diagrams.

### ShowName

The ShowName property determines whether or not a caption is displayed for digital display elements, and if so, what text should be displayed. The possible values are:

- BindedElementFullPath - The full path of the attribute that is bound to the digital display.
- BindedElement - The name of the attribute that is bound to the digital display.
- Name - The name of the digital display element.
- None - No text is displayed.

*Default = Name*

## Gauge

The Gauge metaclass contains properties that determine the appearance and behavior of gauge controls on activity diagrams.

### ShowName

The ShowName property determines whether or not a caption is displayed for gauge elements, and if so, what text should be displayed. The possible values are:

- BindedElementFullPath - The full path of the attribute that is bound to the gauge.
- BindedElement - The name of the attribute that is bound to the gauge.

- Name - The name of the gauge element.
- None - No text is displayed.

*Default = Name*

## General

The General metaclass contains a property that controls the general behavior of activity diagrams.

### AllowTriggerAndActionOnFlow

This property determines whether or not to allow a trigger and action on a flow.

*Default = Cleared*

### AutoOpenUponCreation

By default, when you create an activity diagram, it is added to the model browser, and the new diagram is displayed in the graphic editor. If you do not want the new diagram to be displayed in the graphic editor, set the value of the property AutoOpenUponCreation to False.

*Default = True*

### DefaultMode

The Features window for activity diagrams allows you to specify that an activity diagram should be "Analysis Only". This means that the diagram is for modeling purposes only and no code should be generated for the diagram. Certain types of model elements can only be included in "Analysis Only" activity diagrams.

The DefaultMode property can be used to specify the default value for this setting. The property can take the following values:

- Design - New activity diagrams are created in Design mode by default, that is, code is generated for the diagram
- Analysis - New activity diagrams are created in Analysis mode by default, that is, code is not generated for the diagram

*Default = Design*

### DeleteConfirmation

The DeleteConfirmation property specifies whether confirmation is required before deleting a graphical element from the model. Note that this property does not apply to statechart elements, which have a

separate DeleteConfirmation property.

The possible values are:

- Always - Rational Rhapsody displays a confirmation window each time you try to delete an item from the model.
- Never - Confirmation is not required to delete an element.
- WhenNeeded - Asks for confirmation if there are references to the element.

*(Default = Never)*

## SwimlaneHeadingsDisplayLevel

In activity diagrams that contain swimlanes, Rhapsody includes a bar below the title of the diagram, which shows the names of the swimlanes. This feature helps you keep track of the swimlane you are in even if you scroll down to the point where the names at the top of the lanes are no longer visible.

For diagrams where individual swimlanes have been decomposed to contain a second level of swimlanes, the property SwimlaneHeadingsDisplayLevel can be used to control the level of the lane headings that are displayed at the top. If the value of the property is set to High, only the names of the top-level swimlanes are displayed. If the value of the property is set to Low, the names of the lowest level of lanes are displayed.

*Default = High*

## ShowSwimlaneHeadings

The ShowSwimlaneHeadings property is used to specify that Rational Rhapsody should display swimlane headings at the top of an activity diagram. This is useful when you are working with long swimlanes where the top of the swimlane disappears when you scroll down.

For individual diagrams, you can control the display of these headings by using the context menu items Show Swimlane Headings, Hide Swimlane Headings.

*Default = Checked*

## SysMLDiagramKindTitle

When a diagram is created in SysML projects, the diagram frame contains a header for the diagram that includes information such as the name of the package that contains the diagram. The property SysMLDiagramKindTitle is used to specify a prefix for the diagram frame header that reflects the type of diagram. For example, when you create an internal block diagram, the header in the diagram frame starts with the prefix "ibd".

Note that the structure of the entire header is controlled by the property General::Graphics::DiagramHeader.

*Default = act*

# HistoryConnector

The HistoryConnector metaclass contains properties that control the appearance of history connectors in activity diagrams.

## color

The color property specifies the default color of the border of a graphical item, such as an object box.

*(Default = 0,128,255)*

## UseFillColor

The UseFillColor property is a Boolean value that specifies whether to use the fill color specified for that type of connector or port.

*(Default = True)*

# interruptibleRegion

The property identifies an activity group that supports termination of tokens flowing into portions of an activity.

## ShowName

The ShowName property determines the text that should be displayed next to an element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values are:

- Name - the name of the element
- Label - the label provided for the element
- None - nothing should be displayed

*(Default = None)*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in the activity diagrams.

The possible values are:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*(Default = Label)*

## JunctionConnector

The JunctionConnector metaclass contains properties that control the appearance of junction connectors in activity diagrams.

### color

The color property specifies the default color of the border of a graphical item, such as an object box.

*(Default = 0,128,255)*

### UseFillColor

The UseFillColor property is a Boolean value that specifies whether to use the fill color specified for that type of connector or port.

*(Default = True)*

## Knob

The Knob metaclass contains properties that determine the appearance and behavior of knob controls on activity diagrams.

### Direction

The Direction property determines whether the knob controls are used to input data, display data, or both. The possible values are:

- In - The knobs are only used to input data for the attribute to which it is bound.
- Out - The knobs are only used to display data for the attribute to which it is bound.
- InOut - The knobs are used to input data and display data.

*Default = InOut*

## ShowName

The ShowName property determines whether or not a caption is displayed for knob elements, and if so, what text should be displayed. The possible values are:

- BindedElementFullPath - The full path of the attribute that is bound to the knob.
- BindedElement - The name of the attribute that is bound to the knob.
- Name - The name of the knob element.
- None - No text is displayed.

*Default = Name*

## Labels

The Labels metaclass contains a property that controls the appearance of labels in activity diagrams.

### color

The color property specifies the default color of the border of a graphical item, such as an object box.

*(Default = 0,0,255)*

## Led

The LED metaclass contains properties that determine the appearance and behavior of LED controls on activity diagrams.

## ShowName

The ShowName property determines whether or not a caption is displayed for LED elements, and if so, what text should be displayed. The possible values are:

- BindedElementFullPath - The full path of the attribute that is bound to the LED.
- BindedElement - The name of the attribute that is bound to the LED.
- Name - The name of the LED element.
- None - No text is displayed.

*Default = Name*



## LevelIndicator

The LevelIndicator metaclass contains properties that determine the appearance and behavior of level indicator controls on activity diagrams.

### ShowName

The ShowName property determines whether or not a caption is displayed for level indicator elements, and if so, what text should be displayed. The possible values are:

- BindedElementFullPath - The full path of the attribute that is bound to the level indicator.
- Name - The name of the level indicator element.
- None - No text is displayed.

*Default = Name*

## LoopTransition

The LoopTransition metaclass contains properties that control the appearance of loop transition lines in activity diagrams.

### color

The color property specifies the default color of the border of a graphical item, such as an object box.

*(Default = 0,128,64)*

### label\_color

The label\_color property is currently unused.

Activity\_Diagram::LoopTransition/Transition/0,127,255

Statechart::Transition,0,127,255

*(Default = 0, 127, 255)*

### line\_style

The line\_style property specifies the default line style for a graphical item.

The possible values are:

- `straight_arrows` - straight line.
- `rectilinear_arrows` - rectilinear line with right-angled corners placed at appropriate locations, depending on the starting and ending points of the line
- `spline_arrows` - curved line without corners
- `rounded_rectilinear_arrows` - similar to ordinary rectilinear lines but use rounded bends rather than straight ninety-degree angles

*Default = `spline_arrows`*

*Default = `spline_arrows`*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams.

The possible values are:

- `Label` - Show only the stereotype label (text).
- `Bitmap` - Show only the stereotype bitmap.
- `None` - Do not show stereotypes in diagrams.

*(Default = `None`)*

## MatrixDisplay

The MatrixDisplay metaclass contains properties that determine the appearance and behavior of matrix display controls on activity diagrams.

## ShowName

The ShowName property determines whether or not a caption is displayed for matrix display elements, and if so, what text should be displayed. The possible values are:

- `BindedElementFullPath` - The full path of the attribute that is bound to the matrix display.
- `BindedElement` - The name of the attribute that is bound to the matrix display.
- `Name` - The name of the matrix display element.
- `None` - No text is displayed.

*Default = `Name`*

## Meter

The Meter metaclass contains properties that determine the appearance and behavior of meter controls on activity diagrams.

### ShowName

The ShowName property determines whether or not a caption is displayed for meter elements, and if so, what text should be displayed. The possible values are:

- **BindedElementFullPath** - The full path of the attribute that is bound to the meter.
- **BindedElement** - The name of the attribute that is bound to the meter.
- **Name** - The name of the meter element.
- **None** - No text is displayed.

*Default = Name*

## Names

The Names metaclass contains a property that controls the appearance of activity names in activity diagrams.

### color

The color property specifies the default color of the border of a graphical item, such as an object box.

*(Default = 0,0,0)*

## Note

The Note metaclass contains properties that control the appearance of notes in activity diagrams.

### color

The color property specifies the default color of the border of a graphical item, such as an object box.

*(Default = 0,128,64)*

## Fillcolor

The Fillcolor property specifies the default fill color for the object.

*(Default = 0,255,255)*

## line\_width

The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines).

*(Default = 1)*

## name\_color

The name\_color property specifies the default color of names of graphical items.

*(Default = 0,128,255)*

## ShowForm

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the appearance of the element is determined by the value that was set for the property ShowForm.

The possible values for this property vary for the different elements, as does the default value used. The values used are:

- Plain - No color background behind text
- Note - Color background behind text
- Pushpin - Color background plus pin icon

*Default = Note*

## ObjectFlow

The ObjectFlow metaclass contains properties that control the appearance of object flows in activity diagrams.

## line\_style

The line\_style property specifies the default line style for a graphical item.

The possible values are:

- `straight_arrows` - straight line.
- `rectilinear_arrows` - rectilinear line with right-angled corners placed at appropriate locations, depending on the starting and ending points of the line
- `spline_arrows` - curved line without corners
- `rounded_rectilinear_arrows` - similar to ordinary rectilinear lines but use rounded bends rather than straight ninety-degree angles

*Default = `spline_arrows`*

*Default = `spline_arrows`*

## ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values are:

- Name - the name of the element
- Label - the label provided for the element
- None - nothing should be displayed

*Default = Name*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams.

The possible values are:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = None*

## ObjectFlowState

The ObjectFlowState metaclass contains properties that control the appearance of object flow states in activity diagrams.

## color

The color property specifies the default color of the border of a graphical item, such as an object box.

*(Default = 0,255,0)*

## Fillcolor

The Fillcolor property specifies the default fill color for the object.

*(Default = 0,255,255)*

## line\_width

The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines).

*(Default = 1)*

## name\_color

The name\_color property specifies the default color of names of graphical items.

*(Default = 0,0,0)*

# ObjectNode

The ObjectNode metaclass contains properties that relate to Object Node elements in activity diagrams.

## ShowInnerSlots

The property ShowInnerSlots can be used to determine the level of detail displayed for slots that represent complex data types in:

- Instance Specification elements in object model diagrams
- Object Node elements that are associated with Instance Specifications, in activities

The property can be set to one of the following values:

- None - only the value of the complex type represented by the slot is displayed, not the values of the individual primitive types that it contains
- ExcludeReferencesToOtherInstances - the values of the individual primitive types that make up the complex type represented by the slot are displayed, but not the value of the complex type itself

- All - both the value of the complex type and the values of the individual primitive types that make up the complex type are displayed

*Default = None*

## ShowInStatePath

The ShowInStatePath property determines how to show the hierarchy of states for nested Object Node elements.

*Default = Disabled*

## ShowName

The ShowName property determines what text is opened at the top of Object Node elements. The possible values are:

- Name\_only - Only the name of the element is displayed.
- Represents - Both the name of the element and the type it represents are displayed.
- RepresentsOnly - Only the type represented by the element is displayed. If no type was selected for the Represents field, the element name is displayed.
- Label - The label of the element is displayed.

Note that this property can only be set at the diagram level or higher (not at the level of individual Object Node elements). When you change the value of this property, the display of any new Object Node elements are affected, but the display of Object Node elements already on the diagram remains as is.

*Default = Represents*

## ShowStereotype

The ShowStereotype property determines if, and how, an stereotypes of the element are displayed in a diagram. The possible values are:

- Label - The stereotypes of the element are displayed as a text label.
- Bitmap - The bitmap image that is associated with the stereotype of the element is displayed.
- None - The stereotypes of the element are not displayed.

Note that this property can only be set at the diagram level or higher (not at the level of individual Call Operation elements). When you change the value of this property, the display of any new Call Operation elements are affected, but the display of Call Operation elements already on the diagram remains as is.

*Default = None*

## OnOffSwitch

The OnOffSwitch metaclass contains properties that determine the appearance and behavior of the on or off switch controls on activity diagrams.

## Direction

The Direction property determines whether the on/off switch controls are used to input data, display data, or both. The possible values are:

- In - The on/off switches are only used to input data for the attribute to which it is bound.
- Out - The on/off switches are only used to display data for the attribute to which it is bound.
- InOut - The on/off switches are used to input data and display data.

*Default = InOut*

## ShowName

The ShowName property determines whether or not a caption is displayed for on/off switch elements, and if so, what text should be displayed. The possible values are:

- BindedElementFullPath - The full path of the attribute that is bound to the on/off switch.
- BindedElement - The name of the attribute that is bound to the on/off switch.
- Name - The name of the on/off switch element.
- None - No text is displayed.

*Default = Name*

## Partition

The Partition metaclass contains properties that control the appearance of partitions (instance lines) in activity diagrams.

### color

The color property specifies the default color of the border of a graphical item, such as an object box.

*(Default = 0,0,255)*

### Fillcolor

The Fillcolor property specifies the default fill color for the object.

*(Default = 0,0,255)*



## **name\_color**

The name\_color property specifies the default color of names of graphical items.

*(Default = 0,0,255)*

## **PartitionFrame**

The PartitionFrame metaclass contains properties that control the appearance of partition frames in activity diagrams.

### **color**

The color property specifies the default color of the border of a graphical item, such as an object box.

*(Default = 0,0,255)*

### **line\_width**

The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines).

*(Default = 1)*

## **PushButton**

The PushButton metaclass contains properties that determine the appearance and behavior of push button controls on activity diagrams.

### **ButtonFont**

The ButtonFont property lets you select the font to use for the text on the face of a push button control.

To change the value of the property, click the "..." button in the box next to the property value to open the Font window. The value of the property affects both buttons already on the activity diagram and new buttons added to the diagram. (The display of buttons already on the activity diagram changes only after you refresh the diagram.)

*Default = Arial 10 NoBold NoItalic*

## ShowName

The ShowName property determines whether or not a caption is displayed for push button elements, and if so, what text should be displayed. The possible values are:

- BindedElementFullPath - The full path of the attribute that is bound to the push button.
- BindedElement - The name of the attribute that is bound to the push button.
- Name - The name of the push button element.
- None - No text is displayed.

*Default = Name*

## ReferenceActivity

The ReferenceActivity metaclass contains properties that control the appearance of references in activity diagrams.

## ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values for this property varies for the different elements, as does the default value used. The different values used are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class are displayed with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

The various combinations of possible values are:

- Name\_only, Label (default is Name\_only)
- Used for following subject::metaclass combinations:

- Statechart::State
- Name, Label, None (default is Name)
- Used for following subject::metaclass combinations:
- Statechart::Transition
- Statechart::DefaultTransition
- ObjectModelGe::Aggregation
- ObjectModelGe::Composition
- ObjectModelGe::Association
- ObjectModelGe::Link
- UseCaseGe::Association
- Activity\_diagram::Transition
- Activity\_diagram::DefaultTransition
- Activity\_diagram::ReferenceActivity
- Activity\_diagram::SubActivityState
- Activity\_diagram::ActivityParameter
- Name, Label, None (default is None)
- Used for following subject::metaclass combinations:
- ObjectModelGe::Inheritance
- ObjectModelGe::Depends
- UseCaseGe::Depends
- UseCaseGe::Inheritance
- Activity\_diagram::ActivityParameter
- Activity\_diagram::Depends
- Statechart::Depends
- Full\_path, Relative, Name\_only, Label (default is Relative)
- Used for following subject::metaclass combinations:
- Statechart::Requirement
- ObjectModelGe::Class
- ObjectModelGe::Object
- ObjectModelGe::UseCase
- ObjectModelGe::Actor
- ObjectModelGe::Requirement
- UseCaseGe::Actor
- UseCaseGe::Association
- UseCaseGe::Requirement
- UseCaseGe::UseCase
- Activity\_diagram::Requirement
- Name, Description, Label (default is Description)

- Used for following subject::metaclass combinations:
- Statechart::Comment
- ObjectModelGe::Comment
- UseCaseGe::Comment
- Activity\_diagram::Comment
- Name, Specification, Description, Label (default is Description)
- Used for following subject::metaclass combinations:
- Statechart::Constraint
- ObjectModelGe::Constraint
- UseCaseGe::Constraint
- Activity\_diagram::Constraint
- Full\_path, Relative, Name\_only, Label (default is Name\_only)
- Used for following subject::metaclass combinations:
- ObjectModelGe::Package
- ObjectModelGe::PrimitiveOperation
- ObjectModelGe::Attribute
- ObjectModelGe::Type
- Name, Label (default is Name)
- Used for following subject::metaclass combinations:
- Activity\_diagram::Swimlane

## ShowStereotype

The ShowStereotype property determines if, and how, a stereotype of the element are displayed in a diagram. The possible values are:

- Label - The stereotype of the element are displayed as a text label.
- Bitmap - The bitmap image that is associated with the stereotype of the element are displayed.
- None - The stereotype of the element are not displayed.

The default value for this property varies for the different elements. For the following subject::metaclass combinations, the default value is None:

- Statechart::DefaultTransition
- Activity\_diagram::Action
- Activity\_diagram::ActionBlock
- Activity\_diagram::DefaultTransition
- Activity\_diagram::ReferenceActivity

For the following subject::metaclass combinations, the default value is Label:

- Statechart::Requirement
- Statechart::Depends

- ObjectModelGe::Depends
- ObjectModelGe::Inheritance
- ObjectModelGe::Actor
- UseCaseGe::Depends
- UseCaseGe::Inheritance
- Activity\_diagram::Depends
- Activity\_diagram::SubActivityState

## Requirement

The Requirement metaclass contains properties that control the appearance of requirements in activity diagrams.

## Compartments

The Compartments property determines which of the available compartments are displayed by default for the various types of elements. The value for this property is a comma-delimited string containing the names of the compartments that should be visible.

Since the available compartments vary from element to element, it is recommended that you do not try to set the value of this property by using the Properties window or directly in the .prp file. Rather, you should use an element's Display Options to set which compartments are visible, and then use the Make Default option to apply these settings at the diagram or project level for new elements of this type.

*Default = ID, Specification*

## QuickNavigationCategories

Graphic elements in diagrams can include a list of shortcuts that are accessible from the title bar of the element. You can use the property QuickNavigationCategories to specify the categories of items that should be included in this shortcut list and the order in which the categories should appear in the list. The value of the property should be a comma-separated list of categories. The categories that can be included are:

- Activities
- DiagramReferences
- Diagrams
- Hyperlinks
- InternalBlockDiagrams
- MainBehavior
- MainDiagram
- References

- StateCharts

If you want greater control over the content of the navigation list (such as showing/hiding specific elements within a given category, or changing the order of elements within a specific category), use the Compartments list, which can be found under "Display Options".

*Default = Hyperlinks*

## RequirementNotation

The RequirementNotation property determines how Requirement elements are displayed. This property can be set to one of two styles:

- Note\_Style
- Box\_Style

If RequirementNotation is set to Note\_Style, then the appearance of the element is determined by the value selected for the property Requirement::ShowForm: Note, Plain, or PushPin.

If RequirementNotation is set to Box\_Style, then the element resembles a class-box, and you can control the display of compartments for the element.

*Default = Note\_Style*

## ShowAnnotationContents

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the text displayed on the element in the diagram determined by the value that was set for the property ShowAnnotationContents. The property can take one of the following values:

- Name - the name of the element
- Label - the label assigned to the element
- Description - the content of the Description field
- Specification (for Requirements only) - the content of the Specification field

*Default = Description*

## ShowForm

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the appearance of the element is determined by the value that was set for the property ShowForm.

The possible values for this property vary for the different elements, as does the default value used. The values used are:

- Plain - No color background behind text
- Note - Color background behind text
- Pushpin - Color background plus pin icon

## ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values for this property varies for the different elements, as does the default value used. The different values used are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class are displayed with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

The various combinations of possible values are:

- Name\_only, Label (default is Name\_only)
- Used for following subject::metaclass combinations:
  - Statechart::State
- Name, Label, None (default is Name)
- Used for following subject::metaclass combinations:
  - Statechart::Transition
  - Statechart::DefaultTransition
  - ObjectModelGe::Aggregation
  - ObjectModelGe::Composition
  - ObjectModelGe::Association
  - ObjectModelGe::Link
  - UseCaseGe::Association
  - Activity\_diagram::Transition
  - Activity\_diagram::DefaultTransition
  - Activity\_diagram::ReferenceActivity
  - Activity\_diagram::SubActivityState
  - Activity\_diagram::ActivityParameter

- Name, Label, None (default is None)
- Used for following subject::metaclass combinations:
- ObjectModelGe::Inheritance
- ObjectModelGe::Depends
- UseCaseGe::Depends
- UseCaseGe::Inheritance
- Activity\_diagram::ActivityParameter
- Activity\_diagram::Depends
- Statechart::Depends
- Full\_path, Relative, Name\_only, Label (default is Relative)
- Used for following subject::metaclass combinations:
- Statechart::Requirement
- ObjectModelGe::Class
- ObjectModelGe::Object
- ObjectModelGe::UseCase
- ObjectModelGe::Actor
- ObjectModelGe::Requirement
- UseCaseGe::Actor
- UseCaseGe::Association
- UseCaseGe::Requirement
- UseCaseGe::UseCase
- Activity\_diagram::Requirement
- Name, Description, Label (default is Description)
- Used for following subject::metaclass combinations:
- Statechart::Comment
- ObjectModelGe::Comment
- UseCaseGe::Comment
- Activity\_diagram::Comment
- Name, Specification, Description, Label (default is Description)
- Used for following subject::metaclass combinations:
- Statechart::Constraint
- ObjectModelGe::Constraint
- UseCaseGe::Constraint
- Activity\_diagram::Constraint
- Full\_path, Relative, Name\_only, Label (default is Name\_only)
- Used for following subject::metaclass combinations:
- ObjectModelGe::Package
- ObjectModelGe::PrimitiveOperation



- ObjectModelGe::Attribute
- ObjectModelGe::Type
- Name, Label (default is Name)
- Used for following subject::metaclass combinations:
- Activity\_diagram::Swimlane

## ShowStereotype

The ShowStereotype property determines if, and how, a stereotype of the element are displayed in a diagram. The possible values are:

- Label - The stereotype of the element are displayed as a text label.
- Bitmap - The bitmap image that is associated with the stereotype of the element are displayed.
- None - The stereotype of the element are not displayed.

The default value for this property varies for the different elements. For the following subject::metaclass combinations, the default value is None:

- Statechart::DefaultTransition
- Activity\_diagram::Action
- Activity\_diagram::ActionBlock
- Activity\_diagram::DefaultTransition
- Activity\_diagram::ReferenceActivity

For the following subject::metaclass combinations, the default value is Label:

- Statechart::Requirement
- Statechart::Depends
- ObjectModelGe::Depends
- ObjectModelGe::Inheritance
- ObjectModelGe::Actor
- UseCaseGe::Depends
- UseCaseGe::Inheritance
- Activity\_diagram::Depends
- Activity\_diagram::SubActivityState

## Requirement

The Requirement metaclass contains properties that control the appearance of requirements in activity diagrams.

### color

The color property specifies the default color of the border of a graphical item, such as an object box.

*(Default = 128,128,0)*

## Fillcolor

The Fillcolor property specifies the default fill color for the object.

*(Default = 0,255,255)*

## line\_width

The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines).

*(Default = 1)*

## name\_color

The name\_color property specifies the default color of names of graphical items.

*(Default = 0,0,0)*

## SpecificationAnnotationFormat

When a requirement is displayed on a diagram, you can choose to have the element show one of the following: its name, its label, its description, the specification text for the requirement.

If you choose to display the specification text, you can use the property SpecificationAnnotationFormat to customize the text that is displayed.

The following keywords can be included in the value of the property: \$Specification (to include the specification text) and \$ID (to include the ID that was specified in the Features dialog for the requirement).

Note that this property affects the displayed text only when a requirement uses the "Note" notation style. When using the "Box" notation style, only the specification text is displayed, regardless of the value of this property.

*Default = \$Specification*

## SelectorConnector

The SelectorConnector metaclass contains properties that control the appearance of selector connectors in

activity diagrams.

## color

The color property specifies the default color of the border of a graphical item, such as an object box.

*(Default = 0,128,255)*

## UseFillColor

The UseFillColor property is a Boolean value that specifies whether to use the fill color specified for that type of connector or port.

*(Default = True)*

## SendAction

The SendAction metaclass contains properties that relate to Send Action elements in activity diagrams.

## ShowNotation

The ShowNotation property determines what text is opened on Send Action elements in an activity diagram. The possible values are:

- Name - The name of the element is displayed.
- Label - The label of the element is displayed.
- Event - The name of the event selected is displayed.
- FullNotation - In addition to the name of the event selected, Rational Rhapsody displays the name of the target selected and the argument values you provided.

Note that this property can only be set at the diagram level or higher (not at the level of individual Send Action elements). When you change the value of this property, the display of any new Send Action elements are affected, but the display of Send Action elements already on the diagram remains as is.

*Default = FullNotation*

## ShowStereotype

The ShowStereotype property determines if, and how, an stereotypes of the element are displayed in a diagram. The possible values are:

- Label - The stereotypes of the element are displayed as a text label.
- Bitmap - The bitmap image that is associated with the stereotype of the element is displayed.

- None - The stereotypes of the element are not displayed.

Note that this property can only be set at the diagram level or higher (not at the level of individual Call Operation elements). When you change the value of this property, the display of any new Call Operation elements are affected, but the display of Call Operation elements already on the diagram remains as is.

*Default = None*

## Slider

The Slider metaclass contains properties that determine the appearance and behavior of slider controls on activity diagrams.

### Direction

The Direction property determines whether slider controls are used to input data, display data, or both. The possible values are:

- In - The sliders are only used to input data for the attribute to which it is bound.
- Out - The sliders are only used to display data for the attribute to which it is bound.
- InOut - The sliders are used to input data and display data.

*Default = InOut*

### ShowName

The ShowName property determines whether or not a caption is displayed for slider elements, and if so, what text should be displayed. The possible values are:

- BindedElementFullPath - The full path of the attribute that is bound to the slider.
- BindedElement - The name of the attribute that is bound to the slider.
- Name - The name of the slider element.
- None - No text is displayed.

*Default = Name*

## State

The State metaclass contains properties that control the appearance of states in activity diagrams.

### color

The color property specifies the default color of the border of a graphical item, such as an object box.

*(Default = 0,128,64)*

## **Fillcolor**

The Fillcolor property specifies the default fill color for the object.

*(Default = 0,255,255)*

## **line\_width**

The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines).

*(Default = 1)*

## **name\_color**

The name\_color property specifies the default color of names of graphical items.

*(Default = 0,0,0)*

# **StateDiagram**

The StateDiagram metaclass contains a property that controls the appearance of state diagrams.

## **Fillcolor**

The Fillcolor property specifies the default fill color for the object.

*(Default = 218,218,218)*

# **SubActivityState**

The SubActivityState metaclass properties control the appearance of subactivity states in activity diagrams.

## **line\_style**

The `line_style` property specifies the type of line used for a graphical item. The possible values are:

- `straight_arrows` - a straight line.
- `rectilinear_arrows` - rectilinear lines with right-angled corners placed at appropriate locations, depending on the start and end points of the line.
- `spline_arrows` - curved line without corners.

The default value for this property varies for the different elements. For the following `subject::metaclass` combinations, the default value is `straight_arrows`:

- `UseCaseGe::Inheritance`
- `Activity_diagram::DefaultTransition`
- `Statechart::Depends`

For the following `subject::metaclass` combinations, the default value is `spline_arrows`:

- `Statechart::DefaultTransition`
- `Activity_diagram::SubActivityState`

## ShowDescription

The `ShowDescription` property specifies whether descriptions for the `subactivitystate` in the activity diagram should be displayed.

*Default = Cleared*

## ShowName

The `ShowName` property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values for this property varies for the different elements, as does the default value used. The different values used are:

- `Description` - the content of the description field; relevant for elements such as comments
- `Full_path` - the full path describing the hierarchical position of an element, for example, `package_1::package_1b::class_0`
- `Label` - the label provided for the element
- `Name` - the name of the element
- `Name_only` - the name of the element only (as opposed to the full or relative path)
- `None` - nothing should be displayed
- `Relative` - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class are displayed with the class name only if the class is not positioned inside the package in the diagram.
- `Specification` - the content of the specification field; relevant for elements such as constraints

The various combinations of possible values are:

- Name\_only, Label (default is Name\_only)
- Used for following subject::metaclass combinations:
  - Statechart::State
- Name, Label, None (default is Name)
- Used for following subject::metaclass combinations:
  - Statechart::Transition
  - Statechart::DefaultTransition
  - ObjectModelGe::Aggregation
  - ObjectModelGe::Composition
  - ObjectModelGe::Association
  - ObjectModelGe::Link
  - UseCaseGe::Association
  - Activity\_diagram::Transition
  - Activity\_diagram::DefaultTransition
  - Activity\_diagram::ReferenceActivity
  - Activity\_diagram::SubActivityState
  - Activity\_diagram::ActivityParameter
- Name, Label, None (default is None)
- Used for following subject::metaclass combinations:
  - ObjectModelGe::Inheritance
  - ObjectModelGe::Depends
  - UseCaseGe::Depends
  - UseCaseGe::Inheritance
  - Activity\_diagram::ActivityParameter
  - Activity\_diagram::Depends
  - Statechart::Depends
- Full\_path, Relative, Name\_only, Label (default is Relative)
- Used for following subject::metaclass combinations:
  - Statechart::Requirement
  - ObjectModelGe::Class
  - ObjectModelGe::Object
  - ObjectModelGe::UseCase
  - ObjectModelGe::Actor
  - ObjectModelGe::Requirement
  - UseCaseGe::Actor
  - UseCaseGe::Association

- UseCaseGe::Requirement
- UseCaseGe::UseCase
- Activity\_diagram::Requirement
- Name, Description, Label (default is Description)
- Used for following subject::metaclass combinations:
- Statechart::Comment
- ObjectModelGe::Comment
- UseCaseGe::Comment
- Activity\_diagram::Comment
- Name, Specification, Description, Label (default is Description)
- Used for following subject::metaclass combinations:
- Statechart::Constraint
- ObjectModelGe::Constraint
- UseCaseGe::Constraint
- Activity\_diagram::Constraint
- Full\_path, Relative, Name\_only, Label (default is Name\_only)
- Used for following subject::metaclass combinations:
- ObjectModelGe::Package
- ObjectModelGe::PrimitiveOperation
- ObjectModelGe::Attribute
- ObjectModelGe::Type
- Name, Label (default is Name)
- Used for following subject::metaclass combinations:
- Activity\_diagram::Swimlane

## ShowStereotype

The ShowStereotype property determines if, and how, a stereotype of the element are displayed in a diagram. The possible values are:

- Label - The stereotype of the element are displayed as a text label.
- Bitmap - The bitmap image that is associated with the stereotype of the element are displayed.
- None - The stereotype of the element are not displayed.

The default value for this property varies for the different elements. For the following subject::metaclass combinations, the default value is None:

- Statechart::DefaultTransition
- Activity\_diagram::Action
- Activity\_diagram::ActionBlock
- Activity\_diagram::DefaultTransition



- Activity\_diagram::ReferenceActivity

For the following subject::metaclass combinations, the default value is Label:

- Statechart::Requirement
- Statechart::Depends
- ObjectModelGe::Depends
- ObjectModelGe::Inheritance
- ObjectModelGe::Actor
- UseCaseGe::Depends
- UseCaseGe::Inheritance
- Activity\_diagram::Depends
- Activity\_diagram::SubActivityState

## Swimlane

The Swimlane metaclass properties control the that control the appearance of swimlanes in activity diagrams.

### ShowName

The ShowName property determines what text is opened at the top of a swimlane. The possible values are:

- Represents - Only the class, package, or component represented by the swimlane is displayed with the "type" prefix. For example << rep. type>>rep.name If the Represents field was left blank, the swimlane name is displayed.
- RepresentsOnly - Only the class, package, or component represented by the swimlane is displayed. If the Represents field was left blank, the swimlane name is displayed.
- Name\_only - Only the name of the swimlane is displayed.
- Label - The label of the swimlane is displayed.

Note that this property can only be set at the diagram level or higher (not at the level of individual swimlanes).

When you change the value of this property, the display of any new swimlanes is affected, but the display of swimlanes already on the diagram remains as is.

*Default = Name\_only*

### ShowStereotype

The ShowStereotype property determines if, and how, the stereotypes of a swimlane are displayed in a diagram. The possible values are:

- Label - The stereotypes of the swimlane are displayed as a text label.
- Bitmap - The bitmap image that is associated with the stereotype of the swimlane is displayed.
- None - The stereotypes of the swimlane are not displayed.

Note that this property can only be set at the diagram level or higher (not at the level of individual swimlanes).

When you change the value of this property, the display of any new swimlanes is affected, but the display of swimlanes already on the diagram remains as is.

*Default = Label*

## TerminationConnector

The TerminationConnector metaclass contains properties that control the appearance of termination connectors in activity diagrams.

### color

The color property specifies the default color of the border of a graphical item, such as an object box.

*(Default = 0,128,0)*

### UseFillColor

The UseFillColor property is a Boolean value that specifies whether to use the fill color specified for that type of connector or port.

*(Default = True)*

## TextBox

The TextBox metaclass contains properties that determine the appearance and behavior of text box controls on activity diagrams.

### Direction

The Direction property determines whether text box controls are used to input data, display data, or both. The possible values are:

- In - The text boxes are only used to input data for the attribute to which it is bound.

- Out - The text boxes are only used to display data for the attribute to which it is bound.
- InOut - The text boxes are used to input data and display data.

*Default = InOut*

## ShowName

The ShowName property determines whether or not a caption is displayed for text box elements, and if so, what text should be displayed. The possible values are:

- BindedElementFullPath - The full path of the attribute that is bound to the text box.
- BindedElement - The name of the attribute that is bound to the text box.
- Name - The name of the text box element.
- None - No text is displayed.

*Default = Name*

## Transition

The Transition metaclass contains properties that control the appearance of transition lines in activity diagrams.

### ArrowHead

The property ArrowHead can be used to select the style of arrowhead you would like to use for transitions in diagrams such as activity diagrams and statecharts - an open arrowhead or closed arrowhead.

Note that when you change the value of the property, existing transitions will be displayed as they were previously until you refresh the diagram (F5).

*Default = OpenArrow*

### color

The color property specifies the default color of the border of a graphical item, such as an object box.

*(Default = 255,0,0)*

### label\_color

The label\_color property is currently unused.

LoopTransition

Statechart::Transition,0,127,255

(Default = 0,127,255)

## line\_style

The line\_style property specifies the default line style for a graphical item.

The possible values are:

- straight\_arrows - straight line.
- rectilinear\_arrows - rectilinear line with right-angled corners placed at appropriate locations, depending on the starting and ending points of the line
- spline\_arrows - curved line without corners
- rounded\_rectilinear\_arrows - similar to ordinary rectilinear lines but use rounded bends rather than straight ninety-degree angles

Default = spline\_arrows

Default = spline\_arrows

## ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values are:

- Name - the name of the element
- Label - the label provided for the element
- None - nothing should be displayed

Default = Name

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams.

The possible values are:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

Default = None

## **line\_width**

The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines).

*(Default = 1)*

# Ada\_CG

---

The Ada\_CG subject contains metaclasses that contain properties that control aspects of operating system environments.

The metaclasses are as follows:

- Argument
- Attribute
- Class
- Component
- Configuration
- Dependency
- File
- ModelElement
- Package
- Port
- Type
- Operation
- Relation
- Requirement
- Statechart
- Framework
- GNAT
- GNATVxWorks
- INTEGRITY
- INTEGRITY5
- MultiWin32
- Multi4Win32
- OBJECTADA
- RAVEN\_PPC
- SPARK
- Event
- Transition
- Generalization

## Argument

The Argument metaclass contains properties that control how arguments are generated in code.

The properties are as follows:

- `AccessTypeUsage`
- `AsAccess`
- `ClassWide`
- `DescriptionTemplate`
- `Constant`
- `NotNull`

## **AccessTypeUsage**

The `AccessTypeUsage` property defines the access type usage of Argument's type.

*(Default = None)*

## **AsAccess**

The `AsAccess` property sets the mode of a parameter to be access (as opposed to in, out, or in out). Note that access parameters are supported by Ada95, not Ada83.

*(Default = False)*

## **ClassWide**

The `ClassWide` property determines whether a class-wide modifier is generated for the argument.

*(Default = False)*

## **DescriptionTemplate**

The `DescriptionTemplate` property specifies how to generate the element description in the code. An empty `MultiLine` (the default value) tells Rational Rhapsody to use the default description generation rules. The property supports the following keywords:

`$Name` - The element name

`$FullName` - The full path of the element (`P1::P2::C.a`)

`$Description` - The element description

Element-specific keywords, as shown in the following table:

Metatype Describes Additional Supported Keywords

Argument Arguments \$Type - The argument type

\$Direction - The argument direction (in, out, and so on)

Attribute Attributes \$Type - The attribute type

Class Classes, actors, objects, and blocks

Event Events \$Arguments - The description for the event argument

Operation Primitive operations, triggered operations, \$Arguments - The description for the operation argument

constructors, and destructors \$Signature - The operation signature

Package Packages

Relation Association ends \$Target - The other end of the association

Type Types \$Type - Applicable to Typedef types

Tag - The value of the tag for the specified element

Property - The value of the element property with the specified name

The keywords are resolved in the following order:

Predefined keywords (such as \$Name)

Tag keywords

Property keywords

Note the following:

Keyword names can be written in parentheses. For example: \$(Name)

If the value of a keyword is a MultiLine, each new line (except the first one) starts with the value of the lang\_CG::Configuration::DescriptionBeginLine property; each line ends with the value of the ADA\_CG::Configuration::DescriptionEndLine property.

*(Default = Empty string)*

## Constant

The Constant property determines whether "constant" must be added on access typed arguments. This property must be used with Ada2005



example

arg : access constant integer

*(Default = False)*

## NotNull

The NotNull property determines whether "not null" must be added on access typed arguments. This property must be used with Ada2005

example

arg : not null access integer

*(Default = False)*

## Attribute

The Attribute metaclass contains properties that control attributes of code generation, such as whether to generate accessor operations.

The properties are as follows:

- Accessor
- AccessConstant
- AccessTypeUsage
- AccessorGenerate
- DeclarationPosition
- DeferredInitializationPosition
- DescriptionTemplate
- GenerateRenamesForSingleton
- ImplementationEpilog
- ImplementationProlog
- InlineAccessor
- InlineMutator
- IsAliased
- Mutator
- MutatorGenerate
- NotNull
- ParentDiscriminantValue

- RedefiningDiscriminantPolicy
- Renames
- SpecificationEpilog
- SpecificationProlog
- Visibility
- AccessorVisibility
- AttributeInitializationFile
- ConstantVariableAsDefine
- InitializationStyle
- Inline
- IsMutable
- Kind
- MarkPrologEpilogInAnnotations
- MutatorVisibility
- ReferenceImplementationPattern
- VariableInitializationFile

## Accessor

The Accessor property is ignored by Rational Rhapsody.

*(Default = Get\_\$attribute:c)*

## AccessConstant

This property is used to generated "constant" modifier on anonymous access type.

example

Attribute\_0 : access constant Integer;

*(Default = False)*

## AccessTypeUsage

This property defines the access type.

*(Default = None)*

## AccessorGenerate

The AccessorGenerate property specifies whether to generate accessor operations for attributes. The

possible values are as follows:

Checked - A `get()` method is generated for the attribute.

Cleared - A `get()` method is not generated for the attribute.

Setting this property to Cleared is one way to optimize your code for size.

*(Default = True)*

## DeclarationPosition

The DeclarationPosition property controls the declaration order of attributes. The possible values are as follows:

Default - Similar to the AfterClassRecord setting, with the following difference:

For static attributes defined in a class with the `ADA_CG::Attribute::Visibility` property set to Public, these attributes are generated after types whose `ADA_CG::Type::Visibility` property is set to Public.

You should not use this setting for new models. For more information, see the Sodus documentation for Ada.

BeforeClassRecord - Generate the attribute immediately before the class record.

AfterClassRecord - Generate the attribute immediately after the class record.

StartOfDeclaration - Generate the attribute immediately after the start of the section (private or public part of the specification, or package body).

EndOfDeclaration - Generate the attribute immediately before the end of the section (private or public part of the specification, or package body).

*(Default = Default)*

## DeferredInitializationPosition

Applicable to public constants only, this property controls where the deferred initialization is generated in the private part.

*(Default = None)*

## DescriptionTemplate

The DescriptionTemplate property specifies how to generate the element description in the code. An empty MultiLine (the default value) tells Rational Rhapsody to use the default description generation rules. The property supports the following keywords:

\$Name - The element name

\$FullName - The full path of the element (P1::P2::C.a)

\$Description - The element description

Element-specific keywords, as shown in the following table:

Metatype Describes Additional Supported Keywords

Argument Arguments \$Type - The argument type

\$Direction - The argument direction (in, out, and so on)

Attribute Attributes \$Type - The attribute type

Class Classes, actors, objects, and blocks

Event Events \$Arguments - The description for the event argument

Operation Primitive operations, triggered \$Arguments - The description for the operation argument

operations, constructors, and destructors \$Signature - The operation signature

Package Packages

Relation Association ends \$Target - The other end of the association

Type Types \$Type - Applicable to Typedef types

Tag - The value of the tag for the specified element

Property - The value of the element property with the specified name

The keywords are resolved in the following order:

Predefined keywords (such as \$Name)

Tag keywords

Property keywords

Note the following:

Keyword names can be written in parentheses. For example: \$(Name)

If the value of a keyword is a MultiLine, each new line (except the first one) starts with the value of the ADA\_CG::Configuration::DescriptionBeginLine property; each line ends with the value of the

ADA\_CG::Configuration::DescriptionEndLine property.

*(Default = )*

## **GenerateRenamesForSingleton**

This property controls the generation of renaming statements for attributes in singleton classes.

*(Default = True)*

## **ImplementationEpilog**

The ImplementationEpilog property adds code to the end of the definition of the model element.

*(Default = )*

## **ImplementationProlog**

The ImplementationProlog property adds code to the beginning of the definition of the model element

*(Default = )*

## **InlineAccessor**

This property controls generation of inline pragma for the accessor.

Checked

*(Default = True)*

## **InlineMutator**

This property controls generation of inline pragma for the mutator.

*(Default = True)*

## **IsAliased**

The IsAliased property is a Boolean value that specifies whether attributes are aliased.

*(Default = False)*

## **Mutator**

The Mutator property is ignored by Rational Rhapsody.

*(Default = Set\_\$attribute:c)*

## **MutatorGenerate**

The MutatorGenerate property specifies whether to generate mutators for attributes. The possible values are as follows:

Smart - Mutators are not generated for attributes that have the Constant modifier.

Always - Mutators are generated, regardless of the modifier.

Never - Mutators are not generated.

*(Default = Smart)*

## **NotNull**

This property is used to generated "not null" modifier before attribute's type definition.

example

attribute : not null access Integer;

*(Default = False)*

## **ParentDiscriminantValue**

This property holds the value to assign to the parent discriminant if it exists.

*(Default = Empty string)*

## **RedefiningDiscriminantPolicy**

This property controls the generation policy for this discriminant if the attribute is a <<Discriminant>> attribute and it is already defined as a <<Discriminant>> in one of the parent classes of the current class.

AsNew - attribute is generated as a regular discriminant

AsNewAndOverriding - attribute is generated as a regular discriminant, and the parent discriminant of the same name is assigned the value defined in the Ada\_CG::Attribute::ParentDiscriminantValue property.

AsOverriding - the parent discriminant of the same name is assigned the value defined in the Ada\_CG::Attribute::ParentDiscriminantValue property.

*(Default = AsNew)*

## Renames

This property enables an attribute to rename another attribute. Note that this feature can work only for static attributes in a class or for attributes in a package.

*(Default = Empty string)*

## SpecificationEpilog

The SpecificationEpilog property adds code to the end of the declaration of the model element.

*(Default = )*

## SpecificationProlog

The SpecificationProlog property adds code to the beginning of the declaration of the model element.

*(Default = )*

## Visibility

The Visibility property specifies the visibility of that kind of model element. Code generation maps the visibility specified for an element to the same visibility in the generated language.

*(Default = Public)*

## AccessorVisibility

The AccessorVisibility property specifies the access level of the generated accessor for attributes. This property defines the access level of an accessor operation regardless of the visibility of the attribute. The possible values are as follows:

fromAttribute - Use the access level for the accessor for the attribute.

public - Set the access level to public for the accessor.

private - Set the access level to private for the accessor.

protected - Set the access level to protected for the accessor.

default - Set the access level to default for the accessor.

*(Default = Empty string)*

## AttributeInitializationFile

The AttributeInitializationFile property specifies how static const attributes are initialized. In Rational Rhapsody, you can initialize these attributes in the specification file or directly in the initialization file. This property is analogous to the VariableInitializationFile property for global const variables. The possible values are as follows:

Default - The attribute is initialized in the specification file if the type declaration begins with const. Otherwise, the variable is initialized in the implementation file.

Implementation - Initialize constant attributes in the implementation file.

Specification - Initialize constant attributes in the specification file.

*(Default = Empty string)*

## ConstantVariableAsDefine

This property is a Boolean value that determines whether the variable, defined as constant in file or package, is generated by using a #define macro. Otherwise, it is generated by using the const qualifier.

*(Default = Empty string)*

## InitializationStyle

The InitializationStyle property specifies the initialization style used for attributes. When you specify an initial value for an attribute, Rational Rhapsody initializes the attribute based on the value of this property. In Rational Rhapsody Developer for C++, the possible values are as follows:

ByInitializer - Initialize the attribute in the initializer (a(y)). This is the default value.

If the initialization style is ByInitializer, the attribute initialization should be done after the user initializer, in the same order as the order of attributes in the code.

ByAssignment - Initialize the attribute in the constructor body (a = y).

In Rational Rhapsody Developer for Java, the possible values are as follows:

InClass - Initialize the attribute in the class declaration. This is the default value.

InConstructor - Initialize the attribute in each of the class constructors.

In Rational Rhapsody Developer for C, the attribute is initialized in the initializer body.

*(Default = Empty string)*

## Inline



The Inline property specifies how inline operations are generated. Which operations are affected by the Inline property depends on the metaclass:

Attribute - Applies only to operations that handle attributes (such as accessors and mutators)

Operation - Applies to all operations

Relation - Applies only to operations that handle relations

Inlining in Rational Rhapsody Developer for Ada

If the operation is marked as inline, the package specification includes an inline pragma after the declaration. For example:

```
pragma inline (operation name);
```

The two possible values for Rational Rhapsody Developer for Ada are as follows:

none

use\_pragma

*(Default = Empty string)*

## IsMutable

The IsMutable property is a Boolean value that allows you to specify that an attribute is a mutable attribute.

*(Default = Empty string)*

## Kind

The Kind property specifies the kind of operation that should be generated for an element. The kind of operations that can be generated is language-dependent (for example, virtual and abstract exist only in C++ and Java). In Java, Kind can be defined only for attributes and operations, but not for relations. This property affects class operations, in addition to accessors and mutators for relations and attributes. The possible values are as follows:

common - Class operations and accessor/mutator are non-virtual.

virtual - Class operations and accessor/mutator are virtual. This type is valid for C++ and Java only.

abstract - Class operations and accessor/mutator are pure virtual. This type is valid for C++ and Java only.

*(Default = Empty string)*

## MarkPrologEpilogInAnnotations

The `MarkPrologEpilogInAnnotations` property specifies whether to generate ignore annotations for the Specification, Implementation Prolog, and Epilog properties so they are ignored during roundtrip. When you insert code element declarations (variables, types, functions, and so on) in the Specification, Implementation Prolog, and Epilog properties, after a full roundtrip those elements are added to the model and are duplicated on the next code generation. Use the `MarkPrologEpilogInAnnotations` property to have Rational Rhapsody automatically ignore the information specified in the Specification, Implementation Prolog, and Epilog properties instead of adding the ignore annotations manually. The possible values for the `MarkPrologEpilogInAnnotations` property are:

**None** - Rational Rhapsody does not generate any annotations. Any models created before Version 4.1 automatically have this property setting.

**Ignore** - Rational Rhapsody generates the `///  
[ ignore annotation before the code specified in the Specification, Implementation Prolog, and Epilog properties, and generates the ///  
[ annotation after the code specified in those properties.`

**Auto** - If the code in the Specification, Implementation Prolog, and Epilog properties is one line (it does not contain any newline characters (`\n`)), no annotations are generated (the same behavior has the **None**

setting). If there is more than one line, Rational Rhapsody generates the `///  
[ ignore annotation before the code specified in the Specification, Implementation Prolog, and Epilog properties, and generates the ///  
[ annotation after the code specified in those properties (the same behavior as the Ignore setting).`

During roundtrip, any ignore annotations in the comments of the element are not included in its description. Because the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties are generated between the annotation and its declaration for the element, you cannot rename those elements on roundtrip. If you change the name of an element, it is removed from the model and added with the new name. Some model information (for example, property settings) might be lost.

*(Default = Empty string)*

## MutatorVisibility

The `MutatorVisibility` property specifies the access level of the generated mutator for attributes. This property defines the access level of a mutator operation regardless of the visibility of the attribute. The possible values are as follows:

**fromAttribute** - Use the access level for the attribute for the mutator.

**public** - Set the access level to public for the mutator.

**private** - Set the access level to private for the mutator.

**protected** - Set the access level to protected for the mutator.

**default** - Set the access level to default for the mutator.

*(Default = Empty string)*

## ReferenceImplementationPattern

The ReferenceImplementationPattern property specifies how the Reference option for attribute/typedefs (composite types) is mapped to code.

*(Default = Empty string)*

## VariableInitializationFile

The VariableInitializationFile property specifies how global constant variables are initialized. You can initialize these variables in the specification file. You can use these variables as compile-time constants that can be used to define array sizes, for example. Rational Rhapsody automatically identifies constant variables with const. By modifying

this property, you can choose the initialization file directly. The possible values are as follows:

Default - The variable is initialized in the specification file if the type declaration begins with const. Otherwise, the variable is initialized in the implementation file.

Implementation - Initialize global constant variables in the implementation file.

Specification - Initialize global constant variables in the specification file.

*(Default = Empty string)*

## Class

The Class metaclass contains properties that affect the generated classes.

The properties are as follows:

- AccessTypeName
- BaseNumberOfInstances
- ClassWideAccessTypeName
- DeclarationPosition
- ExceptionHandlerActive
- ExceptionHandlerReactive
- Final
- GenerateAccessType
- GenerateClassWideAccessType
- GenerateRecordType
- HasUnknownDiscriminant

- ImplementationEpilog
- ImplementationPragmas
- ImplementationPragmasInContextClause
- ImplementationProlog
- InitializationCode
- InterfaceModifier
- IsLimited
- IsNested
- IsPrivate
- IsStatic
- NestingVisibility
- OptimizeStatechartsWithoutEventsMemoryAllocation
- RecordTypeName
- RelativeEventDataRecordTypeComponentsNaming
- Renames
- SingletonExposeThis
- SingletonInstanceVisibility
- SpecificationEpilog
- SpecificationPragmas
- SpecificationPragmasInContextClause
- SpecificationProlog
- TaskBody
- UseAda83Framework
- Visibility
- ActiveMessageQueueSize
- ActiveStackSize
- ActiveThreadName
- ActiveThreadPriority
- AdditionalBaseClasses
- AdditionalNumberOfInstances
- Animate
- ComplexityForInlining
- DeclarationModifier
- DescriptionTemplate
- Destructor
- Embeddable
- EnableDynamicAllocation
- EnableUseFromCPP

- GenerateDestructor
- ImpIncludes
- InitCleanUpRelations
- InstanceDeclaration
- IsCompletedOperation
- IsInOperation
- IsReactiveInterface
- MarkPrologEpilogInAnnotations
- MaximumPendingEvents
- ObjectTypeAsSingleton
- PackageName
- PrivateInherits
- ReactiveThreadSettingPolicy
- ReturnType
- SpecIncludes
- TriggerArgument
- VirtualInherits

## AccessTypeName

The AccessTypeName property specifies the name of the access type generated for the class record. If this is not set, Rational Rhapsody uses <class\_name>\_acc\_t.

Some macros can be inserted in the property in order to be replaced by execution of a rule.

Possible macro are:

[[Name]]

\$name

\$(name)

example

[[name]]\_acc\_t

*(Default = Empty string)*

## BaseNumberOfInstances

The BaseNumberOfInstances property is a string that specifies the size of the local heap memory pool allocated for either:

Instances of the class (ADA\_CG::Class)

Instances of the event (ADA\_CG::Event)

This property provides support for static architectures found in hard real-time and safety-critical systems without memory management capabilities during run time. All instances of events are dynamically allocated during initialization. Once allocated, the event queue for the thread remains static in size.

Triggered operations use the properties defined for events. When the memory pool is exhausted, an additional amount, specified by the AdditionalNumberOfInstances property, is allocated. Memory pools for classes can be used only with the Flat statechart implementation scheme. The possible values are as follows:

An empty string (blank) - Memory is always dynamically allocated.

n (positive integer) - An array is allocated in this size for instances.

The related properties are as follows:

AdditionalNumberOfInstances - Specifies the number of instances to allocate if the pool runs out.

ProtectStaticMemoryPool - Specifies whether the pool should be protected (to support a multithreaded environment)

EmptyMemoryPoolCallback - Specifies a user callback function to be called when the pool is empty. This property should be used instead of the AdditionalNumberOfInstance property for error handling.

EmptyMemoryPoolMessage - When set to true, this property causes a message to be displayed if the pool runs out of memory in instrumented mode.

*(Default = Empty string)*

## ClassWideAccessTypeName

This property specifies the name of the class-wide access type generated for the class record. If this is not set, Rational Rhapsody uses <class\_name>\_wide\_acc\_t.

Some macros can be inserted in the property in order to be replaced by execution of a rule.

Possible macro are:

[[Name]]

\$name

\$(name)

example

[[name]]\_wide\_acc\_t

*(Default = Empty string)*

## DeclarationPosition

This property allows you to control the declaration order for a class.

Determines declaration position relative to the section it is declared in (public part of spec, private part of spec, body) and to the “virtual” location of the class record type if it is/was declared in this section.

This property applies only to nested classes.

The possible values are:

BeforeClassRecord - generate the class immediately before the class record

AfterClassRecord - generate the class immediately after the class record

StartOfDeclaration - generate the class immediately after the start of the section (private or public part of the specification, or package body)

EndOfDeclaration - generate the class immediately before the end of the section (private or public part of the specification, or package body)

*(Default = EndOfDeclaration)*

## ExceptionHandlerActive

This property enables implementation of an exception handler at active level. If this property is not blank, then CG will create a new function:

```
procedure Exception_Handler (  
  this : in out handled_active.handled_active_t;  
  E : in Ada.Exceptions.Exception_Occurrence  
);
```

Implementation of this function is defined in this property. It will be called after the active context unended loop, if an exception occurs.

CG will add "Ada.Exceptions" with clause.

Implementation example:

```
Ada.Text_IO.put_line(Ada.Exceptions.Exception_Message(E));
```

In this example, user will have to add an "Ada.Text\_IO" with clause.

*(Default = )*

## ExceptionHandlerReactive

This property enables implementation of an exception handler at reactive level.

If this property is not blank, then CG will create a new function

```
procedure Reactive_Exception_Handler(  
  this : in out class_1_t;  
  e : Ada.Exceptions.Exception_Occurrence  
);
```

Implementation of this function is defined in this property. It will be called in the Root\_State\_Process\_Event() function if an exception occurs.

CG will add "Ada.Exceptions" with clause.

Implementation example:

```
Ada.Text_IO.put_line(Ada.Exceptions.Exception_Message(e));
```

in this example, user will have to add an "Ada.Text\_IO" with clause.

*(Default = )*

## Final

The Final property, when set to Cleared, specifies that the generated record for the class is a tagged record. This property applies to ADA95.

*Default = Cleared*

*(Default = False)*



## **GenerateAccessType**

The GenerateAccessType property determines which access types are generated for the class. The possible values are as follows:

None - Access types are not generated.

Standard - An access type is generated.

General - General access types are generated.

*(Default = General)*

## **GenerateClassWideAccessType**

This property determines which Wide access types are generated for the class. The possible values are:

None - wide access types are not generated

Standard - a standard wide access type is generated

General - general wide access types are generated

Constant - constant wide access types are generated

*(Default = None)*

## **GenerateRecordType**

This property determines whether the class record is generated. If this property is set to False, the user must provide one manually.

*(Default = True)*

## **HasUnknownDiscriminant**

This property determines whether an unknown discriminant (encased in quotation marks) is generated for this class.

*(Default = False)*

## **ImplementationEpilog**

The ImplementationEpilog property adds code to the end of the definition of the model element.

*(Default = )*

## ImplementationPragmas

The ImplementationPragmas property specifies the user-defined pragmas to generate in the body.

*(Default = )*

## ImplementationPragmasInContextClause

The ImplementationPragmasInContextClause property specifies the user-defined pragmas to generate in the context clause of the body.

*(Default = )*

## ImplementationProlog

The ImplementationProlog property adds code to the beginning of the definition of the model element

*(Default = )*

## InitializationCode

The InitializationCode property adds the specified initialization code in the body of the class. A non-abstract class can have initialization code that is executed during elaboration of the associated package.

*(Default = )*

## InterfaceModifier

This property enables adding a modifier on interface definition in order to create a task interface or a protected interface or a synchronized interface.

Example

type interface\_1\_t is task interface;

*(Default = None)*

## IsLimited

The IsLimited property determines whether the class or record type is generated as limited.

*(Default = False)*

## **IsNested**

The IsNested property specifies whether to generate the class or package as nested.

*(Default = False)*

## **IsPrivate**

The IsPrivate property specifies whether to generate the class or package as private.

*(Default = False)*

## **IsStatic**

This property indicates whether the class is a regular class or a static class.

A static class has no record type and all its attributes and operations are static. The parameter "this" is never generated.

*(Default = False)*

## **NestingVisibility**

The NestingVisibility property specifies the visibility of the generated specification of the nested class or package.

*(Default = Public)*

## **OptimizeStatechartsWithoutEventsMemoryAllocation**

The OptimizeStatechartsWithoutEventsMemoryAllocation property determines whether the generated code uses dynamic memory allocation for statecharts that use only triggered operations.

*(Default = False)*

## **RecordTypeName**

The RecordTypeName property specifies the name of the class record type. If this is not set, Rational Rhapsody uses <class\_name>\_t.

Some macros can be inserted in the property in order to be replaced by execution of a rule.

Possible macro are:

[[Name]]

\$name

\$(name)

example

[[name]]\_t

*(Default = Empty string)*

## RelativeEventDataRecordTypeComponentsNaming

This property enables relative naming of event data record type components that represent events and triggered operation parameters. If this is Checked, no events or triggered operations share argument names because they would generate record components with the same name (which would not compile).

*(Default = False)*

## Renames

This property enables a class to rename another class. You can also rename a class using a renames dependency. In case of a conflict, the dependency has precedence.

Note that using this feature on classes limits what you can do with the renaming class. Specifically:

You cannot derive other classes from it

Adding attributes or operations to it has no effect on the generated code

*(Default = Empty string)*

## SingletonExposeThis

The SingletonExposeThis property, when set to Cleared, specifies that all non-static methods are considered as static methods and does not have this parameter passed in.

*(Default = False)*

## SingletonInstanceVisibility

This property controls where the singleton unique instance is generated. The possible values are:

Body

Private

*(Default = Body)*

## **SpecificationEpilog**

The SpecificationEpilog property adds code to the end of the declaration of the model element.

*(Default = )*

## **SpecificationPragmas**

The SpecificationPragmas property specifies the user-defined pragmas to generate in the specification.

*(Default = )*

## **SpecificationPragmasInContextClause**

The SpecificationPragmasInContextClause property specifies the user-defined pragmas to generate in the context clause of the specification.

*(Default = )*

## **SpecificationProlog**

The SpecificationProlog property adds code to the beginning of the declaration of the model element.

*(Default = )*

## **TaskBody**

The TaskBody property defines an alternate task body for ADA Task and ADA Task Type classes.

*(Default = )*

## **UseAda83Framework**

This property specifies what Framework version should be used. If set to True, the generated code for statecharts, events, and guarded operations and attributes will use Ada 83 constructs. If set to False, Ada 95 constructs will be used.

*(Default = False)*

## Visibility

The Visibility property specifies the visibility of the model element. Code generation maps the visibility specified for an element to the same visibility in the generated language. The possible values are as follows:

Public - The model element is public.

Protected - The model element is protected.

Private - The element is private.

*(Default = Public)*

## ActiveMessageQueueSize

The ActiveMessageQueueSize property specifies the size of the message queue allocated for active classes. The possible values are as follows:

A string - Specifies the message queue size for an active class.

An empty string (blank) - The value is set in an operating system-specific manner.

*(Default = Empty string)*

## ActiveStackSize

The ActiveStackSize property specifies the size of the stack allocated for active objects. The possible values are as follows:

Any integer - Specifies that a stack of that size is allocated for active objects.

An empty string (blank) - If not specified, the stack size is set in an operating system-specific manner, based on the value of the ActiveStackSize property for the framework.

*(Default = Empty string)*

## ActiveThreadName

The ActiveThreadName property specifies the name of the active thread. This facilitates debugging in complex environments in which many threads are constantly being created and deleted on-the-fly. This property is effective only in the pSoSystem (both PPC and X86) and VxWorks environments. In pSoSystem, the thread name is truncated

to three characters. The animation thread name is not taken from the active thread name. The possible

values are as follows:

A string - Names the active thread.

An empty string (blank) - The value is set in an operating system-specific manner, based on the value of the `ActiveThreadName` property for the framework.

*(Default = Empty string)*

## ActiveThreadPriority

The `ActiveThreadPriority` property specifies the priority of active class threads. The possible values are as follows:

A string - Specifies thread priority of an active class.

An empty string (blank) - The value is set in an operating system-specific manner.

*(Default = Empty string)*

## AdditionalBaseClasses

The `AdditionalBaseClasses` property adds inheritance from external classes to the model.

*(Default = Empty string)*

## AdditionalNumberOfInstances

The `AdditionalNumberOfInstances` property is a string that specifies the size of the local heap allocated for events when the current pool is full. Triggered operations use the event properties. This property provides support for static architectures found in hard real-time and safety-critical systems without memory management capabilities during run time. All events are dynamically allocated during initialization. Once allocated, the event queue for the thread remains static in size. The possible values are as follows:

An empty string (blank) - No additional memory is to be allocated when the initial memory pool is exhausted.

n (a positive integer) - Specifies the size of the array allocated for additional instances.

*(Default = Empty string)*

## Animate

The `Animate` property specifies whether animation code is generated for an element. You can specify your own animation function by using the `CG::Attribute::AnimSerializeOperation` property. The semantics of the `Animate` property is always in favor of the owner settings:

If a package `Animate` property is set to `Cleared`, all the classes owned by the package are not animated, regardless of the class `Animate` settings.

If a class `Animate` property is set to `Cleared`, all the elements in the class (attributes, operations, relations, and so on) are not animated.

If an operation `Animate` property is set to `Cleared`, all the arguments are not animated.

If the `AnimateArguments` property is set to `Cleared`, all the arguments are not animated, regardless of the specific argument `Animate` property settings.

*(Default = Empty string)*

## ComplexityForInlining

The `ComplexityForInlining` property specifies the upper bound for the number of lines in user code that are allowed to be inlined. User code is the action part of transitions in statecharts. For example, if you use the value 3, all transitions with actions consisting of three lines or fewer of code are automatically inlined in the calling function. Inlining is replacing a function call in the generated code with the actual code statements that make up the body of the function. This optimizes the code execution at the expense of an increase in code size. For example, increasing the number of lines that can be inlined from 3 to 5 has shortened the code execution time in some cases up to 10%. This property applies only to the Flat implementation scheme for statecharts.

*(Default = Empty string)*

## DeclarationModifier

The `DeclarationModifier` property adds a string to the class or event declaration. The string is displayed between the class keyword and the class name in the generated code. For example, for a class `A`, the `DeclarationModifier` would appear as follows:

`class DeclarationModifier> A { ... }`; This property adds a modifier to the class declaration. For example, if you have a class `myExportableClass` that is exported from a DLL by using the `MYDLL_API` macro, you can set the `DeclarationModifier` property to “`MYDLL_API`.” The generated code would then be as follows:

```
class MYDLL_API myExportableClass
```

```
{ ... };
```

 This property supports two keywords: `$component` and `$class`.

*(Default = Empty string)*

## DescriptionTemplate

The `DescriptionTemplate` property specifies how to generate the element description in the code. An empty `MultiLine` (the default value) tells Rational Rhapsody to use the default description generation rules. The property supports the following keywords:



\$Name - The element name

\$FullName - The full path of the element (P1::P2::C.a)

\$Description - The element description

Element-specific keywords, as shown in the following table:

Metatype Describes Additional Supported Keywords

Argument Arguments \$Type - The argument type

\$Direction - The argument direction (in, out, and so on)

Attribute Attributes \$Type - The attribute type

Class Classes, actors, objects, and blocks

Event Events \$Arguments - The description for the event argument

Operation Primitive operations, triggered operations, \$Arguments - The description for the operation argument

constructors, and destructors \$Signature - The operation signature

Package Packages

Relation Association ends \$Target - The other end of the association

Type Types \$Type - Applicable to Typedef types

Tag - The value of the specified the element tag

Property - The value of the element property with the specified name

The keywords are resolved in the following order:

Predefined keywords (such as \$Name)

Tag keywords

Property keywords

Note the following:

Keyword names can be written in parentheses. For example:

\$(Name)

If the value of a keyword is a MultiLine, each new line (except the first one) starts with the value of the ADA\_CG::Configuration::DescriptionBeginLine

property; each line ends with the value of the ADA\_CG::Configuration::DescriptionEndLine property.

*(Default = Empty string)*

## Destructor

The Destructor property controls the generation of virtual destructors in C++. The property exists for C for historical reasons, with a single value of auto, but it has no effect on the generated C code. The possible values are as follows:

auto - A virtual destructor is generated for an object only if it has at least one virtual function.

virtual - A virtual destructor is generated in all cases.

abstract - A virtual destructor is generated as a pure virtual function.

common - A nonvirtual destructor is generated.

*(Default = Empty string)*

## Embeddable

The Embeddable property is a Boolean property that specifies whether a class can be allocated by value (nested) inside another class

or package. For example, if the Embeddable property is True, 20 instances of a class A can be allocated inside another class by using the following syntax:

```
A itsA[20];
```

The possible values are as follows:

True - The object can be allocated by value inside a composite object or package. The object declaration and definition are generated in the specification file of the composite.

False - The object cannot be embedded inside another object. The object declaration and definition are generated in the implementation file of the composite.

The Embeddable property is used with the EmbeddedScalar and EmbeddedFixed properties to determine how to generate code for an embedded object. The Embeddable property must be set to True for either of those properties to take effect. It is also closely related to the ImplementWithStaticArray property, which also needs to be set in order to support by-value allocation. To generate C-like code in C++, set the Embeddable property to True. Relations can be generated by value only under the following circumstances:

The Embeddable property of the nested class is set to True.

The multiplicity of the relation is well-defined (not “\*”).

The `ImplementWithStaticArray` property of the component relation is set to `FixedAndBounded`.

When the `Embeddable` property is `False`:

The attributes of the object are encapsulated. Clients of the object are forced to use it only by way of its operations, because there is no direct access to its attributes.

Dynamic allocation must be used. The compiler does not know how to statically allocate an object when its declaration is not visible.

The nested object cannot be reactive. This is because of the reactive macros. There is a complex workaround for this issue.

*(Default = Empty string)*

## EnableDynamicAllocation

The `EnableDynamicAllocation` property specifies whether to use dynamic memory allocation for objects. The possible values are as follows:

`True` - Dynamic allocation of events is enabled. `Create()` and `Destroy()` operations are generated for the object or object type.

`False` - Events are dynamically allocated during initialization, but not during run time. `Create()` and `Destroy()` operations are not generated for the object. This setting is recommended for static architectures that do not use dynamic memory management during run time.

If you are managing your own memory pools, set this property to `False` and call `CPPReactive_gen()` directly. The following example shows how to call `RiCReactive_gen()` directly to send a static event to a reactive object A, when you use a member function of A `genStaticEv2A()`:

```
void A_genStaticEv2A(struct A_t* const me) {  
  
    {  
  
        /*#[ operation genStaticEv2A() */  
  
        static struct ev _ev;  
  
        ev_Init(_ev);  
  
        RiCEvent_setDeleteAfterConsume(((RiCEvent*)_ev),  
  
        RiCFALSE);  
  
        (void) RiCReactive_gen(me-ric_reactive,  
  
        ((RiCEvent*)_ev), RiCFALSE);  
    }  
}
```

```
/*#]*/
```

```
}
```

```
}
```

Alternatively, you can use internal memory pools by setting the `BaseNumberOfInstances` property, which results in the use of framework memory pools. If you use the framework memory pools, do not disable the `Create()` and `Destroy()` methods because these methods are used to manage the memory pool. When you disable the generation of the `Create()` and `Destroy()` methods, you can still inject events in animation by supplying an alternate function to get an event instance. To do this, set the `AnimInstanceCreate` property.

*(Default = Empty string)*

## EnableUseFromCPP

The `EnableUseFromCPP` property specifies whether to wrap C operations with an appropriate extern C {} wrapper to prevent problems when code

is compiled with a C++ compiler. Wrapping C code with extern C includes C code in a C++ application. Note that the structure definition for the object is not wrapped - only the functions are. For example, if the `EnableUseFromCPP` is set to `True` for an object, the following wrapper code is generated for its operations:

```
#ifdef __cplusplus
```

```
extern "C" {
```

```
#endif /* __cplusplus */
```

```
/* Operations */
```

```
#ifdef __cplusplus
```

```
}
```

```
#endif /* __cplusplus */
```

*(Default = Empty string)*

## GenerateDestructor

The `GenerateDestructor` property specifies whether to generate a destructor for a class.

*(Default = Empty string)*

## ImplIncludes

The `ImpIncludes` property specifies the names (including full paths) of header files to be included at the top of implementation files generated for classes, objects or object types, or packages. Separate multiple file names by using commas, without spaces.

*(Default = Empty string)*

## InitCleanUpRelations

The `InitCleanUpRelations` property specifies whether to generate `initRelations()` and `cleanUpRelations()` operations for sets of related global instances. This property applies only to composites and global relations.

*(Default = Empty string)*

## InstanceDeclaration

The `InstanceDeclaration` property specifies how instances are declared in code. The default value for C is as follows:

```
struct $cname$suffix
```

In the generated code, the variable `$cname` is replaced with the object (or object type) name. The variable `$suffix` is replaced with the type suffix “\_t,” if the object is of implicit type.

*(Default = Empty string)*

## IsCompletedOperation

The `IsCompletedOperation` specifies whether `state_IS_COMPLETED` operations are generated as functions or macros ( by using `#define`). The possible values are as follows:

Plain - `state_IS_COMPLETED` operations are generated as functions (pre-V4.2 behavior). This is the default value.

Inline - `state_IS_COMPLETED` operations are generated by using `#define` macros, if the body contains only a return statement.

*(Default = Empty string)*

## IsInOperation

The `IsInOperation` specifies how `state_IN` methods are generated.

*(Default = Empty string)*

## IsReactiveInterface

The `IsReactiveInterface` property modifies the way reactive classes are generated. It has the following effects:

Virtual inheritance from `OMReactive`

Prevents instrumentation

Prevents the thread argument and the initialization code (setting the active context) in the class constructor

Creates a pure-virtual destructor (by default)

This property affects only classes that declare themselves as interfaces by having a stereotype with a name that contains the word “interface” (case-insensitive). In previous versions of Rational Rhapsody, a class could inherit from a single reactive class only, regardless of whether it was an interface or implementation class. Beginning with Version 4.0.1 MR2, a class can inherit (implement) several reactive interfaces. In Rational Rhapsody Developer for C++, you must explicitly designate reactive interfaces because the code generator applies special translation rules involving multiple inheritance from the Rational Rhapsody framework. You can designate a reactive interface in two ways:

Set the `ADA_CG::Class::IsReactiveInterface` property to true.

Use the predefined stereotype `Reactive_interface`. This stereotype uses stereotype-based code generation in order to automatically apply the correct property value.

Alternatively, you can define another stereotype (such as `PortSpec`;) that sets `IsReactiveInterface` to true and use that stereotype. A class is considered reactive if it meets all the following conditions:

The `ADA_CG::Framework::ReactiveBase` property is not empty.

The `ADA_CG::Framework::ReactiveBaseUsage` property is set to true.

One or more of the following conditions are true:

The class has a statechart or activity diagram.

The class is a composite class.

The class has event receptions or triggered operations.

*(Default = Empty string)*

## MarkPrologEpilogInAnnotations

The `MarkPrologEpilogInAnnotations` property specifies whether to generate ignore annotations for the Specification, Implementation Prolog, and Epilog properties so they are ignored during roundtrip. When you insert code element declarations (variables, types, functions, and so on) in the Specification, Implementation Prolog, and Epilog properties, after a full roundtrip those elements are added to the model and are duplicated on the next code generation. When you use the `MarkPrologEpilogInAnnotations`

property, you can have Rational Rhapsody automatically ignore the information specified in the

Specification, Implementation Prolog, and Epilog properties instead of adding the ignore annotations manually. The possible values for the MarkPrologEpilogInAnnotations property are as follows:

None - Rational Rhapsody does not generate any annotations. Any models created before Version 4.1 automatically have this property setting.

Ignore - Rational Rhapsody generates the `///  
Specification, Implementation Prolog, and Epilog properties, and generates the ///  
code specified in those properties.`

Auto - If the code in the Specification, Implementation Prolog, and Epilog properties is one line (it does not contain any newline characters (`\n`)), no annotations are generated (the same behavior has the None

setting). If there is more than one line, Rational Rhapsody generates the `///  
code specified in the Specification, Implementation Prolog, and Epilog properties, and generates the ///  
annotation after the code specified in those properties (the same behavior as the Ignore setting).`

During roundtrip, any ignore annotations in the comments of the element are not included in its description. Because the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties are generated between the annotation and its declaration for the element, you cannot rename those elements on roundtrip. If you change the name of an element, it is removed from the model and added with the new name. Some model information (for example, property settings) might be lost.

*(Default = Empty string)*

## MaximumPendingEvents

The MaximumPendingEvents property specifies the maximum number of events that can be simultaneously pending in the event queue of the active class. The possible values are as follows:

-1 - Memory is dynamically allocated.

Positive integer - Specifies the maximum number of events.

*(Default = Empty string)*

## ObjectTypeAsSingleton

The ObjectTypeAsSingleton property generates singleton code for object-types and actors. This functionality saves a singleton-type (actor) in its own repository unit, and manage that unit by using a configuration management tool. Set this property for a single object-type or higher. An object-type is generated as a singleton when all of the following conditions are met:

The object-type has the «Singleton» stereotype.

There is one and only one object of the object-type and the object multiplicity is 1.

The ObjectTypeAsSingleton property is set to True.

Note that when you expose a singleton object (for example, by creating a singleton object-type), Rational Rhapsody also modifies the code that is generated for the singleton.

*(Default = Empty string)*

## PackageName

The PackageName property enables setting a package name different than the one which is automatically generated. If the string is empty, then automatic naming rules is used. If the string is not empty, then it will be used to set the package name.

*(Default = Empty string)*

## PrivateInherits

The PrivateInherits property, when set for a particular class, contains the names of the base classes from which the class privately inherits. For example, if a class B is a subclass of a class A and PrivateInherits is set to A for class B, the code that is generated for B contains a declaration showing inheritance from A, as follows:

```
class B: private A { ... };
```

This property is interpreted as a string list. To combine classes for multiple inheritance by concatenation, separate the class names by using commas. For example, if you want to have class C inherit privately from classes A and B, set the PrivateInherits property for class C to “A,B.”

*(Default = Empty string)*

## ReactiveThreadSettingPolicy

The ReactiveThreadSettingPolicy property specifies how threads are set for reactive classes. The possible values are as follows:

Default - During code generation, Rational Rhapsody adds a thread argument to the constructor.

MainThread - Rational Rhapsody does not add an argument; the thread is set to the main thread.

UserDefined - Rational Rhapsody does not add an argument; you must set the value for the thread yourself.

*(Default = Empty string)*

## ReturnType

The ReturnType property specifies how code is generated when the type is used as a return type.

*(Default = Empty string)*



## SpecIncludes

The SpecIncludes property specifies the names (including full paths) of header files to be included at the top of specification

files generated for classes (C++ and Java), objects or object types (C), and packages. Separate multiple file names by using commas, without spaces.

*(Default = Empty string)*

## TriggerArgument

The TriggerArgument property specifies how the type should be passed in when used as an argument for events\triggered operations. By default, classes that are used as an argument are passed by way of a pointer. There are 4 other properties that effect how types are passed into and returned by regular operations: "in", "out", "in/out" and "return."

In

InOut

Out

*(Default = Empty string)*

## VirtualInherits

The VirtualInherits property, when set for a particular class, contains the names of base classes from which the inherits from as virtual. For example, if class B is a subclass of class A and VirtualInherits is set to "A" for class B, the code that is generated for B contains a declaration showing inheritance from A, as follows:

```
class B: virtual public A { ... };
```

This property is interpreted as a string list. To combine classes for multiple inheritance by concatenation. separate the names with commas. For example, if you want to have class C inherit virtually from classes A and B, set the VirtualInherits property for class C to "A,B."

*(Default = Empty string)*

## Component

The Component metaclass contains properties that affect the Ada component.

The properties are as follows:

- `RespectCodeLayout`
- `AdaVersion`
- `UseBoochComponents`
- `UseAdaFramework`
- `ClassStateDeclaration`

## **RespectCodeLayout**

This property activates the code preserving feature. The possible values are:

None - code preserving not activated

Ordering - code preserving is activated. The order of elements in the code is preserved. Full roundtripping is not available.

*(Default = None)*

## **AdaVersion**

This property indicates what version of ADA is being used.

*(Default = Ada95)*

## **UseBoochComponents**

This property specifies what version of the Booch components should be used.

*(Default = Booch\_95)*

## **UseAdaFramework**

This property specifies what version of the Framework should be used.

*(Default = NewFWK95)*

## **ClassStateDeclaration**

The `ClassStateDeclaration` property supports C compilers that cannot handle enum declarations inside struct declaration. The possible values are as follows:

`InClassDeclaration` - Generate the reactive statechart enum declaration in the class declaration.

`BeforeClassDeclaration` - Generate the reactive class statechart enum declaration before the declaration of

the class.

*(Default = Empty string)*

## Configuration

The Configuration metaclass contains properties that affect the configuration.

The properties are as follows:

- ContainerSet
- DefaultActiveGeneration
- DescriptionBeginLine
- DescriptionEndLine
- Environment
- ExternalGenerationTimeout
- ExternalGeneratorFileMappingRules
- GenerateAnnotationsForNonSPARKConfigurations
- GeneratorExtraPropertyFiles
- GeneratorRulesSet
- GeneratorScenarioName
- ImplementationEpilog
- ImplementationProlog
- LocalVariablesDeclaration
- SpecificationEpilog
- SpecificationProlog
- ClassStateDeclaration
- CodeGeneratorTool
- DefaultImplementationDirectory
- DefaultSpecificationDirectory
- DependencyRuleScheme
- EmptyArgumentListName
- GenerateDirectoryPerModelComponent
- GenericEventHandling
- InitializeEmbeddableObjectsByValue
- MarkPrologEpilogInAnnotations
- SourceListFile

## ContainerSet

The ContainerSet property specifies the container set used to implement relations.

*(Default = None)*

## DefaultActiveGeneration

The DefaultActiveGeneration property specifies whether the default active class is created, as well as the classes for which it acts

as the active context. The possible values are as follows:

Disable - The default active singleton is not created.

ReactiveWithoutContext - The default active singleton is created if there are reactive classes that consume events and do not have an active context explicitly specified. The default active singleton can handle only these classes.

All - The default active singleton is generated if there is at least one event-consuming reactive class and the active singleton can handle all reactive classes that consume events - even those reactive

classes that specify another active class as their active context.

*(Default = ReactiveWithoutContext)*

## DescriptionBeginLine

This property specifies the prefix for the beginning of comment lines in the generated code. This functionality uses a documentation system (such as Doxygen), which looks for a certain prefix to produce the documentation.

This property affects only the code that is generated for descriptions of model elements; other auto-generated comments are not affected.

The following table lists the default value for each language.

Language Edition Default Value

C `"//"`

C++ `""`

When you set this property, you should check the value of the C\_CG::DiffDelimiter property - if the same prefix is used, Rational Rhapsody does not update the generated code when the description is modified. If both DescriptionBeginLine and DiffDelimiter use the same prefix, modify the values of the following

properties under C\_CG::File:

DiffDelimiter

ImplementationHeader

SpecificationHeader

*(Default = --)*

## DescriptionEndLine

This property specifies the prefix for the end of comment lines in the generated code. This functionality uses a documentation system (such as Doxygen), which looks for a certain prefix to produce the documentation.

This property affects only the code that is generated for descriptions of model elements; other auto-generated comments are not affected.

The following table lists the default value for each language.

Language Edition Default Value

C "/\*"

C++ "\*/"

*(Default = Empty string)*

## Environment

The Environment property determines the target environment for a configuration. Generated code is targeted for that environment. See the Release Notes for the environments supported by Rational Rhapsody "out-of-the-box." "Out-of-the-box" support means that Rational Rhapsody includes a set of preconfigured code generation properties for the environment and precompiled versions of the relevant OXF libraries. The precompiled OXF libraries have been fully tested. You can also add new environments, for example if you want to generate code for another RTOS. This involves retargeting the OS wrapper files in the Rational Rhapsody framework and creating a new set of code generation properties for the target environment.

*(Default = GNAT)*

## ExternalGenerationTimeout

The ExternalGenerationTimeout property specifies how long, in seconds, Rational Rhapsody waits for the each class in the configuration scope to complete so you can once again make changes to the model. This property applies to both the full-featured external generator and makefile generator. For example, if you

set this property to 2 and you have 10 classes, Rational Rhapsody sets a timeout of 20. If the external code generator does not complete generation in this timeframe, Rational Rhapsody displays a message in the output window saying that the generator is not responding, and you are allowed to make changes to the model. If you set this property to 0, Rational Rhapsody does not time out the generation session, and waits for the code generator to complete its task - even if it takes forever. Rational Rhapsody waits for a notification from the full-featured external code generator, or for the process termination of a makefile generator.

*(Default = 0)*

## ExternalGeneratorFileMappingRules

The ExternalGeneratorFileMappingRules property specifies whether the external code generator uses the same file mapping and naming scheme ( mapping rules ) as Rational Rhapsody. If the mapping rules are different , the external generator must implement handlers to the GetFileName, GetMainFileName, and GetMakefileName events that Rational Rhapsody runs to get a requested file name and path. The possible values are as follows:

AsRhapsody - The external generator uses the same mapping rules as Rational Rhapsody.

DefinedByGenerator - The external generator has its own mapping rules.

*(Default = DefinedByGenerator)*

## GenerateAnnotationsForNonSPARKConfigurations

The GenerateAnnotationsForNonSPARKConfigurations property specifies whether or not to generate annotations for this configuration.

*(Default = False)*

## GeneratorExtraPropertyFiles

The GeneratorExtraPropertyFiles property opens the default Text Editor allowing the user to edit the \$OMROOT\CodeGenerator\GenerationRules\LangC\RI\_CG.ini file.

*(Default = \$OMROOT\..\Sodius\RI\_CG\RI\_CG.ini;\$OMROOT\..\Sodius\RI\_CG\LineWrap.ini;\$OMROOT\..\Sodius\RI\_CG\...*

## GeneratorRulesSet

The GeneratorRulesSet property specifies your own rules set.

*(Default = \$OMROOT\..\Sodius\RI\_CG\compiled\_rules\RI\_CG.classpath)*

## GeneratorScenarioName

The `GeneratorScenarioName` property specifies the scenario name for the rule, if you write your own set of code generation rules.

*(Default = `scenarios.Rhapsody_Generation.main`)*

## ImplementationEpilog

The `ImplementationEpilog` property adds code to the end of the definition of the model element.

*(Default = )*

## ImplementationProlog

The `ImplementationProlog` property adds code to the beginning of the definition of the model element

*(Default = )*

## LocalVariablesDeclaration

The `LocalVariablesDeclaration` property specifies variables that you want to appear in the declaration of the entrypoint or operation.

*(Default = )*

## SpecificationEpilog

The `SpecificationEpilog` property adds code to the end of the declaration of the model element.

*(Default = )*

## SpecificationProlog

The `SpecificationProlog` property adds code to the beginning of the declaration of the model element.

*(Default = )*

## ClassStateDeclaration

The `ClassStateDeclaration` property supports C compilers that cannot handle enum declarations inside struct declaration. The possible values are as follows:

`InClassDeclaration` - Generate the reactive statechart enum declaration in the class declaration.

`BeforeClassDeclaration` - Generate the reactive class statechart enum declaration before the declaration of

the class.

*(Default = Empty string)*

## CodeGeneratorTool

The CodeGeneratorTool property specifies which code generation tool to use for the given configuration. The possible values are as follows:

External - Use the registered, external code generator.

Internal - Use the Rational Rhapsody internal code generator.

*(Default = Empty string)*

## DefaultImplementationDirectory

The DefaultImplementationDirectory property specifies the relative path to the default directory for generated implementation files. The value of this property is added after the configuration path. Consider the following case:

File C.cpp is an implementation of class C mapped to a folder Foo.

The active configuration (cfg) is under component cmp.

DefaultImplementationDirectory is set to “src”

Rational Rhapsody generates C.cpp to root>\cmp\cfg\src\Foo. Note the following limitations:

This feature is not supported in COM- or CORBA-related components (C++ only).

The predefined OSE environments (OsePPCDiab and OseSfk) are not supported due to makefile flexibility issues.

This feature is not supported by the INTEGRITY adapter build file generator.

*(Default = Empty string)*

## DefaultSpecificationDirectory

The DefaultSpecificationDirectory property specifies the relative path to the default directory for generated specification files. The value of this property is added after the configuration path. Consider the following case:

File B.h is a specification of class B that is not mapped to any file.

The active configuration (cfg) is under component cmp.



DefaultSpecificationDirectory is set to “inc”

Rational Rhapsody generates B.h to root>\cmp\cfg\inc. Note the following limitations:

This feature is not supported in COM- or CORBA-related components (C++ only).

The predefined OSE environments (OsePPCDiab and OseSfk) are not supported due to makefile flexibility issues.

This feature is not supported by the INTEGRITY adapter build file generator.

*(Default = Empty string)*

## DependencyRuleScheme

The DependencyRuleScheme property specifies how dependency rules should be generated in the makefile. The possible values are as follows:

Basic - Generates only the local implementation and specification files in the dependency rule in the makefile.

ByScope - In addition to generating the same files as the Basic option, generates the specification files of related elements (dependencies, associations, generalizations, and so on) that are in the scope of the active component.

Extended - In addition to generating the same files as the ByScope option, generates the specification files of related external elements (specified by using the CG::Class/Package::UseAsExternal properties)

and elements that are not in the scope of the active component.

*(Default = Empty string)*

## EmptyArgumentListName

The EmptyArgumentListName specifies the string generated for the argument list when an operation has no arguments. For example, if you set this value to “void”, for an operation foo that has no arguments, Rational Rhapsody generates the following code:

```
int foo (void){ ...}
```

*(Default = Empty string)*

## GenerateDirectoryPerModelComponent

The GenerateDirectoryPerModelComponent property specifies whether to generate a separate directory for each package in the component. The possible values are as follows:

True - Rational Rhapsody creates a separate directory for each package in the component.

False - A separate directory is not created for each package.

*(Default = Empty string)*

## GenericEventHandling

The GenericEventHandling property is a Boolean value that determines whether to generate generic event-handling code. This property supports large-scale collaboration, where you might not be aware of which classes consume a base event of your part in the event hierarchy, and might not have access to parts of the model that use base events. Beginning with Rational Rhapsody 4.0, the framework base event class includes a new, virtual method that checks the event ID against the specified ID, thereby supplying a generic mechanism for events

without super events. The language-specific methods are as follows: C

```
#define RiCEvent_isTypeOf(event, id) ((event)-  
  
==  
  
(id))  
  
C++  
  
virtual OMBoolean isTypeOf(short id) const {return  
  
!!id  
  
==id;}
```

Each generated event that has a super event overrides the method to check the ID against its own ID, then calls its base event directly to continue the check. An event without a base event

returns False if the ID does not equal its own. When you set the GenericEventHandling property to False, event consumption code is generated as in version 3.0.1. Setting this property affects only the way events are consumed - the override on the isTypeOf() method is still generated, to allow handling of events in components that use the generic event handling. To support complete generic event handling, you should regenerate the code for all events and reactive classes. The default value for C is False; the default value for C++ and Java is True.

*(Default = Empty string)*

## InitializeEmbeddableObjectsByValue

The InitializeEmbeddableObjectsByValue property specifies whether embeddable classes and object types selected in the configuration

initial instances list should be allocated by value in the main() routine.

*(Default = Empty string)*

## MarkPrologEpilogInAnnotations

The MarkPrologEpilogInAnnotations property specifies whether to generate ignore annotations for the Specification, Implementation Prolog, and Epilog properties so they are ignored during roundtrip. When you insert code element declarations (variables, types, functions, and so on) in the Specification, Implementation Prolog, and Epilog properties, after a full roundtrip those elements are added to the model and are duplicated on the next code generation. When you use the MarkPrologEpilogInAnnotations

property, you can have Rational Rhapsody automatically ignore the information specified in the Specification, Implementation Prolog, and Epilog properties instead of adding the ignore annotations manually. The possible values for the MarkPrologEpilogInAnnotations property are as follows:

None - Rational Rhapsody does not generate any annotations. Any models created before Version 4.1 automatically have this property setting.

Ignore - Rational Rhapsody generates the `///  
[ ignore annotation before the code specified in the Specification, Implementation Prolog, and Epilog properties, and generates the ///  
] annotation after the code specified`

in those properties.

Auto - If the code in the Specification, Implementation Prolog, and Epilog properties is one line (it does not contain any newline characters (`\n`)), no annotations are generated (the same behavior has the None

setting). If there is more than one line, Rational Rhapsody generates the `///  
[ ignore annotation before the code specified in the Specification, Implementation Prolog, and Epilog properties, and generates the ///  
] annotation after the code specified in those properties (the same behavior as the Ignore setting).`

During roundtrip, any ignore annotations in the comments of the element are not included in its description. Because the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties are generated between the annotation and its declaration for the element, you cannot rename those elements on roundtrip. If you change the name of an element, it is removed from the model and added with the new name. Some model information (for example, property settings) might be lost.

*(Default = Empty string)*

## SourceListFile

The SourceListFile property specifies the name of the file containing a list of .java source files to be compiled with javac. The batch file used by the Build command (jdkmake.bat) can use the following call, rather than including a long list of source files:

`javac -g @files.lst` This same command is generated from the following line in the MakeFileContent property for Java:

`javac -g @$SourceListFile` If the SourceListFile property is empty, \$SourceListFile is replaced with a string containing all source file names, separated by spaces (for example, "A.java B.java"). This means

that if the MakeFileContent default value is not changed, you receive:

`javac -g @A.java B.java ...` If you do not want to use the file containing the list of sources, you must also change the MakeFileContent property to replace “`javac -g @$SourceListFile`” with “`javac -g $SourceListFile`”.

*(Default = Empty string)*

## Dependency

The Dependency metaclass controls the dependency for a package that defines a namespace.

The properties are as follows:

- AccessTypeUsage
- CreateUseStatement
- GeneratePragmaElaborate
- GeneratePragmaElaborateAll
- GenerateRenamesPackage
- GenerateWithClause
- ImplementationEpilog
- ImplementationProlog
- SpecificationEpilog
- SpecificationProlog
- UsesStatementPosition
- GenerateOriginComment
- IncludeStyle
- MarkPrologEpilogInAnnotations
- UseNameSpace
- IsLimited
- IsPrivate

### AccessTypeUsage

This property defines the access type. It can be used when the property `Ada_CG.Dependency.CreateUseStatement` is set to "UseType"

*(Default = None)*

### CreateUseStatement

This property determines whether a use statement is added to the code after the with statement. The supplier of the dependency must be a class or type.

*(Default = None)*

## **GeneratePragmaElaborate**

This property determines whether to generate an elaborate pragma for the supplier class in the client class or package.

*(Default = False)*

## **GeneratePragmaElaborateAll**

This property determines whether to generate a pre-elaborate pragma for the supplier class in the client class or package.

*(Default = False)*

## **GenerateRenamesPackage**

The GenerateRenamesPackage property enables to generate a Renames package with the usage dependency. The name of the renames package will be the name of the dependency.

*(Default = False)*

## **GenerateWithClause**

The GenerateWithClause property determines whether with clauses are generated for "Usage" dependencies. For example, you can generate a with clause for a package, P1, in the specification of another package, P2, by using a dependency, D1, and generate a use clause for P1 in the body of P2.

In addition, this functionality is useful for modeling inherited annotations across classes and packages.

*(Default = True)*

## **ImplementationEpilog**

The ImplementationEpilog property adds code to the end of the definition of the model element.

*(Default = )*

## **ImplementationProlog**

The ImplementationProlog property adds code to the beginning of the definition of the model element

*(Default = )*

## SpecificationEpilog

The SpecificationEpilog property adds code to the end of the declaration of the model element.

*(Default = )*

## SpecificationProlog

The SpecificationProlog property adds code to the beginning of the declaration of the model element.

*(Default = )*

## UsesStatementPosition

This property specifies whether the “Use” or “Use type” clause should be generated in the package context clause (before the “package” keyword) or in the package declaration or body (after the ”package” keyword).

*(Default = BeforePackage)*

## GenerateOriginComment

When set to Checked, generates a comment before #include statements that indicate which element "caused" the #include.

*(Default = Empty string)*

## IncludeStyle

The IncludeStyle property controls the style of #include statements. When you use this property, you can control the style of a specific dependency, or the entire configuration/component/project. To set the style for include files that are synthesized based on associations between model elements (for example, setting the type of some attribute

to a class), add a «Usage» dependency between the elements and set this property to the appropriate value. The possible values are as follows:

Default - Use angle brackets for include statements for external elements, and quotes for include statements for other elements.

Quotes - Enclose include files in quotation marks. For example:

```
#include "A.h"
```

When a compiler encounters an include file in quotes, it searches for the file in both the current directory and the directories specified in the include path. Note that the specific algorithm used is compiler-dependent.

AngledBrackets - Enclose include files in angle brackets. For example:

```
#include A.h
```

When a compiler encounters an include file in angle brackets, it searches for the file only in the directories specified in the include path.

If you set the property to AngledBrackets at the configuration level, you must also change the CG::File::IncludeScheme property to RelativeToConfiguration to ensure successful compilation.

*(Default = Empty string)*

## MarkPrologEpilogInAnnotations

The MarkPrologEpilogInAnnotations property specifies whether to generate ignore annotations for the Specification, Implementation Prolog, and Epilog properties so they are ignored during roundtrip. When you insert code element declarations (variables, types, functions, and so on) in the Specification, Implementation Prolog, and Epilog properties, after a full roundtrip those elements are added to the model and are duplicated on the next code generation. When you use the MarkPrologEpilogInAnnotations

property, you can have Rational Rhapsody automatically ignore the information specified in the Specification, Implementation Prolog, and Epilog properties instead of adding the ignore annotations manually. The possible values for the MarkPrologEpilogInAnnotations property are as follows:

None - Rational Rhapsody does not generate any annotations. Any models created before Version 4.1 automatically have this property setting.

Ignore - Rational Rhapsody generates the `/// ignore annotation before the code specified in the Specification, Implementation Prolog, and Epilog properties, and generates the /// annotation after the code specified`

in those properties.

Auto - If the code in the Specification, Implementation Prolog, and Epilog properties is one line (it does not contain any newline characters (`\n`)), no annotations are generated (the same behavior has the None setting). If there is more than one line, Rational Rhapsody generates the `/// ignore annotation before the code specified in the Specification, Implementation Prolog, and Epilog properties, and generates the /// annotation after the code specified in those properties (the same behavior as the Ignore setting).`

During roundtrip, any ignore annotations in the comments of the element are not included in its description. Because the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties are generated between the annotation and its declaration for the element, you cannot rename those elements on roundtrip. If you change the name of an element, it is removed from the model and added with the new name. Some model information (for example, property settings) might be lost.

*(Default = Empty string)*

## UseNameSpace

The UseNameSpace property models namespace usage. When you set a dependency to a package that defines a namespace and set this property to True, Rational Rhapsody generates a “using namespace” statement to the package namespace.

*(Default = Empty string)*

## IsLimited

The IsLimited property is used to generate limited with clause. It must be used in Ada2005

example

limited with class\_1;

*(Default = False)*

## IsPrivate

The IsPrivate property is used to generate private with clause. It must be used in Ada2005

example

private with class\_1;

*(Default = False)*

## File

The File metaclass contains properties that control the generated code files.

The properties are as follows:

- DiffDelimiter
- ImplementationFooter
- ImplementationHeader
- SpecificationFooter
- SpecificationHeader
- Footer
- Header
- ImplementationEpilog



- ImplementationProlog
- MarkPrologEpilogInAnnotations
- SpecificationEpilog
- SpecificationProlog

## DiffDelimiter

The DiffDelimiter property defines a symbol that is used to avoid overwriting an unchanged line of code during code generation. Use this property to avoid touching the source code file when the “diff-delimited” line has not changed. In general, fewer source files need to be recompiled if fewer source files are touched.

For example, the DiffDelimiter symbol “//!” is used in the ADA\_CG::File::Header property. This symbol is at the beginning of a line of code that includes the current code generation date. The code generator compares the code it would normally generate for that line (the current code generation date) to that previously generated (the last code generation date). If the date has not changed, the line is not overwritten, possibly preventing the modification time for the file from changing (being “touched”).

*(Default = --!)*

## ImplementationFooter

The ImplementationFooter property specifies the multiline footer to be generated at the end of implementation files.

To avoid redundant compilation, Rational Rhapsody avoids unnecessary changes to specific lines prefixed with a special string, defined by the ADA\_CG::File::DiffDelimiter property. The default DiffDelimiter value is “//!”. The keywords are resolved in the following order:

Predefined keywords

Property keywords

Tag keywords

Note the following:

Keyword names can be written in parentheses. For example:

\$(Name)

If the value of a keyword is a MultiLine, each new line (except the first one) starts with the value of the ADA\_CG::Configuration::DescriptionBeginLine property; each line ends with the value of the ADA\_CG::Configuration::DescriptionEndLine property.

*(Default = )*

## ImplementationHeader

The ImplementationHeader property specifies the multiline header that is generated at the beginning of implementation files.

To avoid redundant compilation, Rational Rhapsody avoids unnecessary changes to specific lines prefixed with a special string, defined by the ADA\_CG::File::DiffDelimiter property. The default DiffDelimiter value is "///!". The keywords are resolved in the following order:

Predefined keywords

Property keywords

Tag keywords

Note the following:

Keyword names can be written in parentheses. For example:

\$(Name)

If the value of a keyword is a MultiLine, each new line (except the first one) starts with the value of the ADA\_CG::Configuration::DescriptionBeginLine property; each line ends with the value of the ADA\_CG::Configuration::DescriptionEndLine property.

*(Default = )*

## SpecificationFooter

The SpecificationFooter property specifies the multiline footer to be generated at the end of specification files.

Footer format strings can contain any of the following keywords:

\$ProjectName - The project name.

\$ComponentName - The component name.

\$ConfigurationName - The configuration name.

\$ModelElementName - The name of the element mapped to the file. If there is more than one, this is the name of the first element.

\$FullModelElementName - The name of the element mapped to the file, including the full path. If there is more than one, this is the name of the first element.

\$CodeGeneratedDate - The generation date.

\$CodeGeneratedTime - The generation time.

\$RhapsodyVersion - The version of Rational Rhapsody that generated the file.

\$Login - The user who generated the file.

\$CodeGeneratedFileName - The name of the generated file.

\$FullCodeGeneratedFileName - The full file name.

\$Tag - The value of the specified the element tag.

\$Property - The value of the element property with the specified name.

To avoid redundant compilation, Rational Rhapsody avoids unnecessary changes to specific lines prefixed with a special string, defined by the ADA\_CG::File::DiffDelimiter property. The default DiffDelimiter value is "///". The keywords are resolved in the following order:

Predefined keywords (such as \$Name)

Property keywords

Tag keywords

Note the following:

Keyword names can be written in parentheses. For example:

\$(Name)

If the value of a keyword is a MultiLine, each new line (except the first one) starts with the value of the ADA\_CG::Configuration::DescriptionBeginLine property; each line ends with the value of the ADA\_CG::Configuration::DescriptionEndLine property.

*(Default = )*

## SpecificationHeader

The SpecificationHeader property specifies the multiline header to be generated at the beginning of specification files.

Header format strings can contain any of the following keywords:

\$ProjectName - The project name.

\$ComponentName - The component name.

\$ConfigurationName - The configuration name.

\$ModelElementName - The name of the element mapped to the file. If there is more than one, this is the name of the first element.

\$FullModelElementName - The name of the element mapped to the file, including the full path. If there is more than one, this is the name of the first element.

\$CodeGeneratedDate - The generation date.

\$CodeGeneratedTime - The generation time.

\$RhapsodyVersion - The version of Rational Rhapsody that generated the file.

\$Login - The user who generated the file.

\$CodeGeneratedFileName - The name of the generated file.

\$FullCodeGeneratedFileName - The full file name.

\$Tag - The value of the specified the element tag.

\$Property - The value of the element property with the specified name.

To avoid redundant compilation, Rational Rhapsody avoids unnecessary changes to specific lines prefixed with a special string, defined by the ADA\_CG::File::DiffDelimiter property. The default DiffDelimiter value is "//!". The keywords are resolved in the following order:

Predefined keywords (such as \$Name)

Property keywords

Tag keywords

Note the following:

Keyword names can be written in parentheses. For example:

\$(Name)

If the value of a keyword is a MultiLine, each new line (except the first one) starts with the value of the ADA\_CG::Configuration::DescriptionBeginLine property; each line ends with the value of the ADA\_CG::Configuration::DescriptionEndLine property.

(Default = )

## Footer

The Footer property specifies a multiline footer that is added to the end of generated Java files. The default footer template is as follows:

```
"/*****
```

File Path: \$FullCodeGeneratedFileName

```
*****/"
```

Footer format strings can contain any of the following keywords:

\$ProjectName - The project name.

\$ComponentName - The component name.

\$ConfigurationName - The configuration name.

\$ModelElementName - The name of the element mapped to the file. If there is more than one, this is the name of the first element.

\$FullModelElementName - The name of the element mapped to the file, including the full path. If there is more than one, this is the name of the first element.

\$CodeGeneratedDate - The generation date.

\$CodeGeneratedTime - The generation time.

\$RhapsodyVersion - The version of Rational Rhapsody that generated the file.

\$Login - The user who generated the file.

\$CodeGeneratedFileName - The name of the generated file.

\$FullCodeGeneratedFileName - The full file name.

\$Tag - The value of the specified the element tag.

\$Property - The value of the element property with the specified name.

To avoid redundant compilation, Rational Rhapsody avoids unnecessary changes to specific lines prefixed with a special string, defined by the ADA\_CG::File::DiffDelimiter property. The default DiffDelimiter

value is “//!”.

*(Default = Empty string)*

## Header

The Header property specifies a multiline header that is added to the top of all generated Java files. The default header template is as follows:

```
/*****
```

```
Rhapsody : $RhapsodyVersion
```

```
Login : $Login
```

```
Component : $ComponentName
```

Configuration : \$ConfigurationName

Model Element : \$FullModelElementName

//! Generated Date : \$CodeGeneratedDate

File Path : \$FullCodeGeneratedFileName

\*\*\*\*\*/

Header format strings can contain any of the following keywords:

\$ProjectName - The project name.

\$ComponentName - The component name.

\$ConfigurationName - The configuration name.

\$ModelElementName - The name of the element mapped to the file. If there is more than one, this is the name of the first element.

\$FullModelElementName - The name of the element mapped to the file, including the full path. If there is more than one, this is the name of the first element.

\$CodeGeneratedDate - The generation date.

\$CodeGeneratedTime - The generation time.

\$RhapsodyVersion - The version of Rational Rhapsody that generated the file.

\$Login - The user who generated the file.

\$CodeGeneratedFileName - The name of the generated file.

\$FullCodeGeneratedFileName - The full file name.

\$Tag - The value of the specified the element tag.

\$Property - The value of the element property with the specified name.

To avoid redundant compilation, Rational Rhapsody avoids unnecessary changes to specific lines prefixed with a special string, defined by the ADA\_CG::File::DiffDelimiter property. The default DiffDelimiter

value is “//!”.

*(Default = Empty string)*

## ImplementationEpilog

The ImplementationEpilog property adds code to the end of the definition of the model element.

*(Default = Empty string)*

## ImplementationProlog

The ImplementationProlog property adds code to the beginning of the definition of the model element

*(Default = Empty string)*

## MarkPrologEpilogInAnnotations

The MarkPrologEpilogInAnnotations property specifies whether to generate ignore annotations for the Specification, Implementation Prolog, and Epilog properties so they are ignored during roundtrip. When you insert code element declarations (variables, types, functions, and so on) in the Specification, Implementation Prolog, and Epilog properties, after a full roundtrip those elements are added to the model and are duplicated on the next code generation. When you use the MarkPrologEpilogInAnnotations property, you can have Rational Rhapsody automatically ignore the information specified in the Specification, Implementation Prolog, and Epilog properties instead of adding the ignore annotations manually. The possible values for the MarkPrologEpilogInAnnotations property are as follows:

None - Rational Rhapsody does not generate any annotations. Any models created before Version 4.1 automatically have this property setting.

Ignore - Rational Rhapsody generates the `///  
[ ignore annotation before the code specified in the Specification, Implementation Prolog, and Epilog properties, and generates the ///  
] annotation after the code specified`

in those properties.

Auto - If the code in the Specification, Implementation Prolog, and Epilog properties is one line (it does not contain any newline characters (`\n`)), no annotations are generated (the same behavior has the None setting). If there is more than one line, Rational Rhapsody generates the `///  
[ ignore annotation before the code specified in the Specification, Implementation Prolog, and Epilog properties, and generates the ///  
] annotation after the code specified in those properties (the same behavior as the Ignore setting).`

During roundtrip, any ignore annotations in the comments of the element are not included in its description. Because the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties are generated between the annotation and its declaration for the element, you cannot rename those elements on roundtrip. If you change the name of an element, it is removed from the model and added with the new name. Some model information (for example, property settings) might be lost.

*(Default = Empty string)*

## SpecificationEpilog

The SpecificationEpilog property adds code to the end of the declaration of the model element.

*(Default = Empty string)*

## SpecificationProlog

The SpecificationProlog property adds code to the beginning of the declaration of the model element.

*(Default = Empty string)*

## ModelElement

Contains properties related to multiple types of model elements.

## RefactorRenameRegularExpression

*Default = ([^A-Za-z0-9\_]/^)(*\$keyword*)([^A-Za-z0-9\_]/\$)*

## Package

The Package metaclass contains properties that affect packages.

The properties are as follows:

- `ContributesToNamespace`
- `DeclarationPosition`
- `EventsBaseID`
- `ImplementationEpilog`
- `ImplementationPragmas`
- `ImplementationPragmasInContextClause`
- `ImplementationProlog`
- `InitializationCode`
- `InstanceAlphabeticalSort`
- `InstanceImplementationProlog`
- `IsNested`
- `IsPrivate`
- `NestingVisibility`
- `PackageEventIdRange`
- `Renames`



- SpecificationEpilog
- SpecificationPragmas
- SpecificationPragmasInContextClause
- SpecificationProlog
- UseAda83Framework
- Animate
- DefineNameSpace
- DescriptionTemplate
- ImpIncludes
- MarkPrologEpilogInAnnotations
- PackageClassNamePolicy
- SpecIncludes

## ContributesToNamespace

The ContributesToNamespace property specifies whether the packages contained in this package is declared as children packages of this package. Regardless of the setting, a directory is created for the current package to hold its contained elements.

To generate the class name without a package name, set to False. This removes the package name prefix from the class name.

*(Default = True)*

## DeclarationPosition

The DeclarationPosition property controls the declaration order of attributes. The possible values are as follows:

Default - Similar to the AfterClassRecord setting, with the following difference:

For static attributes defined in a class with the ADA\_CG::Attribute::Visibility property set to Public, these attributes are generated after types whose ADA\_CG::Type::Visibility property is set to Public.

You should not use this setting for new models.

BeforeClassRecord - Generate the attribute immediately before the class record.

AfterClassRecord - Generate the attribute immediately after the class record.

StartOfDeclaration - Generate the attribute immediately after the start of the section (private or public part of the specification, or package body).

EndOfDeclaration - Generate the attribute immediately before the end of the section (private or public part of the specification, or package body).

*(Default = EndOfDeclaration)*

## **EventsBaseID**

The EventsBaseID property specifies the base ID for events.

*(Default = 1)*

## **ImplementationEpilog**

The ImplementationEpilog property adds code to the end of the definition of the model element.

*(Default = )*

## **ImplementationPragmas**

The ImplementationPragmas property specifies the user-defined pragmas to generate in the body.

*(Default = )*

## **ImplementationPragmasInContextClause**

The ImplementationPragmasInContextClause property specifies the user-defined pragmas to generate in the context clause of the body.

*(Default = )*

## **ImplementationProlog**

The ImplementationProlog property adds code to the beginning of the definition of the model element

*(Default = )*

## **InitializationCode**

The InitializationCode property specifies the user-defined initialization code to add to the package body.

*(Default = )*

## **InstanceAlphabeticalSort**

The InstanceAlphabeticalSort property enables sorting instance declaration and initialization, in file <Package\_Name>.RiA\_Instances.adb, when the package contains some objects. If this property is false,

the user can control the order of initialization in Browser interface.

*(Default = True)*

## **InstanceImplementationProlog**

The InstanceImplementationProlog property specifies some text to be generated on start of the file which is automatically generated for object which needs to be instantiated. This file is usually named <Package\_Name>.RiA\_Instances.ad\*.

*(Default = )*

## **IsNested**

The IsNested property specifies whether to generate the class or package as nested.

*(Default = False)*

## **IsPrivate**

The IsPrivate property specifies whether to generate the class or package as private.

*(Default = False)*

## **NestingVisibility**

The NestingVisibility property specifies the visibility of the generated specification of the nested class or package.

*(Default = Public)*

## **PackageEventIdRange**

The PackageEventIdRange property specifies the maximum number of events allowed in a package. This property is set on the component level.

*(Default = 200)*

## **Renames**

The Renames property enables one element to rename another element of the same type. You can also rename an element by using a "renames" dependency. In the case of a conflict, the dependency has precedence. Note the following:

For attributes, this property works only for static attributes in a class or for attributes in a package.

For operations, this property contains the name of the operation this operation is renaming. The signatures of the two operations must match.

*(Default = Empty string)*

## SpecificationEpilog

The SpecificationEpilog property adds code to the end of the declaration of the model element.

*(Default = )*

## SpecificationPragmas

The SpecificationPragmas property specifies the user-defined pragmas to generate in the specification.

*(Default = )*

## SpecificationPragmasInContextClause

The SpecificationPragmasInContextClause property specifies the user-defined pragmas to generate in the context clause of the specification.

*(Default = )*

## SpecificationProlog

The SpecificationProlog property adds code to the beginning of the declaration of the model element.

*(Default = )*

## UseAda83Framework

This property specifies what version of the Framework should be used. If set to True, then the generated code for statecharts, events, and guarded operations and attributes will use Ada 83 constructs. If set to False, Ada 95 constructs will be used.

*(Default = False)*

## Animate

The Animate property specifies whether animation code is generated for an element. You can specify your own animation function by using the CG::Attribute::AnimSerializeOperation property. The semantics of the Animate property is always in favor of the owner settings:

If a package `Animate` property is set to `False`, all the classes owned by the package are not animated, regardless of the class `Animate` settings.

If a class `Animate` property is set to `False`, all the elements in the class (attributes, operations, relations, and so on) are not animated.

If an operation `Animate` property is set to `False`, all the arguments are not animated.

If the `AnimateArguments` property is set to `False`, all the arguments are not animated, regardless of the specific argument `Animate` property settings.

*(Default = Empty string)*

## DefineNameSpace

The `DefineNameSpace` property specifies whether a package defines a namespace. A namespace is a declarative region that attaches an additional identifier to any names declared inside it.

*(Default = Empty string)*

## DescriptionTemplate

The `DescriptionTemplate` property specifies how to generate the element description in the code. An empty `MultiLine` (the default value) tells Rational Rhapsody to use the default description generation rules. The property supports the following keywords:

`$Name` - The element name

`$FullName` - The full path of the element (`P1::P2::C.a`)

`$Description` - The element description

Element-specific keywords, as shown in the following table:

Metatype Describes Additional Supported Keywords

Argument Arguments `$Type` - The argument type

`$Direction` - The argument direction (in, out, and so on)

Attribute Attributes `$Type` - The attribute type

Class Classes, actors, objects, and blocks

Event Events `$Arguments` - The description for the event argument

Operation Primitive operations, triggered operations, `$Arguments` - The description for the operation argument

constructors, and destructors \$Signature - The operation signature

Package Packages

Relation Association ends \$Target - The other end of the association

Type Types \$Type - Applicable to Typedef types

Tag - The value of the specified the element tag

Property - The value of the element property with the specified name

The keywords are resolved in the following order:

Predefined keywords (such as \$Name)

Tag keywords

Property keywords

Note the following:

Keyword names can be written in parentheses. For example:

\$(Name)

If the value of a keyword is a MultiLine, each new line (except the first one) starts with the value of the ADA\_CG::Configuration::DescriptionBeginLine property; each line ends with the value of the ADA\_CG::Configuration::DescriptionEndLine property.

*(Default = Empty string)*

## ImplIncludes

The ImplIncludes property specifies the names (including full paths) of header files to be included at the top of implementation files generated for classes, objects or object types, or packages. Separate multiple file names by using commas, without spaces.

*(Default = Empty string)*

## MarkPrologEpilogInAnnotations

The MarkPrologEpilogInAnnotations property specifies whether to generate ignore annotations for the Specification, Implementation Prolog, and Epilog properties so they are ignored during roundtrip. When you insert code element declarations (variables, types, functions, and so on) in the Specification, Implementation Prolog, and Epilog properties, after a full roundtrip those elements are added to the model and are duplicated on the next code generation. When you use the MarkPrologEpilogInAnnotations property, you can have Rational Rhapsody automatically ignore the information specified in the

Specification, Implementation Prolog, and Epilog properties instead of adding the ignore annotations manually. The possible values for the MarkPrologEpilogInAnnotations property are as follows:

None - Rational Rhapsody does not generate any annotations. Any models created before Version 4.1 automatically have this property setting.

Ignore - Rational Rhapsody generates the `///  
Specification, Implementation Prolog, and Epilog properties, and generates the ///  
code specified in those properties.`

Auto - If the code in the Specification, Implementation Prolog, and Epilog properties is one line (it does not contain any newline characters (`\n`)), no annotations are generated (the same behavior has the None setting). If there is more than one line, Rational Rhapsody generates the `///  
code specified in the Specification, Implementation Prolog, and Epilog properties, and generates the ///  
annotation after the code specified in those properties (the same behavior as the Ignore setting).`

During roundtrip, any ignore annotations in the comments of the element are not included in its description. Because the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties are generated between the annotation and its declaration for the element, you cannot rename those elements on roundtrip. If you change the name of an element, it is removed from the model and added with the new name. Some model information (for example, property settings) might be lost.

*(Default = Empty string)*

## PackageClassNamePolicy

The PackageClassNamePolicy property specifies the naming policy for classes generated by Rational Rhapsody. The software generates a class for each

package in the Rational Rhapsody Developer for Java model. The possible values are as follows:

Default - Use the default naming style (the package class name is the same as the package name).

WithSuffix - Add a suffix to the class name. The suffix is “\_pkgClass.”

*(Default = Empty string)*

## SpecIncludes

The SpecIncludes property specifies the names (including full paths) of header files to be included at the top of specification files generated for classes (C++ and Java), objects or object types (C), and packages. Separate multiple file names by using commas, without spaces.

*(Default = Empty string)*

# Port

The Port metaclass controls whether code is generated for ports.

The properties are as follows:

- Generate
- AddNullCheckToPortWiringCode
- DescriptionTemplate
- SupportMulticast

## Generate

The Generate property specifies whether to generate code for a particular type of element.

*(Default = True)*

## AddNullCheckToPortWiringCode

This property allows generation of a null pointer check in port wiring code.

*(Default = True)*

## DescriptionTemplate

The DescriptionTemplate property specifies how to generate the element description in the code. An empty MultiLine (the default value) tells Rational Rhapsody to use the default description generation rules. The property supports the following keywords:

- \* \$Name - The element name
- \* \$FullName - The full path of the element (P1::P2::C.a)
- \* \$Description - The element description
- \* Tag - The value of the tag for the specified element
- \* Property - The value of the element property with the specified name

The keywords are resolved in the following order:

- \* Predefined keywords (such as \$Name)
- \* Tag keywords



\* Property keywords

Note the following:

\* Keyword names can be written in parentheses. For example: \$(Name)

\* If the value of a keyword is a MultiLine, each new line (except the first one) starts with the value of the ADA\_CG::Configuration::DescriptionBeginLine property; each line ends with the value of the ADA\_CG::Configuration::DescriptionEndLine property.

*(Default = )*

## SupportMulticast

This property allows generation of multicasting on ports.

Never - no multicasting available

Smart - instruments code only if multicasting is necessary

Always - always instruments code for multicasting

*(Default = Empty string)*

## Type

The Type metaclass contains a property that affects the visibility of data types.

The properties are as follows:

- AccessKind
- AccessTypeUsage
- AnimEnumerationTypeImage
- AnimSerializeOperation
- AnimSerializeOperationImpl
- AnimUnserializeOperation
- AnimUnserializeOperationImpl
- DeclarationPosition
- DescriptionTemplate
- Final
- IsLimited
- LanguageMap

- Visibility
- EnumerationAsTypedef
- In
- InOut
- IsNotNull
- Out
- PrivateName
- PublicName
- ReferenceImplementationPattern
- ReturnType
- StructAsTypedef
- TriggerArgument
- UnionAsTypedef

## AccessKind

For typedef types, this property indicates if the access type is basic, general or constant. Effective only if the check box "Reference" is checked.

Example

```
type Type_0 is access Boolean; -- Ada_CG.Type.AccessKind = Standard
```

```
type Type_1 is access all Boolean; -- Ada_CG.Type.AccessKind = General
```

```
type Type_2 is access constant Boolean; -- Ada_CG.Type.AccessKind = Constant
```

*(Default = Standard)*

## AccessTypeUsage

For typedef types, this property indicates if the basic type is referred to as an access type, a class-wide access type or a regular type or not. Effective only if the basic type is a class.

*(Default = None)*

## AnimEnumerationTypeImage

This property is a Boolean value that determines whether the Image attribute is used for enumerated types when you are using animation.

*(Default = False)*

## AnimSerializeOperation

The AnimSerializeOperation property specifies the name of an external function used to animate all attributes and arguments that are of that type. Rational Rhapsody can animate (display) the values of simple types and one-dimensional arrays without any problem.

To display the current values of such attributes during an animation session, run the Features window for the instance. However, if you want to animate a more complex type, such as a date, the type must be converted to a string (char \*) for Rational Rhapsody to display it. This is generally done by writing a global function, an instrumentation function, that takes one argument of the type you want to display, and returns a char \*.

You must disable animation of the instrumentation function itself (by using the Animate and AnimateArguments properties for the function). For example, you can have a type tDate, defined as follows:

```
typedef struct date {  
  
    int day;  
  
    int month;  
  
    int year; } %s;
```

You can have an object with an attribute count of type int, and an attribute date of type tDate. The object can have an initializer with the following body:

```
me-date.month = 5;  
  
me-date.day = 12;  
  
me-date.year = 2000;
```

If you want to animate the date attribute, the AnimSerializeOperation property for date must be set to the name of a function that is to convert the type tDate to char \*. For example, you can set the property to a function named showDate. This function name must be entered without any parentheses. It must take an attribute of type tDate and return a char \*. The Animate and AnimateArguments properties for the showDate function must be set to False. The implementation of the showDate function might be as follows:

```
showDate(tDate aDate) {  
  
    char* buff;  
  
    buff = (char*) malloc(sizeof(char) * 20);  
  
    sprintf(buff,"%d %d %d",  
  
    aDate.month,aDate.day,aDate.year);
```

```

return buff;

}

```

When you run this model with animation, instances of this object displays a value of 5 12 2000 for the date attribute in the browser. If the showDate function is defined in the same class that the attribute belongs to and the function is not static, the AnimSerializeOperation property value should be similar to the following:

```
myReal-showDate
```

This value shows that the function is called from the serializeAttributes function, located in the class OMAAnimatedclassname. The showDate function must allocate memory for the returned string through the malloc/alloc/calloc function in C, or the new operator in C++. Otherwise, the system shuts down unexpectedly.

*(Default = Empty string)*

## AnimSerializeOperationImpl

The AnimSerializeOperationImpl property specifies the implementation of an auto generated function used to animate all attributes and arguments that are of that type. Rational Rhapsody can animate (display) the values of simple types and one-dimensional arrays without any problem. If you are defining your own types, serialize operation implementation must be defined by you. If this property is not empty, code generator will generate automatically this function with the specified implementation. The signature is autogenerated. The udtValue argument will have the type of animated type. The operation name is controled with the property Ada\_CG::Type::AnimSerializeOperation. For example, your Serialization operation might look similar to the following:

```

procedure Add_Attribute (

udtName : in String;

udtValue : in My_Natural;

udtAttrList : in System.address

) is

begin

RhpAnim.Add_Attribute( udtName,

My_Natural'Image(udtValue),

udtAttrList

);

end Add_Attribute;

```

for this example the property Ada\_CG::Type::AnimSerializeOperationImpl should have the value

begin

RhpAnim.Add\_Attribute( udtName,

%s'Image(udtValue),

udtAttrList

);

(Default = )

## AnimUnserializeOperation

The AnimUnserializeOperation property converts a string to the value of an element (the opposite of the AnimSerializeOperation property). Unserialize functions are used for event generation or operation invocation by using the Animation toolbar to convert the string (received from the user) to the value of the event or operation before the event generation or operation invocation. For example, your serialization operation might look similar to the following:

```
char* myX2String(const Rec f)
```

```
{
```

```
char* cS = new char[OutputStringLength];
```

```
/* conversion from the Rec type to string */
```

```
return (cS);
```

```
}
```

The unserialization operation would be:

```
Rec myString2X (char* C, Rec T)
```

```
{
```

```
T = new Trc;
```

```
/* conversion of the string C to the Rec type */
```

```
delete C;
```

```
return (T);
```

```
}
```

(*Default = Empty string*)

## AnimUnserializeOperationImpl

The AnimUnserializeOperationImpl property provides implementation of unserialize operation to converts a string to the value of an element. Unserialize functions are used for event generation or operation invocation by using the Animation toolbar to convert the string (received from the user) to the value of the event or operation before the event generation or operation invocation. The signature of the operation will be auto generated. The data argument will have the type of animated type. The operation name is controlled with the property Ada\_CG::Type::AnimUnserializeOperation. For example, your Unserialization operation might look similar to the following:

```
procedure get_Attribute (  
  
data : in out My_Natural;  
  
address : in System.address;  
  
position : in System.address  
  
) is  
  
value : Natural;  
  
begin  
  
rhpanim.get_attribute(value,address,position);  
  
data := My_Natural(value);  
  
end Get_Attribute;
```

for this example the property Ada\_CG::Type::AnimUnserializeOperationImpl should have the value

```
value : Natural;  
  
begin  
  
rhpanim.get_attribute(value,address,position);  
  
data := %s(value);  
  
(Default = )
```

## DeclarationPosition

The DeclarationPosition property specifies where the type declaration is displayed. The possible values

are as follows:

**BeforeClassRecord** - The type declaration is displayed before the class record (CR) declaration if CR has a visibility set to public, and before the class record forward declaration if CR has a visibility set to private.

**AfterClassRecord** - The type declaration is displayed after the class record declaration if CR has a visibility set to public, and after the class record forward declaration if CR has a visibility set to private.

**StartOfDeclaration** - The type declaration is displayed among the first declarations (together with other types having the same settings) in the public section if CR has a visibility set to public, and among the first declarations in the private section (together with other types having the same settings) if CR has a visibility set to private.

**EndOfDeclaration** - The type declaration is displayed among the last declarations (together with other types having the same settings) in the public section if CR has a visibility set to public, and among the last declarations in the private section (together with other types having the same settings) if CR has a visibility set to private.

If the `ADA.CG::Type::Visibility` property is set to "Body," no matter the settings of `ADA.CG::Type::DeclarationPosition` property, the type declaration still displays in the package body.

*(Default = BeforeClassRecord)*

## DescriptionTemplate

The `DescriptionTemplate` property specifies how to generate the element description in the code. An empty `MultiLine` (the default value) tells Rational Rhapsody to use the default description generation rules. The property supports the following keywords:

`$Name` - The element name

`$FullName` - The full path of the element (P1::P2::C.a)

`$Description` - The element description

Element-specific keywords, as shown in the following table:

Metatype Describes Additional Supported Keywords

Argument Arguments `$Type` - The argument type

`$Direction` - The argument direction (in, out, and so on)

Attribute Attributes `$Type` - The attribute type

Class Classes, actors, objects, and blocks

Event Events `$Arguments` - The description for the event argument

Operation Primitive operations, triggered operations, \$Arguments - The description for the operation argument

constructors, and destructors \$Signature - The operation signature

Package Packages

Relation Association ends \$Target - The other end of the association

Type Types \$Type - Applicable to Typedef types

Tag - The value of the specified the element tag

Property - The value of the element property with the specified name

The keywords are resolved in the following order:

Predefined keywords (such as \$Name)

Tag keywords

Property keywords

Note the following:

Keyword names can be written in parentheses. For example:

\$(Name)

If the value of a keyword is a MultiLine, each new line (except the first one) starts with the value of the ADA\_CG::Configuration::DescriptionBeginLine property; each line ends with the value of the ADA\_CG::Configuration::DescriptionEndLine property.

*(Default = )*

## Final

The Final property, when set to Cleared, specifies that the generated record for the class is a tagged record. This property applies to ADA95.

*(Default = True)*

## IsLimited

The IsLimited property determines whether the class or record type is generated as limited.

*(Default = False)*



## LanguageMap

The LanguageMap property specifies the Ada declaration for Rational Rhapsody language-independent types. This property is not used anymore.

*(Default = Empty string)*

## Visibility

The Visibility property specifies the visibility of the model element. Code generation maps the visibility specified for an element to the same visibility in the generated language. The possible values are as follows:

Public - The model element is public.

Protected - The model element is protected.

Private - The element is private.

*(Default = Public)*

## EnumerationAsTypedef

The EnumerationAsTypedef property specifies whether the generated enum should be wrapped by a typedef. This property is applicable to enumeration types in C and C++.

*(Default = Empty string)*

## In

The In property specifies how code is generated when the type is used with an argument that has the modifier "In."

*(Default = Empty string)*

## InOut

The InOut property specifies how code is generated when the type is used with an argument that has the modifier "InOut."

*(Default = Empty string)*

## IsNotNull

This property is used to generate "not null" modifier for access type definition.

Example

type type\_1 is not null access Integer;

*(Default = False)*

## Out

The Out property specifies how code is generated when the type is used with an argument that has the modifier "Out."

*(Default = Empty string)*

## PrivateName

The PrivateName property specifies the pattern used to generate names of private operations in C.

*(Default = Empty string)*

## PublicName

The PublicName property specifies the pattern used to generate names of public operations in C.

*(Default = Empty string)*

## ReferenceImplementationPattern

The ReferenceImplementationPattern property specifies how the Reference option for attribute/typedefs (composite types) is mapped to code.

*(Default = Empty string)*

## ReturnType

The ReturnType property specifies how code is generated when the type is used as a return type.

*(Default = Empty string)*

## StructAsTypedef

The StructAsTypedef property specifies whether the generated struct should be wrapped by a typedef. This property is applicable to structure types in C and C++.

*(Default = Empty string)*

## TriggerArgument

The TriggerArgument property is used for mapping event and triggered operation arguments to code instead of the In, InOut, and Out properties. A different property is required because of code generation limitations related to event arguments. See also:

In

InOut

Out

*(Default = Empty string)*

## UnionAsTypedef

The UnionAsTypedef property specifies whether the generated union should be wrapped by a typedef. This property is applicable to union types in C and C++.

*(Default = Empty string)*

## Operation

The Operation metaclass contains properties that control operations.

The properties are as follows:

- AlphabeticalSort
- AnimAllowInvocation
- DeclarationPosition
- DescriptionTemplate
- EntryCondition
- ExpressionFunctionForwardDeclPosition
- ExpressionFunctionForwardDeclVisibility
- GenerateImplementation
- GenerateForwardDeclarationInPackageBody
- ImplementationEpilog
- ImplementationProlog
- ImplementationName
- Inline

- IsAnimationHelper
- IsEntry
- Kind
- LocalVariablesDeclaration
- PreserveUserCode
- Renames
- RenamesKind
- ReturnTypeByAccess
- SpecificationEpilog
- SpecificationProlog
- TaskDefaultScheme
- TaskDefaultSchemeDelayStatement
- ThisByAccess
- ThisAccessTypeUsage
- ThisPassingMode
- ThisName
- VirtualMethodGenerationScheme
- ActivityReferenceToAttributes
- ImplementActivityDiagram
- IsExplicit
- IsNative
- MarkPrologEpilogInAnnotations
- Me
- MeDeclType
- PrivateQualifier
- ProtectedName
- PublicName
- PublicQualifier
- ThrowExceptions
- ReturnAsAccess
- ReturnNotNullAccess
- ReturnAccessConstant
- ThisNotNullAccess
- ThisAccessConstant
- IsExpressionFunction

## AlphabeticalSort

This property controls alphabetical sorting of operations on generation.

*(Default = True)*

## **AnimAllowInvocation**

The AnimAllowInvocation property specifies whether primitive and triggered operations can be called during instrumentation. If an operation is called during animation, its return value is displayed in the output window; if it is traced, the return value is displayed in the console. The possible values are as follows:

All - Enable all operation calls, regardless of visibility.

None - Do not enable operation calls.

Public - Enable calls to public operations only.

Protected - Enable calls to protected operations only.

*(Default = None)*

## **DeclarationPosition**

This property specifies where the operation declaration appears. The possible values are:

BeforeClassRecord - the operation declaration appears before the class record (CR) declaration if CR has a visibility set to public, and before the class record forward declaration if CR has a visibility set to private

AfterClassRecord - the operation declaration appears after the class record declaration if CR has a visibility set to public, and after the class record forward declaration if CR has a visibility set to private

StartOfDeclaration - the operation declaration appears among the first declarations (together with other operations having the same settings) in the public section

EndOfDeclaration - the operation declaration appears among the last declarations (together with other operations having the same settings) in the public section

*(Default = AfterClassRecord)*

## **DescriptionTemplate**

The DescriptionTemplate property specifies how to generate the element description in the code. An empty MultiLine (the default value) tells Rational Rhapsody to use the default description generation rules. The property supports the following keywords:

\$Name - The element name

\$FullName - The full path of the element (P1::P2::C.a)

\$Description - The element description

Element-specific keywords, as shown in the following table:

Metatype Describes Additional Supported Keywords

Argument Arguments \$Type - The argument type

\$Direction - The argument direction (in, out, and so on)

Attribute Attributes \$Type - The attribute type

Class Classes, actors, objects, and blocks

Event Events \$Arguments - The description for the event argument

Operation Primitive operations, triggered operations, \$Arguments - The description for the operation argument

constructors, and destructors \$Signature - The operation signature

Package Packages

Relation Association ends \$Target - The other end of the association

Type Types \$Type - Applicable to Typedef types

Tag - The value of the specified the element tag

Property - The value of the element property with the specified name

The keywords are resolved in the following order:

Predefined keywords (such as \$Name)

Tag keywords

Property keywords

Note the following:

Keyword names can be written in parentheses. For example:

\$(Name)

If the value of a keyword is a MultiLine, each new line (except the first one) starts with the value of the

ADA\_CG::Configuration::DescriptionBeginLine

property; each line ends with the value of the ADA\_CG::Configuration::DescriptionEndLine property.

*(Default = )*

## EntryCondition

The EntryCondition property specifies the task guard.

*(Default = Empty string)*

## ExpressionFunctionForwardDeclPosition

This property is used in conjunction with property Ada\_CG.operation.IsExpressionFunction. It is used to generate a forward declaration for expression function, and to specify where the operation declaration appears. The possible values are:

None - no forward declaration is generated

BeforeClassRecord - the operation forward declaration appears before the class record (CR) declaration if CR has a visibility set to public, and before the class record forward declaration if CR has a visibility set to private

AfterClassRecord - the operation forward declaration appears after the class record declaration if CR has a visibility set to public, and after the class record forward declaration if CR has a visibility set to private

StartOfDeclaration - the operation forward declaration appears among the first declarations (together with other operations having the same settings) in the public section

EndOfDeclaration - the operation forward declaration appears among the last declarations (together with other operations having the same settings) in the public section

*(Default = None)*

## ExpressionFunctionForwardDeclVisibility

This property is used in conjunction with property Ada\_CG.operation.IsExpressionFunction and Ada\_CG\_operation.ExpressionFunctionForwardDeclPosition. It is used to specify in which section the operation declaration appears. The possible values are:

Public - the forward declaration will be generated in public section of specification file.

Protected - the forward declaration will be generated in private section of specification file.

Private - the forward declaration will be generated in body file.

*(Default = Public)*

## GenerateImplementation

The GenerateImplementation property specifies whether to generate the body for the operation.

To generate Import pragmas in Rational Rhapsody Developer for Ada, set this property to Cleared and add the "pragma..." declaration in the Ada\_CG::Operation::SpecificationEpilog property.

*(Default = True)*

## GenerateForwardDeclarationInPackageBody

The GenerateForwardDeclarationInPackageBody property allows generation of forward declaration of an operation in package body.

*(Default = True)*

## ImplementationEpilog

The ImplementationEpilog property adds code to the end of the definition of the model element.

*(Default = )*

## ImplementationProlog

The ImplementationProlog property adds code to the beginning of the definition of the model element

*(Default = )*

## ImplementationName

The ImplementationName property gives an operation one model name and generate it with another name. It is introduced as a workaround that generates const and non-const operations with the same name. For example:

Create a class A.

Add a non-const operation f().

Add a const operation f\_const().

Set the ADA\_CG::Operation::ImplementationName property for f\_const() to "f."

Generate the code.

The resulting code is as follows:



```

class A {

...

void f(); /* the non const f */

...

void f() const; /* actually f_const() */

...

};

```

The creation of two operations with the same signature, differing only in whether it is a const, is a common practice in C++, especially for STL users.

*(Default = Empty string)*

## Inline

The Inline property specifies how inline operations are generated. Which operations are affected by the Inline property depends on the metaclass:

Attribute - Applies only to operations that handle attributes (such as accessors and mutators)

Operation - Applies to all operations

Relation - Applies only to operations that handle relations

Inlining in Rational Rhapsody Developer for Ada

If the operation is marked as inline, the package specification includes an inline pragma after the declaration. For example:

```
pragma inline (operation name);
```

The two possible values for Rational Rhapsody Developer for Ada are as follows:

none (default)

use\_pragma

*(Default = none)*

## IsAnimationHelper

The IsAnimationHelper property indicates whether the operation should be generated only when animating the model.

*(Default = False)*

## IsEntry

The IsEntry property indicates whether the operation is a task entry or a regular operation in "AdaTask" and "AdaTaskType" classes.

*(Default = False)*

## Kind

The Kind property specifies the kind of operation that should be generated for an element. The kind of operations that can be generated is language-dependent (for example, virtual and abstract exist only in C++ and Java). In Java, Kind can be defined only for attributes and operations, but not for relations. This property affects class operations, in addition to accessors and mutators for relations and attributes. The possible values are as follows:

common - Class operations and accessor/mutator are non-virtual.

virtual - Class operations and accessor/mutator are virtual. This type is valid for C++ and Java only.

abstract - Class operations and accessor/mutator are pure virtual. This type is valid for C++ and Java only.

*(Default = common)*

## LocalVariablesDeclaration

The LocalVariablesDeclaration property specifies variables that you want to appear in the declaration of the entrypoint or operation.

*(Default = )*

## PreserveUserCode

This property controls the generation of tags that allow preservation of user changes to a file over successive generations.

*(Default = False)*

## Renames

The Renames property enables one element to rename another element of the same type. You can also rename an element by using a renames dependency. In the case of a conflict, the dependency has precedence. Note the following:

For attributes, this property works only for static attributes in a class or for attributes in a package.

For operations, this property contains the name of the operation this operation is renaming. The signatures of the two operations must match.

*(Default = Empty string)*

## **RenamesKind**

The RenamesKind property specifies whether the renaming of the operation designated in the ADA\_CG::Operation::Renames property is "as specification" or "as body."

*(Default = Specification)*

## **ReturnTypeByAccess**

The ReturnTypeByAccess property determines whether the return type is generated as an access type or a regular type.

Note that this property is applicable only to classes for which an access type is generated.

*(Default = None)*

## **SpecificationEpilog**

The SpecificationEpilog property adds code to the end of the declaration of the model element.

*(Default = )*

## **SpecificationProlog**

The SpecificationProlog property adds code to the beginning of the declaration of the model element.

*(Default = )*

## **TaskDefaultScheme**

The TaskDefaultScheme property sets the task default entry scheme. The possible values are as follows:

Conditional

Timed

None

*(Default = None)*

## TaskDefaultSchemeDelayStatement

The TaskDefaultScheme property sets the task default entry statement for timed entry schemes.

(Default = )

## ThisByAccess

The ThisByAccess property specifies whether to pass this parameter as an access mode parameter for a non-static operation.

(Default = False)

## ThisAccessTypeUsage

Controls whether the actual type for the “this” parameter is the class record type, the regular access type, or the class-wide access type.

None - “this” parameter is the class record type

Regular - “this” parameter is the regular access type

ClassWide - “this” parameter is the class-wide access type

(Default = None)

## ThisPassingMode

When set to a value other than FromGUI, it overrides the setting of the “constant” check box of an operation and the value of the Ada\_CG::Operation::ThisByAccess property.

(Default = FromGUI)

## ThisName

The ThisName property specifies the name of this parameter, which specifies the instance.

(Default = this)

## VirtualMethodGenerationScheme

The VirtualMethodGenerationScheme property enables compatibility with earlier versions for methods of interface and abstract classes. The possible values are as follows:

Default - The class type is class-wide, but these parameters are not.

ClassWideOperations - The class type is not class-wide, but these parameters are.

*(Default = Default)*

## ActivityReferenceToAttributes

The ActivityReferenceToAttributes property specifies whether Rational Rhapsody should generate references in the functor object, thereby giving you direct access to the attributes of the class that owns the modeled operation (without the need for this\_).

*(Default = Empty string)*

## ImplementActivityDiagram

The ImplementActivity Diagram property enables or disables code generation for activity diagrams.

*(Default = Empty string)*

## IsExplicit

The IsExplicit property is a Boolean value that allows you to specify that a constructor is an explicit constructor.

*(Default = Empty string)*

## IsNative

The IsNative property specifies whether the Java modifier “native” should be added to an operation in the source file. The body of such operations, if specified, is ignored by the code generator.

*(Default = Empty string)*

## MarkPrologEpilogInAnnotations

The MarkPrologEpilogInAnnotations property specifies whether to generate ignore annotations for the Specification, Implementation Prolog, and Epilog properties so they are ignored during roundtrip. When you insert code element declarations (variables, types, functions, and so on) in the Specification, Implementation Prolog, and Epilog properties, after a full roundtrip those elements are added to the model and are duplicated on the next code generation. When you use the MarkPrologEpilogInAnnotations

property, you can have Rational Rhapsody automatically ignore the information specified in the Specification, Implementation Prolog, and Epilog properties instead of adding the ignore annotations manually. The possible values for the MarkPrologEpilogInAnnotations property are as follows:

None - Rational Rhapsody does not generate any annotations. Any models created before Version 4.1 automatically have this property setting.

Ignore - Rational Rhapsody generates the `///  
 ignore` annotation before the code specified in the Specification, Implementation Prolog, and Epilog properties, and generates the `///  
]` annotation after the code specified

in those properties.

Auto - If the code in the Specification, Implementation Prolog, and Epilog properties is one line (it does not contain any newline characters (`\n`)), no annotations are generated (the same behavior has the None setting). If there is more than one line, Rational Rhapsody generates the `///  
 ignore` annotation before the code specified in the Specification, Implementation Prolog, and Epilog properties, and generates the `///  
]` annotation after the code specified in those properties (the same behavior as the Ignore setting).

During roundtrip, any ignore annotations in the comments of the element are not included in its description. Because the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties are generated between the annotation and its declaration for the element, you cannot rename those elements on roundtrip. If you change the name of an element, it is removed from the model and added with the new name. Some model information (for example, property settings) might be lost.

*(Default = Empty string)*

## Me

The Me property specifies the name of the first argument to operations generated in C.

*(Default = Empty string)*

## MeDeclType

The MeDeclType property is a string that specifies the type of the first argument to operations generated in C, as a pointer to an object

or object type. The default value is as follows:

`$ObjectName* const`

The variable \$ObjectName is replaced with the name of the object or object type.

*(Default = Empty string)*

## PrivateQualifier

The PrivateQualifier property specifies the qualifier that is printed at the beginning of a private operation declaration or definition. You can set this property to an empty string to prevent the generation of the static qualifier in the private function declaration or definition.

*(Default = Empty string)*

## ProtectedName

The ProtectedName property specifies the pattern used to generate names of private operations in C. The default value is as follows:

\$opName

The \$opName variable specifies the name of the operation. For example, the generated name of a private operation go() of an object A is generated as:

go()

*(Default = Empty string)*

## PublicName

The PublicName property specifies the pattern used to generate names of public operations in C. The default value is as follows:

\$objectName\_\$opName

The \$objectName variable specifies the name of the object; the \$opName variable specifies the name of the operation. For example, the generated name of a public operation go() of an object A is generated as:

A\_go()

*(Default = Empty string)*

## PublicQualifier

The PublicQualifier property specifies the qualifier that is printed at the beginning of a public operation declaration or definition. Note that the Static checkmark in the operation window UI is disabled in Rational Rhapsody Developer for C because the checkmark is associated with class-wide semantics that are not supported by Rational Rhapsody Developer for C. When loading models from previous versions, the Static check box is cleared; if the operation is public, the C\_CG::Operation::PublicQualifier property value is set to Static in order to generate the same code.

*(Default = Empty string)*

## ThrowExceptions

The ThrowExceptions property specifies the exceptions that an operation can throw. Separate multiple exceptions with commas.

*(Default = Empty string)*

## ReturnAsAccess

The ReturnAsAccess property enables generating return type of access kind. This property must be used in Ada2005

example

```
function my_function return access integer;
```

*(Default = False)*

## ReturnNotNullAccess

The ReturnNotNullAccess property enables generating return type of not null access kind. This property must be used in Ada2005

example

```
function my_function return not null access integer;
```

*(Default = False)*

## ReturnAccessConstant

The ReturnAccessConstant property enables generating return type of access constant kind. This property must be used in Ada2005

example

```
function my_function return access constant integer;
```

*(Default = False)*

## ThisNotNullAccess

The ThisNotNullAccess property enables generating instance argument type of not null access kind. This property must be used in Ada2005

example

```
procedures my_procedure(this : not null access integer);
```

*(Default = False)*

## ThisAccessConstant



The ThisAccessConstant property enables generating instance argument type of access constant kind. This property must be used in Ada2005

example

```
procedures my_procedure(this : access constant integer);
```

*(Default = False)*

## IsExpressionFunction

This property enables generating an expression function. The expression of the function is specified in the implementation of the operation in features window. The position of the function can be controlled with the operation's visibility and the property Ada\_CG.Operation.DeclarationPosition.

If a forward declaration is needed, the properties Ada\_CG.Operation.ExpressionFunctionForwardDeclPosition and Ada\_CG.Operation.ExpressionFunctionForwardDeclVisibility must be used.

*(Default = False)*

## Relation

The Relation metaclass contains properties that affect relations.

The properties are as follows:

- Add
- AddGenerate
- BidirectionalRelationsScheme
- Clear
- ClearGenerate
- CreateComponent
- CreateComponentGenerate
- DeleteComponent
- DeleteComponentGenerate
- DescriptionTemplate
- Find
- FindGenerate
- Get
- GetAt
- GetAtGenerate

- GetEnd
- GetEndGenerate
- GetGenerate
- ImplementWithStaticArray
- ImplementationEpilog
- ImplementationProlog
- InitializeComposition
- IsAliased
- ObjectInitialization
- Remove
- RemoveAt
- RemoveAtGenerate
- RemoveGenerate
- Set
- SetGenerate
- SpecificationEpilog
- SpecificationProlog
- Animate
- DataMemberVisibility
- GetKey
- GetKeyGenerate
- Inline
- Kind
- RemoveKey
- RemoveKeyGenerate
- RemoveKeyHelpersGenerate
- SafeInitScalar
- Static
- Visibility
- AccessTypeUsage

## Add

The Add property specifies the command used to add an item to a container.

*(Default = add\_\$(Name))*

## AddGenerate

This property specifies whether to generate an Add() operation for relations.

Setting this property to Cleared is one way to optimize your code for size.

*(Default = True)*

## BidirectionalRelationsScheme

The BidirectionalRelationsScheme property determines what implementation scheme will be used when code is generated for bidirectional relations. The possible values are:

SubtypingAndRenaming - The actual class members for the classes involved the bidirectional relation are all generated in the same package in order to have reciprocal visibility. The classes are "emulated" in packages made up of subtyping and renaming the class members.

IntermediateParentClasses - For each class involved in the bidirectional relation, an intermediate parent class is generated and inserted into the inheritance hierarchy. The bidirectional relations pointing to a class are redirected to point to its intermediate parent. Note that this implementation is not compatible with Ada83-only configurations.

The following limitations apply to the SubtypingAndRenaming implementation scheme:

The classes cannot be template classes.

The classes cannot contain elements with the same name (for types, static attributes, or association end role names), or signatures (for operations).

The classes cannot contain statechart code.

The classes cannot contain triggered operations.

Deferred initialization of public constants is not supported.

Roundtripping is not supported.

Ports are not supported.

*(Default = SubtypingAndRenaming)*

## Clear

The Clear property specifies the name of an operation that removes all items from a relation.

*(Default = clear\_<Name>)*

## ClearGenerate

This property specifies whether to generate a Clear() operation for relations.

Setting this property to Cleared is one way to optimize your code for size.

*(Default = True)*

## CreateComponent

The CreateComponent property specifies the name of an operation that creates a new component in a composite class.

*Default = New\_\$target:c*

*(Default = new\_\$Name)*

## CreateComponentGenerate

This property specifies whether to generate a CreateComponent operation for composite objects.

Setting this property to Cleared is one way to optimize your code for size.

*(Default = True)*

## DeleteComponent

The DeleteComponent property specifies the name of an operation that deletes a component from a composite class.

*(Default = delete\_\$Name)*

## DeleteComponentGenerate

This property specifies whether to generate a DeleteComponent() operation for composite objects.

Setting this property to Cleared is one way to optimize your code for size.

*(Default = True)*

## DescriptionTemplate

The DescriptionTemplate property specifies how to generate the element description in the code. An empty MultiLine (the default value) tells Rational Rhapsody to use the default description generation rules. The property supports the following keywords:

\$Name - The element name

\$FullName - The full path of the element (P1::P2::C.a)

\$Description - The element description

Element-specific keywords, as shown in the following table:

Metatype Describes Additional Supported Keywords

Argument Arguments \$Type - The argument type

\$Direction - The argument direction (in, out, and so on)

Attribute Attributes \$Type - The attribute type

Class Classes, actors, objects, and blocks

Event Events \$Arguments - The description for the event argument

Operation Primitive operations, triggered operations, \$Arguments - The description for the operation argument

constructors, and destructors \$Signature - The operation signature

Package Packages

Relation Association ends \$Target - The other end of the association

Type Types \$Type - Applicable to Typedef types

Tag - The value of the specified the element tag

Property - The value of the element property with the specified name

The keywords are resolved in the following order:

Predefined keywords (such as \$Name)

Tag keywords

Property keywords

Note the following:

Keyword names can be written in parentheses. For example:

\$(Name)

If the value of a keyword is a MultiLine, each new line (except the first one) starts with the value of the

ADA\_CG::Configuration::DescriptionBeginLine property; each line ends with the value of the ADA\_CG::Configuration::DescriptionEndLine property.

*(Default = )*

## Find

The Find property specifies the name of an operation that locates an item among relational objects.

*(Default = find\_\$(Name))*

## FindGenerate

This property specifies whether to generate a Find() operation for relations.

Setting this property to Cleared is one way to optimize your code for size.

*(Default = False)*

## Get

The Get property specifies the name of an operation that retrieves the relation currently pointed to by the iterator.

*(Default = get\_\$(Name))*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation by using an index. The ContainerTypes>RelationtypeGetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the at() operation for the container to retrieve the item at the indexed position:

```
$cname-at($index)
```

*(Default = get\_[[Name]]\_at\_pos)*

## GetAtGenerate

The GetAtGenerate property specifies whether to generate a getAt() operation for relations. The possible values are as follows:

Checked - Generate a getAt() operation for relations.

Cleared - Do not generate a getAt() operation for relations. Setting the GetAtGenerate property to False is one way to optimize your code for size.

*(Default = False)*

## **GetEnd**

The GetEnd property specifies the name of an operation that points the iterator to the last item in a collection.

*(Default = get\_[[Name]]End)*

## **GetEndGenerate**

The GetEndGenerate property specifies whether to generate a GetEnd() operation for relations.

*(Default = True)*

## **GetGenerate**

The GetGenerate property specifies whether to generate accessor operations for relations.

*(Default = True)*

## **ImplementWithStaticArray**

The ImplementWithStaticArray property specifies whether to implement relations as static arrays. The possible values are as follows:

Default - Rational Rhapsody provides the appropriate implementation for all fixed and bounded relations.

FixedAndBounded - All fixed and bounded relations are generated into static arrays.

To generate C-like code in C++ or Java, modify the value of the ImplementWithStaticArray property to FixedAndBounded.

*(Default = FixedAndBounded)*

## **ImplementationEpilog**

The ImplementationEpilog property adds code to the end of the definition of the model element.

*(Default = )*

## **ImplementationProlog**

The ImplementationProlog property adds code to the beginning of the definition of the model element

*(Default = )*

## **InitializeComposition**

The InitializeComposition property controls how a composition relation is initialized. The possible values are as follows:

InInitializer

InRecordType

None

*(Default = InInitializer)*

## **IsAliased**

The IsAliased property is a Boolean value that specifies whether attributes are aliased.

*(Default = False)*

## **ObjectInitialization**

The ObjectInitialization property specifies what kind of initialization occurs for the initial instances of a configuration. The possible values are as follows:

Full - Instances are initialized and their behavior is started.

Creation - Instances are initialized but their behavior is not started.

None - Instances are not initialized and their behavior is not started.

*(Default = Full)*

## **Remove**

The Remove property specifies the name of an operation that removes an item from a relation.

*(Default = remove\_{\$Name})*

## **RemoveAt**

*(Default = remove\_[[Name]]\_at\_pos)*



## RemoveAtGenerate

*(Default = False)*

## RemoveGenerate

This property specifies whether to generate a Remove() operation for relations.

Setting this property to Cleared is one way to optimize your code for size.

*(Default = True)*

## Set

The Set property specifies the name of the mutator generated for scalar relations.

*(Default = set\_ \$Name)*

## SetGenerate

This property specifies whether to generate mutators for relations.

Setting this property to Cleared is one way to optimize your code for size.

*(Default = True)*

## SpecificationEpilog

The SpecificationEpilog property adds code to the end of the declaration of the model element.

*(Default = )*

## SpecificationProlog

The SpecificationProlog property adds code to the beginning of the declaration of the model element.

*(Default = )*

## Animate

The Animate property specifies whether animation code is generated for an element. You can specify your own animation function by using the CG::Attribute::AnimSerializeOperation property. The semantics of the Animate property is always in favor of the owner settings:

If a package `Animate` property is set to `False`, all the classes owned by the package are not animated, regardless of the class `Animate` settings.

If a class `Animate` property is set to `False`, all the elements in the class (attributes, operations, relations, and so on) are not animated.

If an operation `Animate` property is set to `False`, all the arguments are not animated.

If the `AnimateArguments` property is set to `False`, all the arguments are not animated, regardless of the specific argument `Animate` property settings.

*(Default = Empty string)*

## **DataMemberVisibility**

The `DataMemberVisibility` property specifies the visibility of the relation data member. For example, if the relation is implemented as a pointer, this property determines whether the pointer data member is declared as `public`, `private`, or `protected`. The default value for `Ada` and `C` is `Private`; the default value for `C++` and `Java` is `Protected`.

*(Default = Empty string)*

## **GetKey**

The `GetKey` property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key. For example, the following command retrieves an item based on the key name by using the subscript operator `[]`, which has been overloaded according to the STL definition for maps:

```
$cname-operator[]($keyName)
```

*(Default = Empty string)*

## **GetKeyGenerate**

The `GetKeyGenerate` property specifies whether to generate `getKey()` operations for relations. Setting this property to `False` is one way to optimize your code for size.

*(Default = Empty string)*

## **Inline**

The `Inline` property specifies how inline operations are generated. Which operations are affected by the `Inline` property depends on the metaclass:

Attribute - Applies only to operations that handle attributes (such as accessors and mutators)

Operation - Applies to all operations

Relation - Applies only to operations that handle relations

### Inlining in Rational Rhapsody Developer for Ada

If the operation is marked as inline, the package specification includes an inline pragma after the declaration. For example:

```
pragma inline (operation name);
```

The two possible values for Rational Rhapsody Developer for Ada are as follows:

none

use\_pragma

*(Default = Empty string)*

## Kind

The Kind property specifies the kind of operation that should be generated for an element. The kind of operations that can be generated is language-dependent (for example, virtual and abstract exist only in C++ and Java). In Java, Kind can be defined only for attributes and operations, but not for relations. This property affects class operations, in addition to accessors and mutators for relations and attributes. The possible values are as follows:

common - Class operations and accessor/mutator are non-virtual.

virtual - Class operations and accessor/mutator are virtual. This type is valid for C++ and Java only.

abstract - Class operations and accessor/mutator are pure virtual. This type is valid for C++ and Java only.

*(Default = Empty string)*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation)

based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed:

```
$cname-erase($keyName)Default = remove$cname:c
```

*(Default = Empty string)*

## RemoveKeyGenerate

The RemoveKeyGenerate property specifies whether to generate a removeKey() operation for qualified relations. Setting this property to False is one way to optimize your code for size.

*(Default = Empty string)*

## RemoveKeyHelpersGenerate

The RemoveKeyHelpersGenerate property controls the generation of the relation helper methods (for example, \_removeItsX()

and \_\_removeItsX()). The possible values are as follows:

True - Generate the helpers whenever code generation analysis determines that the methods are needed.

False - Never generate the helpers.

FromModifier - Generate the helpers based on the value of the ADA\_CG::Relation::RemoveKey property.

*(Default = Empty string)*

## SafeInitScalar

The SafeInitScalar property specifies whether to initialize scalar relations as null pointers.

*(Default = Empty string)*

## Static

The Static property is a Boolean value that determines whether class-wide relations are enabled. Class-wide members of a class

are shared between all instances of that class and are mapped as static. When a relation is tagged as static:

The data member is generated as static (with the static keyword).

The relation accessors are generated as static.

The mutators of directional relations are generated as static. The mutators of symmetric relations are generated as common (non-static) operations.

Note the following behavior and restrictions:

If there are links between instances based on static relations, code generation initializes all the valid links. In case of a limited relation size, the last initialization is preserved.

When you generate instrumented code (animation or tracing), relation NOTIFY calls are not added to static relation mutators.

Animation associates static relations with the class instances, not the class itself.

In an instrumented application (animation or tracing), the static relations names appear in each instance node; however, the values of directional static relations are not visible.

See also the `CG::Relation::Containment`, `Containertype::Relationtype::CreateStatic`, and `Containertype::Relationtype::InitStatic` properties.

*(Default = Empty string)*

## Visibility

The Visibility property specifies the visibility of that kind of model element. Code generation maps the visibility specified for an element to the same visibility in the generated language.

*(Default = Empty string)*

## AccessTypeUsage

The AccessTypeUsage property defines the access type usage of relation's type.

*(Default = Regular)*

# Requirement

The Requirement metaclass contains properties that affect requirements.

The properties are as follows:

- DescriptionTemplate
- DescriptionTemplateForImplementation

## DescriptionTemplate

The DescriptionTemplate property specifies how to generate the element description in the code. If left blank, Rational Rhapsody uses the default description generation rules. The following variables can be used when setting the value of this property:

\$Name - The requirement name

\$FullName - The full path of the requirement (P1::P2::C.a)

\$ID - The requirement ID

\$Description - The requirement description

\$Specification - The requirement specification

*(Default = Realizes Requirement \$Name[[ # \$ID]])*

## DescriptionTemplateForImplementation

The DescriptionTemplateForImplementation property specifies how to generate the element description in the code. If left blank, Rational Rhapsody uses the default description generation rules. The following variables can be used when setting the value of this property:

\$Name - The requirement name

\$FullName - The full path of the requirement (P1::P2::C.a)

\$ID - The requirement ID

\$Description - The requirement description

\$Specification - The requirement specification

*(Default = Realizes Requirement \$Name[[ # \$ID]][[ : \$Specification]])*

## Statechart

The Statechart metaclass contains properties that affect statecharts.

The properties are as follows:

- HistoryConnectorDepth
- DescriptionTemplate

## HistoryConnectorDepth

This property controls the depth of history connectors. It is only effective when using the Ada 95 Behavioral framework. History connectors are always shallow when using the Ada 83 framework.

*(Default = Deep)*

## DescriptionTemplate

The DescriptionTemplate property specifies how to generate the element description in the code. An empty MultiLine (the default value) tells Rational Rhapsody to use the default description generation

rules. The property supports the following keywords:

\$Name - The element name

\$FullName - The full path of the element (P1::P2::C.a)

\$Description - The element description

Element-specific keywords, as shown in the following table:

Metatype Describes Additional Supported Keywords

Argument Arguments \$Type - The argument type

\$Direction - The argument direction (in, out, and so on)

Attribute Attributes \$Type - The attribute type

Class Classes, actors, objects, and blocks

Event Events \$Arguments - The description for the event argument

Operation Primitive operations, triggered operations, \$Arguments - The description for the operation argument

constructors, and destructors \$Signature - The operation signature

Package Packages

Relation Association ends \$Target - The other end of the association

Type Types \$Type - Applicable to Typedef types

Tag - The value of the specified the element tag

Property - The value of the element property with the specified name

The keywords are resolved in the following order:

Predefined keywords (such as \$Name)

Tag keywords

Property keywords

Note the following:

Keyword names can be written in parentheses. For example:

\$(Name)

If the value of a keyword is a MultiLine, each new line (except the first one) starts with the value of the ADA\_CG::Configuration::DescriptionBeginLine property; each line ends with the value of the ADA\_CG::Configuration::DescriptionEndLine property.

(Default = )

## Framework

The Framework metaclass contains properties that affect the Rational Rhapsody framework.

The properties are as follows:

- ActiveBase
- ActiveBaseUsage
- ActiveInit
- EventBaseUsage
- HeaderFile
- IncludeHeaderFile
- ProtectedBase
- ProtectedBaseUsage
- ProtectedInit
- ReactiveBase
- ReactiveBaseUsage
- ReactiveConsumeEventOperationName
- ReactiveInit
- ReactiveSetTask
- TimerMaxTimeouts
- TimerResolution
- EventBase
- ActivateFrameworkDefaultEventLoop
- ActiveDestructorGuard
- ActiveExecuteOperationName
- ActiveGuardInitialization
- ActiveIncludeFiles
- ActiveMessageQueueSize
- ActiveStackSize
- ActiveThreadName



- ActiveThreadPriority
- ActiveVtblName
- BooleanType
- CurrentEventId
- DefaultProvidedInterfaceName
- DefaultReactivePortBase
- DefaultReactivePortIncludeFiles
- DefaultRequiredInterfaceName
- EnableDirectReactiveDeletion
- EventIncludeFiles
- EventSetParamsStatement
- FrameworkInitialization
- InnerReactiveClassName
- InnerReactiveInstanceName
- InstrumentVtblName
- IsCompletedCall
- IsInCall
- MakeFileName
- NullTransitionId
- OperationGuard
- ProtectedClassDeclaration
- ProtectedIncludeFiles
- ReactiveCtorActiveArgDefaultValue
- ReactiveCtorActiveArgName
- ReactiveCtorActiveArgType
- ReactiveDestructorGuard
- ReactiveEnableAccessEventData
- ReactiveGuardInitialization
- ReactiveHandleEventNotConsumed
- ReactiveHandleTONotConsumed
- ReactiveIncludeFiles
- ReactiveInterface
- ReactiveSetEventHandlingGuard
- ReactiveVtblName
- SetManagedTimeoutCanceling
- SetRhp5CompatibilityAPI
- StaticMemoryIncludeFiles
- StaticMemoryPoolDeclaration

- StaticMemoryPoolImplementation
- TestEventTypeCall
- TimeoutId
- UseDirectReactiveDeletion
- UseManagedTimeoutCanceling
- UseRhp5CompatibilityAPI

## ActiveBase

The ActiveBase property specifies the superclass from which to specialize all threads, if the ActiveBaseUsage property is set to Checked.

*(Default = Empty string)*

## ActiveBaseUsage

The ActiveBaseUsage property specifies whether to use the superclass specified by the ActiveBase property as the superclass for all threads.

*(Default = False)*

## ActiveInit

The ActiveInit property specifies the format of the declaration generated for the initializer for an active class.

*(Default = Empty string)*

## EventBaseUsage

The EventBaseUsage property specifies whether to use the event superclass specified by the EventBase property as the parent of all events.

*(Default = False)*

## HeaderFile

The HeaderFile property specifies the framework header files to be included in objects that are within the scope of a particular configuration.

To optimize your code for size, leave the HeaderFile property blank. In this way, you can explicitly include the framework only when needed.

*(Default = Empty string)*

## **IncludeHeaderFile**

The IncludeHeaderFile property specifies whether to include the framework header files specified by the CG::Framework::HeaderFile property in the project.

*(Default = False)*

## **ProtectedBase**

The ProtectedBase property specifies the base class for protected objects, if the ProtectedBaseUsage property is set to Checked.

*(Default = Empty string)*

## **ProtectedBaseUsage**

The ProtectedBaseUsage property specifies whether to use the class specified by the ProtectedBase property as the base class for protected objects.

*(Default = False)*

## **ProtectedInit**

The ProtectedInit property specifies the declaration generated for the initializer for guarded objects.

*(Default = Empty string)*

## **ReactiveBase**

The ReactiveBase property specifies the base class for all reactive classes, if the ReactiveBaseUsage property is set to Checked.

*(Default = Empty string)*

## **ReactiveBaseUsage**

The ReactiveBaseUsage property specifies whether to use the class specified by the ReactiveBase property as the base class for all reactive objects.

*(Default = False)*

## **ReactiveConsumeEventOperationName**

The `ReactiveConsumeEventOperationName` property sets the user object virtual table for a reactive object. Follow these steps:

Create a procedure with the following signature:

```
procedure operation (a: in out RiAReactive; ev: in RiAEvent ev)
```

Use the `ReactiveConsumeEventOperationName` property to set the operation name.

Rational Rhapsody Developer for Ada ignores all the values for the properties under the Framework metaclass except for this one.

*(Default = Empty string)*

## ReactiveInit

The `ReactiveInit` property specifies the declaration for the initializer generated for reactive objects. The default pattern for C is as follows:

```
$base_init($member, (void*)$mePtr, $task, $VtblName);
```

The `$base` variable is replaced with the name of the reactive object during code generation. The string “\_init” is appended to the object name in the name of the operation. For example, if the reactive object is named A, the initializer generated for A is named `A_init()`. The `$member` variable is replaced with the name of the reactive member (equivalent to the base class) of the object during code generation. The `$mePtr` variable is replaced with the name of the user object (the value of the `Me` property). The member and `mePtr` objects are not equivalent if the user object is active. The `$VtblName` variable is replaced with the name of the virtual function table for the object, specified by the `ReactiveVtblName` property.

*(Default = Empty string)*

## ReactiveSetTask

The `ReactiveSetTask` property specifies the string that tells a reactive object whether it is an active or a sequential instance.

*(Default = Empty string)*

## TimerMaxTimeouts

The `TimerMaxTimeouts` property specifies the maximum number of timeouts allowed simultaneously in the system, if the `TimerMaxTimeouts`

property for the configuration is not overridden. In the framework, the default number of timers is 100.

*(Default = Empty string)*

## TimerResolution

The TimerResolution property specifies the length of time that must pass until the timer should check for matured timeouts. In the framework, the default number of timers is 100.

*(Default = Empty string)*

## EventBase

The EventBase property specifies the base class for all events, if the EventBaseUsage property is set to Checked.

*(Default = Empty string)*

## ActivateFrameworkDefaultEventLoop

The ActivateFrameworkDefaultEventLoop property specifies the framework call that initializes the framework main event loop.

*(Default = Empty string)*

## ActiveDestructorGuard

The ActiveDestructorGuard property specifies the macro that starts protection for an active user object destructor.

*Default = START\_DTOR\_THREAD\_GUARDED\_SECTION*

*(Default = Empty string)*

## ActiveExecuteOperationName

The ActiveExecuteOperationName property sets the user object virtual table for an active object and passes it to a task in the task initialization function (RiCTask\_init()). Follow these steps:

Create a method with the following signature:

```
struct RiCReactive * operation name (RiCTask * const)
```

Set the operation name in the ActiveExecuteOperationName property.

Start the execution of the active object task by calling the RICTASK\_START() macro on the object.

The virtual function table member name is stored in the ActiveVtblName property.

*(Default = Empty string)*

## ActiveGuardInitialization

The ActiveGuardInitialization property specifies the call that makes the active object event dispatching guarded.

*(Default = Empty string)*

## ActiveIncludeFiles

The ActiveIncludeFiles property specifies the base class for threads when you use selective framework includes. If a class is active and this property is defined, the file specified by the property is included in the class specification file.

*(Default = Empty string)*

## ActiveMessageQueueSize

The ActiveMessageQueueSize

property specifies the size of the message queue allocated for active objects, if the ActiveMessageQueueSize property for classes is left blank. The default value for the size of the message queue is language-dependent, as follows:

C - The default value is RiCOSDefaultMessageQueueSize, which is a variable set in the implementation file of the OS adapter for a given operating system.

C++ - The default value is OMOSThread::DefaultMessageQueueSize.

Java - The default value is an empty string (blank).

*(Default = Empty string)*

## ActiveStackSize

The ActiveStackSize property specifies the size of the stack allocated for active objects, if the ActiveStackSize property for classes

is left blank. The default value for the stack size is language-dependent, as follows:

C - The default value is RiCOSDefaultStackSize, which is a variable set in the implementation file of the OS adapter for a given operating system.

*(Default = Empty string)*

## ActiveThreadName

The ActiveThreadName property specifies the name of threads, if the ActiveThreadName property for classes is left blank. The default values are as follows:

A string - Names the active thread.

An empty string (blank) - The value is set in an operating system-specific manner, based on the value of the ActiveThreadName property for the framework.

*(Default = Empty string)*

## ActiveThreadPriority

The ActiveThreadPriority priority specifies the priority of threads, if the ActiveThreadPriority property for classes is left blank. The default value for the priority of threads is language-dependent, as follows:

*(Default = Empty string)*

## ActiveVtblName

The ActiveVtblName property stores the name of the virtual function table that is associated with a task (the RiCTask member of the structure).

*(Default = Empty string)*

## BooleanType

The BooleanType property specifies the Boolean type used by the framework.

*(Default = Empty string)*

## CurrentEventId

The CurrentEventId property specifies the call or macro used to obtain the ID of the currently consumed event.

*Default = OM\_CURRENT\_EVENT\_ID*

*(Default = Empty string)*

## DefaultProvidedInterfaceName

The DefaultProvidedInterfaceName property specifies the interface that must be implemented by the “in” part of a rapid port.

*(Default = Empty string)*

## **DefaultReactivePortBase**

The DefaultReactivePortBase

property stores the base class for the generic rapid port (or default reactive port). This base class relays all events.

*(Default = Empty string)*

## **DefaultReactivePortIncludeFiles**

The DefaultReactivePortIncludeFiles property specifies the include files that are referenced in the generated file that implements the class with the rapid ports.

*(Default = Empty string)*

## **DefaultRequiredInterfaceName**

The DefaultRequiredInterfaceName property specifies the interface that must be implemented by the “out” part of a rapid port.

*(Default = Empty string)*

## **EnableDirectReactiveDeletion**

The EnableDirectReactiveDeletion property specifies the call to the framework that supports direct deletion of reactive instances (by using the delete operator) instead of graceful framework termination (by using the reactive destroy() method). When you use destroy(), the object waits in a zombie mode until all the events that are designated to it are removed from the active context queue, and then self -destructs. In this scheme, there is no need to traverse the queue of the active context to cancel pending events, and there is no need to make the reactive destructor guarded to ensure safe deletion. A reactive object can be either in a graceful termination or forced deletion (by using the delete operator) state: you cannot use graceful deletion on an object that allows forced deletion, and vice versa. You can set a single reactive object in a forced deletion state, or set the entire system (all reactive instances) in a forced deletion state (as is done for compatibility with earlier versions). Graceful termination should not be used when a reactive part (of a composite class) runs in a context of an active object that is not part of, and different from, the composite active context. If you are using a Rational Rhapsody library component as part of an application where the main is not generated by Rational Rhapsody (for example, GUI applications), the framework initializes itself in full compatibility mode on the call to OXF::init(). If you want to remove part or all of the compatibility features, call OXF::initialize() instead of OXF::init() (the operation takes the same arguments) and add independent activation calls that are compatible with earlier versions, prior

to the initialize() call. Note that the ADA\_CG::Framework::UseDirectReactiveDeletion property must be set to True for this property to take effect. When it is set to True, the code specified in the EnableDirectReactiveDeletion is generated in the main prior to the call to OXF::init().



*(Default = Empty string)*

## EventIncludeFiles

The EventIncludeFiles property specifies the base class for events when you use selective framework includes. If events are defined in a package, the file specified by this property is included in the package specification file to enable the use of events and timeouts in the package. The default value for C is as follows:

oxf/RiCEvent.h

*(Default = Empty string)*

## EventSetParamsStatement

The EventSetParamsStatement property specifies a template for the body of the setParams() method, provided by the Rational Rhapsody framework

for Java, to set the parameters of an event. For example, for an event of type evOn(), the default template would generate the following code in the body of the setParams()

method:

evOn params = (evOn) event; The default value is as follows:

\$eventType params = (\$eventType) event;

*(Default = Empty string)*

## FrameworkInitialization

The FrameworkInitialization property specifies the framework initialization code that is called by the main. The default value is as follows: OXF::initialize\$(Argc)\$ (Argv)\$ (AnimationPortNumber) \$(RemoteHost)\$ (TimerResolution)\$ (TimerMaxTimeouts) \$(TimeModel))

*(Default = Empty string)*

## InnerReactiveClassName

The InnerReactiveInstanceName property specifies the name of a reactive class that serves as a bridge between a reactive class in your model and the framework. The implementation scheme of reactive classes is different in Java than in C++. Java does not allow inheritance from the reactive framework classes because that would mean that you would not be able to inherit from an additional base class that might not be reactive. The chosen alternative is to delegate an inner class instance that inherits from RiJStateReactive. Delegation is the implementation of an interface that forwards relevant messages to the inner class instance.

*(Default = Empty string)*

## InnerReactiveInstanceName

The InnerReactiveInstanceName property specifies the name of a reactive instance that serves as a bridge between a reactive class in your model and the framework. The implementation scheme of reactive classes is different in Java than in C++. Java does not allow inheritance from the reactive framework classes because that would mean that you would not be able to inherit from an additional base class that might not be reactive. The chosen alternative is to delegate an inner class instance that inherits from RiJStateReactive. Delegation is the implementation of an interface that forwards relevant messages to the inner class instance.

*(Default = Empty string)*

## InstrumentVtblName

The InstrumentVtblName property specifies the name of the virtual function table that is associated with animation objects. Each animated object has

its own virtual function table (Vtbl). This table creates your own framework, with its own virtual instrumentation functions, and connect it to Rational Rhapsody.

*(Default = Empty string)*

## IsCompletedCall

The IsCompletedCall property specifies the call or macro that determines whether the state reached a final state so it can be exited on a null transition. The property supports the \$State keyword so you can use state-based calls. The keyword is resolved to the state implementation (code) name.

*(Default = Empty string)*

## IsInCall

The IsInCall property specifies the query that determines whether the state is in the current active configuration. The property supports the \$State keyword so you can use state-based calls. The keyword is resolved to the state implementation (code) name.

*(Default = Empty string)*

## MakeFileName

The MakeFileName property specifies a new name for the makefile. To use this property, add the following line to the .prp file:

Property MakeFileName String "MyFileName"

In this syntax, MyFileName specifies the name of the makefile.

*(Default = Empty string)*

## **NullTransitionId**

The NullTransitionId property specifies the ID reserved for null transition consumption.

*(Default = Empty string)*

## **OperationGuard**

The OperationGuard property specifies the macro that guards an operation.

*(Default = Empty string)*

## **ProtectedClassDeclaration**

The ProtectedClassDeclaration property affects how protected classes are implemented. Instead of inheriting from OMProtected, the class embeds an aggregate OMProtected. The aggregate member and helper methods are defined in the macro OMDECLARE\_GUARDED (defined in omprotected.h).

*(Default = Empty string)*

## **ProtectedIncludeFiles**

The ProtectedIncludeFiles property specifies the base class for protected classes when you use selective framework includes. The default value for C is as follows:

oxf/RiCProtected.h

*(Default = Empty string)*

## **ReactiveCtorActiveArgDefaultValue**

The ReactiveCtorActiveArgDefaultValue property specifies the default value of the active context argument in a reactive constructor.

*(Default = Empty string)*

## **ReactiveCtorActiveArgName**

The ReactiveCtorActiveArgDefaultValue property specifies the name of the active context argument in a reactive constructor.

*(Default = Empty string)*

## **ReactiveCtorActiveArgType**

The ReactiveCtorActiveArgDefaultValue property specifies the type of the active context argument in a reactive constructor.

*(Default = Empty string)*

## **ReactiveDestructorGuard**

The ReactiveDestructorGuard property specifies the macro that starts protection of a section of code used for destruction of a reactive

instance. This prevents a “race” (between the deletion and event dispatching) when deleting an active instance.

*(Default = Empty string)*

## **ReactiveEnableAccessEventData**

The ReactiveEnableAccessEventData property specifies the code to be used to enable access to the specific event data in a transition (typically

by assigning a local variable of the appropriate type). The property supports the \$Event keyword so you can specify the event type.

*(Default = Empty string)*

## **ReactiveGuardInitialization**

The ReactiveDestructorGuard property specifies the framework call that makes the event consumption of a specific reactive class guarded.

*(Default = Empty string)*

## **ReactiveHandleEventNotConsumed**

The ReactiveHandleEventNotConsumed property registers a method to handle unconsumed events in a reactive class. Specify the method name as this value for this property.

*(Default = Empty string)*

## **ReactiveHandleTONotConsumed**

The ReactiveHandleTONotConsumed property registers a method to handle unconsumed trigger operations in a reactive class. Specify the method name as the value for this property.

*(Default = Empty string)*

## ReactiveIncludeFiles

The ReactiveIncludeFiles property specifies the base classes for reactive classes when you use selective framework includes. If a class is reactive and this property is defined, the file specified by the property is included in the class specification file. For reactive classes, the header files specified by the following properties are also included:

EventIncludeFiles - For the event base class

ActiveIncludeFiles - If the class is guarded or instrumented

The default value for C is as follows:

oxf/RiCReactive.h

*(Default = Empty string)*

## ReactiveInterface

The ReactiveInterface property specifies the name of the interface class that forwards messages to an inner class instance of a reactive

class in order to implement its reactive behavior.

*(Default = Empty string)*

## ReactiveSetEventHandlingGuard

The ReactiveSetEventHandlingGuard property controls the code that is generated within the constructor of a reactive class. When you use this property with guarded triggered operations, it enables guarding of the event handling (in order to provide mutual exclusion between the event and TO handling).

*(Default = Empty string)*

## ReactiveVtblName

The ReactiveVtblName property specifies the name of the virtual function table (Vtbl) associated with a reactive object. Each reactive object

has its own Vtbl, which creates your own framework and connect it to Rational Rhapsody.

*(Default = Empty string)*

## SetManagedTimeoutCanceling

The SetManagedTimeoutCanceling property is a property for compatibility with earlier versions that specifies whether the framework uses a Rational Rhapsody version earlier than the 6.0 scheme of timeout creation and cancellation (where OMTimerManager is responsible for cancellation of timeouts) or the Rational Rhapsody 6.0 scheme. In Rational Rhapsody 6.0, the framework moves the responsibility for a timeout cancellation from the timer manager to the timeout client (the reactive object). This change reduces the timer manager responsibilities and the overhead in timeout management (thus improving timeout scheduling performance). The change also includes changes in the generated code (the user reactive objects hold pointers to the waiting timeouts in order to enable canceling). If you are using a Rational Rhapsody library component as part of an application where the main is not generated by Rational Rhapsody (for example, GUI applications), the framework initializes itself in full compatibility mode on the call to OXF::init(). If you want to remove part or all of the compatibility features, call OXF::initialize() instead of OXF::init() (the operation takes the same arguments) and add independent activation calls that are compatible with earlier versions, prior to the initialize() call.

*(Default = Empty string)*

## SetRhp5CompatibilityAPI

The SetRhp5CompatibilityAPI property specifies the call that configures models created before Rational Rhapsody 6.0 so they use the 5. x version of the framework instead of the new one. See UseRhp5CompatibilityAPI for more information on Version 5. x compatibility mode.

*(Default = Empty string)*

## StaticMemoryIncludeFiles

The StaticMemoryIncludeFiles property specifies the files to be included in the package specification file if static memory management is enabled and you are using selective framework includes.

*(Default = Empty string)*

## StaticMemoryPoolDeclaration

The StaticMemoryPoolDeclaration property specifies the declaration of the memory pool for timeouts. The default value follows:

```
DECLARE_MEMORY_ALLOCATOR($Class, $BaseNumberOfInstances)
```

*(Default = Empty string)*

## StaticMemoryPoolImplementation

The StaticMemoryPoolImplementation property specifies the generated code in the implementation file

for a memory pool implementation (see the `BaseNumberOfInstances` property). The default value is as follows:

```
IMPLEMENT_MEMORY_ALLOCATOR($Class, $BaseNumberOfInstances,  
$AdditionalNumberOfInstances, $ProtectStaticMemoryPool)
```

*(Default = Empty string)*

## TestEventTypeCall

The `TestEventTypeCall` property specifies the test used in event consumption code to check if the currently consumed event is of a given type.

*(Default = Empty string)*

## TimeoutId

The `TimeoutId` property specifies the ID reserved for timeout events.

*(Default = Empty string)*

## UseDirectReactiveDeletion

The `UseDirectReactiveDeletion` property determines whether direct deletion of reactive instances (by using the delete operator) is used instead of graceful framework termination (by using the reactive `destroy()` method). When this property is set to `True`, the code specified in the `EnableDirectReactiveDeletion` is generated in the main prior to the call to `OXF::init()`. See `EnableDirectReactiveDeletion` and the upgrade history on the support site for more information on this functionality.

*(Default = Empty string)*

## UseManagedTimeoutCanceling

The `UseManagedTimeoutCanceling` property specifies whether the framework uses the pre-Rational Rhapsody 6.0 scheme of timeout creation and cancellation (so `OMTimerManager` is responsible for cancellation of timeouts). The framework moves the responsibility for a timeout cancellation from the timer manager to the timeout client (the reactive object). This change reduces the timer manager responsibilities and the overhead in timeout management (thus improving timeout scheduling performance). The change also includes changes in the generated code (the user reactive objects hold pointers to the waiting timeouts in order to enable canceling). When loading a pre-6.0 model, Rational Rhapsody sets the project `ADA_CG::Framework::UseManagedTimeoutCanceling` to `True` to set the system-compatibility mode. See the upgrade history on the support site for more information.

*(Default = Empty string)*

## UseRhp5CompatibilityAPI

The UseRhp5CompatibilityAPI property specifies whether to use the virtual functions of the core implementation classes that existed in the pre-Rational Rhapsody 6.0 framework. The Rational Rhapsody 6.0 framework introduces a set of interfaces for the core behavioral framework. The interfaces define a concise API for the framework and allow you to replace the actual implementation of these interfaces while maintaining the framework behavior. As a result of the interfaces' introduction, the framework behavioral classes (OMReactive, OMThread, and OMEvent) use a new set of virtual operations to implement the interfaces and provide the behavioral infrastructure. To support existing customizations of these classes (made by inheriting and overriding the virtual operations), the framework can work in a mode where the pre-6.0 API virtual operations are called. When loading a pre-6.0 model, Rational Rhapsody sets the project property ADA\_CG::Framework::UseRhp5CompatibilityAPI to True to set the system-compatibility mode. If this is set to True, the pre-6.0 API is called by the framework instead of the interface-based API. Without this flag, user customizations compile but are not called.

*(Default = Empty string)*

## GNAT

This metaclass contains the properties that manipulate the GNAT operating system environment.

The properties are as follows:

- AdditionalReservedWords
- BuildCommandSet
- buildFrameworkCommand
- CompileCommand
- CompileSwitches
- CPPCompileDebug
- CPPCompileRelease
- DependencyRule
- EntryPoint
- ErrorMessageTokensFormat
- ExeExtension
- FileDependencies
- ImpExtension
- Include
- InvokeAnimatedExecutable
- InvokeExecutable
- InvokeMake
- IsFileNameShort



- LibExtension
- LinkDebug
- LinkRelease
- LinkSwitches
- MakeExtension
- MakeFileContent
- MakeFileNameForExe1
- MakeFileContentForExe1
- MakeFileNameForExe2
- MakeFileContentForExe2
- MakeFileNameForLib
- MakeFileContentForLib
- OSFileSystemCaseSensitive
- ObjCleanCommand
- ObjExtension
- ObjectName
- ObjectsDirectory
- ParseErrorMessage
- ParseSeverityError
- ParseSeverityWarning
- ParseErrorDescript
- UseNewBuildOutputWindow
- ParseErrorMoreInfo
- ParseMakeError
- QuoteOMROOT
- RemoteHost
- SpecExtension
- UseNonZeroStdInputHandle
- UseRemoteHost
- AdaptorSearchPath
- BLDAdditionalDefines
- BLDAdditionalOptions
- BLDIncludeAdditionalBLD
- BLDMainExecutableOptions
- BLDMainLibraryOptions
- BLDTarget
- BriefErrorMessage
- BSP

- BSP\_Libraries
- CodeTestSettings
- COM
- ConvertHostToIP
- CPPAdditionalReservedWords
- CPPCompileCommand
- CPPCompileSwitches
- CPU
- DebugSwitches
- DEFExtension
- DllExtension
- DuplicateLibsListInMakeFile
- EntryPointDeclarationModifier
- EnvironmentVarName
- ESTLCompliance
- FrameworkLibPrefix
- GeneratedAllDependencyRule
- GetConnectedRuntimeLibraries
- HasIDEInterface
- IDEInterfaceDLL
- InvokeCodeGeneration
- InvokeMakeGenerator
- MainIncludes
- NullValue
- OMCPU
- OMCPU\_SUFFIX
- OpenHTMLReports
- PathDelimiter
- ProcessToKillAtStopExec
- RCCCompileCommand
- RCExtension
- ReusableStatechartSwitches
- RPFrameWorkDll
- SpecFilesInDependencyRules
- SubSystem
- TargetConfigurationFileName
- UnixLineTerminationStyle
- UnixPathNameForOMROOT

- UseActorsCode

## AdditionalReservedWords

The `AdditionalReservedWords` property is a string that specifies additional reserved keywords that Rational Rhapsody does not

allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody. The property value is checked at run time when you name or rename an element, based on the active configuration environment setting. Note that this property affects the algorithm only when the active configuration is of the selected environment.

(Default = Empty string)

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration. To change this property, use the Configuration window in the browser - do not change it by using the Properties window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link. The possible values are as follows:

Debug - Generate the debug command that is set in the makefile.

DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

Release - Generate the release command that is set in the makefile.

**ReleaseNoExp** - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

(Default = Debug)

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The `buildFrameworkCommand` property is used to specify the command that should be carried out when the Build Framework option is selected.

```
(Default = \"$OMROOT/etc/Executer.exe\"  
|\"|\"|$OMROOT\\..\\Sodius\\RiA_CG\\bin\\recompile_GNAT.bat||\"|\"|\"|
```

## CompileCommand

The `CompileCommand` property is a string that specifies a different compile command.

(Default = )

## CompileSwitches

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property.

*(Default = )*

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*(Default = Empty string)*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*(Default = Empty string)*

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

For example, the following dependency rule lists the file

dependencies for a Windows application with a GUI, including bitmaps,

icons, and resource files:

```
$OMFileObjPath : $OMFileImpPath "*.bmp" "*.ico" "*.rc2"
```

*(Default = Empty string)*

## EntryPoint

The EntryPoint property specifies the name of the main program for a given environment.

See also the definition of the EntryPointDeclarationModifier property for more information.

*(Default = Empty string)*

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens has three parameters, each with an integer value:

TotalNumberOfTokens - The number of tokens in the regular expression

FileTokenPosition - The position of the file name token in the expression

LineTokenPosition - The position of the line number token in the expression

*(Default = TotalNumberOfTokens=4,FileTokenPosition=1,LineTokenPosition=2)*

## ExeExtension

The ExeExtension property specifies the extension that is appended to compiled executable components for a given environment.

*(Default = .exe)*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included

in model elements. The file inclusions are generated in the makefile.

*(Default = Empty string)*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment.

*(Default = .adb)*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*(Default = Empty string)*

## InvokeAnimatedExecutable

The InvokeExecutable property specifies the command used to run an animated executable file.

*(Default = \$executable \$port)*

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*(Default = \"\$executable\")*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

```
"$OMROOT/etc/Executer.exe" "$OMROOT\etc\GnatMake.bat $makefile $maketarget"
```

*(Default = \"\$OMROOT/etc/Executer.exe\" \"\\\"\$OMROOT\etc\\GnatMake.bat\\\" \$makefile \$maketarget\")*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format.

If this is Checked:

The file name is not truncated.

If the FileName property is not blank, its value overrides any automatic file name synthesis.

If the file name is longer than eight characters, the Checker reports this prior to code generation.

*(Default = False)*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*(Default = .a)*

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

*(Default = Empty string)*

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

*(Default = Empty string)*

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

*(Default = Empty string)*

## MakeExtension

The MakeExtension property specifies the extension that Rational Rhapsody appends to makefiles.

*(Default = .bat)*

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

Target type

Compilation flags

Commands definitions

Generated macros

Predefined macros

Generated dependencies

Linking instructions

The following sections describe the contents of the makefile in detail.

Target Type

The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration. For example, the default content of the target type section of a C++ makefile for the Microsoft environment is as follows:

```
##### Target type (Debug/Release) #####
#####

CPPCompileDebug=$OMCPPCompileDebug

CPPCompileRelease=$OMCPPCompileRelease

LinkDebug=$OMLinkDebug

LinkRelease=$OMLinkRelease

BuildSet=$OMBuildSet

SUBSYSTEM=$OMSubSystem

COM=$OMCOM

RPFrameWorkDll=$OMRPFrameWorkDll

ConfigurationCPPCompileSwitches=
$OMReusableStatechartSwitches $OMConfigurationCPPCompileSwitches

!IF "$ (RPFrameWorkDll)" == "True"

ConfigurationCPPCompileSwitches=
$(ConfigurationCPPCompileSwitches) /D "FRAMEWORK_DLL"

!ENDIF

!IF "$ (COM)" == "True"

SUBSYSTEM=/SUBSYSTEM:windows

!ENDIF
```

Compilation Flags



The compilation flags section of the makefile contains the default compilation flags stored in the CompileSwitches property. For example, the default content of the compilation flags section of a C++ makefile for the Microsoft environment is as follows:

```
##### Compilation flags #####  
#####
```

INCLUDE\_QUALIFIER=/I

LIB\_PREFIX=MS

#### Commands Definitions

The commands definition section

of the makefile specifies programs to execute from the makefile. For example, the default commands definition section of a C++ makefile for the Microsoft environment is as follows:

```
##### Commands definition #####  
#####
```

RMDIR = rmdir

LIB\_CMD=link.exe -lib

LINK\_CMD=link.exe

LIB\_FLAGS=\$OMConfigurationLinkSwitches

LINK\_FLAGS=\$OMConfigurationLinkSwitches \$(SUBSYSTEM) /

MACHINE:I386

#### Generated Macros

The generated macros section of the makefile contains a variable that expands to the Rational Rhapsody-generated macros in the makefile. For example:

```
##### Generated macros #####  
#####
```

\$OMContextMacros

OBJ\_DIR=\$OMObjectsDir

!IF "\$(OBJ\_DIR)"!=""

CREATE\_OBJ\_DIR=if not exist \$(OBJ\_DIR) mkdir \$(OBJ\_DIR)

```
CLEAN_OBJ_DIR= if exist $(OBJ_DIR) $(RMDIR) $(OBJ_DIR)
```

```
!ELSE
```

```
CREATE_OBJ_DIR=
```

```
CLEAN_OBJ_DIR=
```

```
!ENDIF
```

The `$OMContextMacros` keyword expands several macros in the makefile. Each makefile macro has its own keyword. You can use these keywords separately to customize the makefile. The `$OMContextMacros` variable modifies target-specific variables. Replace the `$OMContextMacros` line in the `MakeFileContent`

property with the following:

```
FLAGSFILE=$OMFlagsFile
```

```
RULESFILE=$OMRulesFile
```

```
OMROOT=$OMROOT
```

```
CPP_EXT=$OMImplExt
```

```
H_EXT=$OMSpecExt
```

```
OBJ_EXT=$OMObjExt
```

```
EXE_EXT=$OMExeExt
```

```
LIB_EXT=$OMLibExt
```

```
INSTRUMENTATION=$OMInstrumentation
```

```
TIME_MODEL=$OMTimeModel
```

```
TARGET_TYPE=$OMTargetType TARGET_NAME=$OMTargetName
```

```
$OMAllDependencyRule
```

```
TARGET_MAIN=$OMTargetMain LIBS=$OMLibs
```

```
INCLUDE_PATH=$OMIncludePath
```

```
ADDITIONAL_OBJS=$OMAdditionalObjs
```

```
OBJS= $OMObjs
```

Predefined Macros

The predefined macros section of the makefile contains other macros than the Rational Rhapsody-generated macros specified in the generated macros section. For example, the default predefined macros section of a C++ makefile for the Microsoft environment is as follows:

```
##### Predefined macros #####

#####

$(OBJS) : $(INST_LIBS) $(OXF_LIBS)

LIB_POSTFIX=

!IF "$(BuildSet)"=="Release"

LIB_POSTFIX=R

!ENDIF

!IF "$(TARGET_TYPE)" == "Executable"

LinkDebug=$(LinkDebug) /DEBUG

LinkRelease=$(LinkRelease) /OPT:NOREF

!ELSEIF "$(TARGET_TYPE)" == "Library"

LinkDebug=$(LinkDebug) /DEBUGTYPE:CV

!ENDIF

!IF "$(INSTRUMENTATION)" == "Animation"

INST_FLAGS=/D "OMANIMATOR"

INST_INCLUDES=/I $(OMROOT)\LangCpp\om /I

$(OMROOT)\LangCpp\tom

!IF "$(RPFrameWorkDll)" == "True"

INST_LIBS=

OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfanimdll$(

LIB_POSTFIX)

$(LIB_EXT)

!ELSE
```

```

INST_LIBS=

$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomanim$(LIB_POSTFIX)

(LIB_EXT)

OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfinst$(LIB
POSTFIX)$(LIB_EXT)

$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX)$(LIB_EXT)

!ENDIF

SOCK_LIB=wsock32.lib

!ELSEIF "$(INSTRUMENTATION)" == "Tracing"

INST_FLAGS=/D "OMTRACER"

INST_INCLUDES=/I $(OMROOT)\LangCpp\om /I
$(OMROOT)\LangCpp\tom

!IF "$(RPFrameWorkDll)" == "True"

INST_LIBS=

OXF_LIBS=

$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfracedll$(LIB_POST
FIX)$(LIB_EXT)

!ELSE

INST_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)tomtrace$
(LIB_POSTFIX)

$(LIB_EXT)

$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomtrace$(LIB_POSTFIX)

$(LIB_EXT)

OXF_LIBS=

$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfinst$(LIB_POSTFIX)

```

```

(LIB_EXT) $(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX)$(LIB_EXT)

!ENDIF

SOCK_LIB=wsock32.lib

!ELSEIF "$(INSTRUMENTATION)" == "None"

INST_FLAGS=

INST_INCLUDES=

INST_LIBS=

!IF "$(RPFrameWorkDll)" == "True"

OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfdll$

(LIB_POSTFIX)$(LIB_EXT)

!ELSE

OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$

(LIB_POSTFIX)$(LIB_EXT)

!ENDIF

SOCK_LIB=

!ELSE

!ERROR An invalid Instrumentation $(INSTRUMENTATION) is specified. !ENDIF

```

## Generated Dependencies

The generated dependencies section of the makefile contains a variable that expands to Rational Rhapsody-generated dependencies and compilation instructions. For example, the generated dependencies section of a C++ makefile for the Microsoft environment is as follows:

```
##### Generated dependencies #####
```

```
#####
```

```
$OMContextDependencies
```

```
$OMFileObjPath : $OMMainImplementationFile $(OBJS)
```

```
$(CPP) $(ConfigurationCPPCompileSwitches) /Fo"$OMFileObjPath" $OMMainImplementationFile
```

## Linking Instructions

The linking instructions section

of the makefile contains the predefined linking instructions. For example, the default linking instructions section of a C++ makefile for the Microsoft environment is as follows:

```
##### Linking instructions #####

#####

$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS)

$OMFileObjPath $OMMakefileName $OMModelLibs

@echo Linking $(TARGET_NAME)$(EXE_EXT)

$(LINK_CMD) $OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \

$(LIBS) \

$(INST_LIBS) \

$(OXF_LIBS) \

$(SOCK_LIB) \

$(LINK_FLAGS) /out:$(TARGET_NAME)$(EXE_EXT)

$(TARGET_NAME)$(LIB_EXT) : $(OBJS) $(ADDITIONAL_OBJS)

$OMMakefileName

@echo Building library $@

$(LIB_CMD) $(LIB_FLAGS) /out:$(TARGET_NAME)$(LIB_EXT)

$(OBJS) $(ADDITIONAL_OBJS)

clean:

@echo Cleanup

$OMCleanOBJS

if exist $OMFileObjPath erase $OMFileObjPath

if exist *$(OBJ_EXT) erase *$(OBJ_EXT)

if exist $(TARGET_NAME).pdb erase $(TARGET_NAME).pdb

if exist $(TARGET_NAME)$(LIB_EXT) erase
```

`$(TARGET_NAME)$(LIB_EXT)`

if exist `$(TARGET_NAME).ilk` erase `$(TARGET_NAME).ilk`

if exist `$(TARGET_NAME)$(EXE_EXT)` erase

`$(TARGET_NAME)$(EXE_EXT)`

`$(CLEAN_OBJ_DIR)`

*(Default = )*

## **MakeFileNameForExe1**

This property is used to specify the filename for the makefile generated by Rhapsody from the corresponding `MakeFileContentForExe` property.

Note that this property specifies the entire filename, including the extension.

*(Default = gnat.adc)*

## **MakeFileContentForExe1**

This property defines the content of a file whose name is defined in the corresponding `MakeFileNameForExe` property.

The code generator will create the file and take the file content from the value of this property.

The value of the property can contain text and macros. Macro replacement depends on the code generator rules called by the macro. See the RiA CG User Guide for details about the list of macros.

This property is used for executable components, while the `MakeFileContentForLib` property is used for library components.

*(Default = \$AdaCGGnatAdc)*

## **MakeFileNameForExe2**

This property is used to specify the filename for the makefile generated by Rhapsody from the corresponding `MakeFileContentForExe` property.

Note that this property specifies the entire filename, including the extension.

*(Default = \$ComponentName\$MakeExtension)*

## **MakeFileContentForExe2**

This property defines the content of a file whose name is defined in the corresponding `MakeFileNameForExe` property.

The code generator will create the file and take the file content from the value of this property.

The value of the property can contain text and macros. Macro replacement depends on the code generator rules called by the macro. See the RiA CG User Guide for details about the list of macros.

This property is used for executable components, while the `MakeFileContentForLib` property is used for library components.

*(Default = \$AdaCGGnatMakefile)*

## **MakeFileNameForLib**

This property is used to specify the filename for the makefile generated by Rhapsody from the corresponding `MakeFileContentForLib` property.

Note that this property specifies the entire filename, including the extension.

*(Default = \$ComponentName\$MakeExtension)*

## **MakeFileContentForLib**

This property defines the content of a file whose name is defined in the corresponding `MakeFileNameForLib` property.

The code generator will create the file and take the file content from the value of this property.

The value of the property can contain text and macros. Macro replacement depends on the code generator rules called by the macro. See the RiA CG User Guide for details about the list of macros.

This property is used for library components, while the `MakeFileContentForExe` property is used for executable components.

*(Default = \$AdaCGGnatMakefile)*

## **OSFileSystemCaseSensitive**

The `OSFileSystemCaseSensitive` property specifies whether the OS file system for a given environment is case sensitive.

*(Default = False)*

## **ObjCleanCommand**

The `ObjCleanCommand` property specifies the environment-specific command used to clean the object



files generated by a previous build.

*(Default = Empty string)*

## ObjExtension

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*(Default = .o)*

## ObjectName

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*(Default = Empty string)*

## ObjectsDirectory

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*(Default = Empty string)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example:

(lineNumber), (fileName), and (Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*(Default = ([^:]+):([0-9]+):([0-9]+):(.\*))*

## ParseSeverityError

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*(Default = ([^:]+):([0-9]+):([0-9]+):(.\*))*

## ParseSeverityWarning

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*(Default = ([^:]+):([0-9]+):([0-9]+):(.\*))*

## ParseErrorDescript

*(Default = ([^:]+):([0-9]+):([0-9]+):(.\*))*

## UseNewBuildOutputWindow

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the CG::General::ShowLogViewAfterBuild property to True.

*(Default = True)*

## ParseErrorMoreInfo

*(Default = Empty string)*

## ParseMakeError

The ParseMakeError property is used to define a regular expression that represents the format of make process or linker error messages. This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*(Default = (make):(.\*)(Error))*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*(Default = True)*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely.

In such configurations, the host is the machine running Rational Rhapsody, whereas the target is the machine running the application.

To run remotely, the UseRemoteHost property must be set to Checked. If UseRemoteHost is Checked and RemoteHost is blank, the current host name is used for the remote host. You can use this as a workaround if you have problems running animated applications on Windows 95.

The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*(Default = Empty string)*

## **SpecExtension**

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*(Default = .ads)*

## **UseNonZeroStdInputHandle**

The UseNonZeroStdInputHandle property is a Boolean value that specifies whether to use a non-zero standard input handle.

*(Default = True)*

## **UseRemoteHost**

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*(Default = False)*

## **AdaptorSearchPath**

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated

makefile search path. This property reflects the change in Version 4.1 where the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files. To upgrade a custom adapter to the new scheme, you must do the following:

Create the relevant operating system configuration file.

Add the file directory to the search path in the framework makefiles.

Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*(Default = Empty string)*

## **BLDAdditionalDefines**

The BLDAdditionalDefines property specifies additional compiler preprocessor flags. The default values are as follows:

Environment Default Value

INTEGRITY Empty string

MultiWin32

IntegrityESTL ESTL

*(Default = Empty string)*

## **BLDAdditionalOptions**

The BLDAdditionalOptions property specifies additional compilation switches. The default values are as follows:

Environment Default Value

INTEGRITY :optimizestrategy=space

:driver\_opts=--diag\_suppress=14

:driver\_opts=--diag\_suppress=550

IntegrityESTL :optimizestrategy=space

:driver\_opts=--diag\_suppress=14

:driver\_opts=--diag\_suppress=550

:cx\_mode=extended\_embedded

:cx\_lib=eece

:stdcxxincdirs=\$(INTEGRITY\_ROOT)\eecxx

:stdcxxincdirs=\$(INTEGRITY\_ROOT)\ansi

MultiWin32 :cx\_template\_option=noimplicit

:add\_output\_ext=checked

:cx\_e\_option=msgnumbers

:cx\_option=exceptions

:check=bounds

:check=assignbound

:check=nilderef

:cx\_template=local

:cx\_remark=14

:cx\_remark=161

:cx\_remark=837

:cx\_remark=817

:cx\_remark=815

:cx\_remark=47

:cx\_remark=69

:cx\_remark=830

:cx\_remark=550

:prelink.args=-r

:prelink.args=-X7

*(Default = Empty string)*

## **BLDIncludeAdditionalBLD**

The BLDIncludeAdditionalBLD specifies additional build options.

*(Default = Empty string)*

## BLDMainExecutableOptions

The BLDMainExecutableOptions property specifies the options generated in the main build file of the executable component of the model.

:target\_os=integrity

:ada\_library=full

:integrity\_option=dynamic

:staticlink=true

*(Default = Empty string)*

## BLDMainLibraryOptions

The BLDMainLibraryOptions property specifies the options generated in the main build file of the library component of the model. The default values are as follows:

Environment Default Value

INTEGRITY :target\_os=integrity

IntegrityESTL

MultiWin32 Empty MultiLine

*(Default = Empty string)*

## BLDTarget

The BLDTarget property specifies the target BSP. For example, ":target=Win32". This property also affects the names of the framework libraries used in the link.

*(Default = Empty string)*

## BriefErrorMessages

The BriefErrorMessages property determines whether a /brief option is generated on SPARK Examiner calls.

*(Default = Empty string)*

## BSP

The BSP property specifies the board support package (BSP) for the system. If you need to change the value of the CPU, you can simply reset the value in this property instead of changing the value in the MakeFileContent property.

*(Default = Empty string)*

## BSP\_Libraries

The BSP\_Libraries property specifies the default BSP libraries to link to. The default value is as follows:

```
"%RAVENROOT%/bsp/raven/standard_model" "%RAVENROOT%/bsp/system/simulator"  
"%RAVENROOT%/lib/extensions"
```

*(Default = Empty string)*

## CodeTestSettings

The CodeTestSettings property specifies the compiler command settings. This property supports integration with Applied Microsystems Corporation CodeTest.

*(Default = Empty string)*

## COM

The COM property specifies whether the current component is a COM component. By default, this property is set to Checked for all COM components (stereotypes COM DLL, COM EXE, and COM TLB).

If you set this property in the generated makefile for the component, the linker option /SUBSYSTEM is set to :windows.

*(Default = Empty string)*

## ConvertHostToIP

The ConvertHostToIP property specifies whether to convert the host name to an IP number. This is necessary because pSOSystem does not include a name service.

*(Default = Empty string)*

## CPPAdditionalReservedWords

The CPPAdditionalReservedWords property is a string that specifies additional reserved keywords that

Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody. The property value is checked at run time when you name/rename an element, based on the active configuration environment setting. The default values are as follows:

#### Environment Default Reserved Words

MultiWin32 \_\_asm \_\_finally naked \_\_based \_\_inline \_\_single\_inheritance

\_\_cdecl \_\_int8 \_\_stdcall \_\_declspec \_\_int16 dllexport \_\_int32

\_\_try dllimport \_\_int64 \_\_virtual\_inheritance \_\_except \_\_leave \_\_fastcall \_\_multiple\_inheritance

MicrosoftWinCE.NET

NucleusPLUS-PPC

PsosPPC PsosX86 Empty string

*(Default = Empty string)*

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The default values are as follows:

#### Environment Default Compile Command

Borland @echo Compiling \$OMFileImpPath

\$(CREATE\_OBJ\_DIR)

\$(BCC32) -c @

\$OMFileCPPCompileSwitches

| -o\$OMFileObjPath \$OMFileImpPath

INTEGRITYL echo Compiling \$OMFileObjPath ...

\$(CPP) \$OMFileCPPCompileSwitches -o

"\$OMFileObjPath" "\$OMFileImpPath"



IntegrityEST

Linux @echo Compiling \$OMFileImpPath

\$(CREATE\_OBJ\_DIR)

@\$(CC) \$OMFileCPPCompileSwitches -o \$OMFileObjPath \$OMFileImpPath

Microsoft \$(CREATE\_OBJ\_DIR)

\$(CPP) \$OMFileCPPCompileSwitches /Fo"\$OMFileObjPath" "\$OMFileImpPath"

MicrosoftDLL

MSStandardLibrary

MicrosoftWinCE.NET \$(CPP) \$OMFileCPPCompileSwitches /Fo"\$OMFileObjPath" "\$OMFileImpPath"

MultiWin32 \$(COMPILEINFOLINE) \$@...

\$(CPP) \$OMFileCPPCompileSwitches -c "\$OMFileImpPath" -o "\$OMFileObjPath"

NucleusPLUS-PPC \$(CPP) \$OMFileCPPCompileSwitches -o \$OMFileObjPath \$OMFileImpPath

OsePPCDiab Empty string

OseSfk

PsosPPC @echo Compiling \$OMFileImpPath

@\$(CREATE\_OBJ\_DIR)

@\$(CXX) \$(CXXOPTS) \$OMFileCPPCompileSwitches

\$OMFileImpPath -o \$OMFileObjPath

PsosX86 @echo Compiling \$OMFileImpPath

@\$(CREATE\_OBJ\_DIR)

@\$(CXX) \$(CXXOPTS)

\$OMFileCPPCompileSwitches

\$OMFileImpPath -o \$OMFileObjPath

@\$(INI) \$(INIFLAG) \$OMFileObjPath

QNXNeutrinoCW @echo Compiling \$OMFileImpPath

\$(CREATE\_OBJ\_DIR)

@\$(CC) \$OMFileCPPCompileSwitches -o

\$OMFileObjPath \$OMFileImpPath

QNXNeutrinoGCC

Solaris2 @echo Compiling \$OMFileImpPath

\$(CREATE\_OBJ\_DIR)

@\$(CC) \$OMFileCPPCompileSwitches -o

\$OMFileObjPath \$OMFileImpPath

Solaris2GNU

VxWorks @echo Compiling \$OMFileImpPath

\$(CREATE\_OBJ\_DIR)

@\$(CXX) \$(C++FLAGS)

\$OMFileCPPCompileSwitches -o

\$OMFileObjPath \$OMFileImpPath

*(Default = Empty string)*

## CPPCompileSwitches

The CPPCompileSwitches property specifies the compiler switches. The default values are as follows:

Environment Default Compile Switches

Borland -I\$(BCROOT)\INCLUDE;.; "\$(OMROOT)\LangCpp";

"\$(OMROOT)\LangCpp\oxf"; "\$(OMROOT)\LangCpp\omCom";

-D\_RTLDLL;\_AFXDLL;WIN32;\_CONSOLE;\_MBCS;\_WINDOWS;

BORLAND;\_BOOLEAN \$(INST\_FLAGS) \$(INCLUDE\_PATH) \$(INST\_INCLUDES)  
\$OMCPPCompileCommandSet -c

Linux -I. -I\$(OMROOT) -I\$(OMROOT)/LangCpp -I\$(OMROOT)/LangCpp/oxf \$(INST\_FLAGS)  
\$(INCLUDE\_PATH) \$(INST\_INCLUDES) -DUSE\_IOSTREAM \$OMCPPCompileCommandSet -c

Microsoft /I . /I \$(OMROOT)\LangCpp /I \$(OMROOT)\LangCpp\oxf /nologo /W3 /GX  
\$OMCPPCompileCommandSet /D "\_AFXDLL" /D "WIN32" /D "\_CONSOLE" /D "\_MBCS" /D  
"\_WINDOWS" \$(INST\_FLAGS) \$(INCLUDE\_PATH) \$(INST\_INCLUDES) /c

MicrosoftDLL

MicrosoftWin CE /I . /I \$(OMROOT)\LangCpp /I \$(OMROOT)\LangCpp\oxf /nologo /W3 /GX- /D  
\_WIN32\_WCE=\$(CEVersion) /D "\$(CEConfigName)" \$(MACHINE\_CPP\_FLAGS) /D  
"\_OM\_NO\_IOSTREAM" /D UNDER\_CE=\$(CEVersion) /D "UNICODE" /D  
"\_OM\_UNICODE\_ONLY" \$OMCPPCompileCommandSet /D "\_AFXDLL" /D "WIN32"  
\$(INST\_FLAGS) \$(INCLUDE\_PATH) \$(INST\_INCLUDES) /c

MSStandardLibrary /I . /I \$(OMROOT)\LangCpp /I \$(OMROOT)\LangCpp\oxf /nologo /W3 /GX  
\$OMCPPCompileCommandSet /D "\_AFXDLL" /D "WIN32" /D "\_CONSOLE" /D "\_MBCS" /D  
"\_WINDOWS" /D "OM\_USE\_STL" \$(INST\_FLAGS) \$(INCLUDE\_PATH) \$(INST\_INCLUDES) /c

OsePPCDiab -I. -I\$(OMROOT)/LangCpp \$(INST\_FLAGS) \$(INCLUDE\_PATH) \$(INST\_INCLUDES)

OseSfk

PsosPPC \$OMCPPCompileCommandSet

PsosX86

QNXNeutrinoGCC -I. -I\$(OMROOT) -I\$(OMROOT)/LangCpp -I\$(OMROOT)/LangCpp/oxf  
\$(INST\_FLAGS) \$(INCLUDE\_PATH) \$(INST\_INCLUDES) -DUSE\_IOSTREAM  
\$OMCPPCompileCommandSet -c

Solaris2

Solaris2GNU

VxWorks -I\$(OMROOT) -I\$(OMROOT)/LangCpp -I\$(OMROOT)/LangCpp/oxf -DVxWorks  
\$(INST\_FLAGS) \$(INCLUDE\_PATH) \$OMCPPCompileCommandSet -c

*(Default = Empty string)*

## CPU

The CPU property is a string that specifies the CPU type. The default values are as follows:

Environment Dependency Rule

MicrosoftWinCE.NET x86

*(Default = Empty string)*

## DebugSwitches

The DebugSwitches property sets the debug level used in debug switches. The default values are as follows:

Environment Possible Values Default Value

INTEGRITY Default, Multi, None, Plain, and Stack Default

OBJECTADA -ga, -gc, -ga -gc -ga

RAVEN\_PPC -ga, -gc, -ga -gc -ga

SPARK Empty string

*(Default = Empty string)*

## DEFExtension

The DEFExtension property is a string that specifies the extension for DLL definition files. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*(Default = Empty string)*

## DllExtension

The DllExtension property is a string that specifies the extension for DLL files. In general, this is an environment property that can be contained

in any of the environment metaclasses supported by Rational Rhapsody.

*(Default = Empty string)*

## DuplicateLibsListInMakeFile

The DuplicateLibsListInMakeFile property is a Boolean value that specifies whether Rational Rhapsody should duplicate the libraries list in the generated makefile link command. This property supports linkers that are sensitive to library order in the link command.

*(Default = Empty string)*

## EntryPointDeclarationModifier

The EntryPointDeclarationModifier property specifies a modifier for the entry point declaration. This property allows generation of the main() function in the specified syntax. To modify the main() signature implemented in the OSE adapter, do the following:

Add the EntryPointDeclarationModifier property to your environment properties and set it to the main return value and name. For example:

```
"int main"
```

Set the EntryPoint property to the main arguments. For example:

```
"int a, long b, char**"
```

Generate the code.

You get the following main() declaration:

```
int main(int a, long b, char** c) {  
  
... }
```

*(Default = Empty string)*

## EnvironmentVarName

The EnvironmentVarName property specifies the name of the global variable that you must define in order to use the Embedded C++ compiler. It is used by the MultiMakefileGenerator. The value replaces the EnvironmentVarName value keyword inside the value for the BLDAdditionalOptions property.

*(Default = Empty string)*

## ESTLCompliance

The ESTLCompliance property is a Boolean value that determines whether you are using the Embedded C++ (ESTL) environment and conform to its requirements.

In instrumentation mode, the Rational Rhapsody code generator usually creates an OMAAnimated"UserClass" friend class for each user-defined class. This class inherits from AOMInstance, if its "User Class" does not inherit from another class in the model.

This inheritance is virtual and is needed for multiple inheritance support. Because ESTL does not support multiple inheritance (as far as virtual inheritance), the Rational Rhapsody Developer for C++ code generator does not create “virtual” inheritance if ESTLCompliance is set to Checked.

To support ESTL compliance, Rational Rhapsody includes a new check to recognize the following elements of ESTL-noncompliance:

Multiple inheritance, caused by the user model (several superclasses)

Multiple inheritance, caused by Rational Rhapsody (an active reactive class is generated with two base classes: OMReactive and OMThread)

Multiple inheritance, caused by a combination of the following factors:

An active class containing a superclass

A reactive class containing a superclass

Virtual inheritance, declared by the user in the features of the superclass

In these cases, Rational Rhapsody displays the following warning message for each problematic class:

"ESTL does not support multiple/virtual inheritance"

Note that this check runs only when the ESTLCompliance property is set to Checked.

*(Default = Empty string)*

## FrameworkLibPrefix

The FrameworkLibPrefix property specifies the prefix of the Rational Rhapsody framework library linked with your application. The default values are as follows:

Environment Default Reserved Words

INTEGRITY Integrity

IntegrityESTL IntegrityESTL

MultiWin32 MultiWin32

*(Default = Empty string)*

## GeneratedAllDependencyRule

The GeneratedAllDependencyRule property specifies whether to automatically generate the “all:” rule as part of the expansion of the \$OMContextMacros keyword in the makefile. If this is Cleared, you can define the makefile macros manually.

*(Default = Empty string)*

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated

makefile. Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries). The default values are as follows:

#### Environment Default Value

Borland "\$\$(OMROOT)\LangCpp\lib\bc5WebComponents.lib", "\$\$(OMROOT)\lib\bc5WebServices.lib",  
-lsocket

#### INTEGRITY

"\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)WebComponents\$(LIB\_POSTFIX)\$(LIB\_EXT),  
\$(OMROOT)\lib\\$(LIB\_PREFIX)WebServices\$(LIB\_POSTFIX)\$(LIB\_EXT)"

#### IntegrityESTL

Linux \$(OMROOT)/LangCpp/lib/linuxWebComponents\$(LIB\_EXT),  
\$(OMROOT)/lib/linuxWebServices\$(LIB\_EXT)

Microsoft \$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)

WebComponents \$(LIB\_POSTFIX)\$(LIB\_EXT),  
\$(OMROOT)\lib\\$(LIB\_PREFIX)WebServices\$(LIB\_POSTFIX)\$(LIB\_EXT),ws2\_32\$(LIB\_EXT)

MicrosoftWinCE.NET \$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)

WebComponents \$(LIB\_POSTFIX)\$(LIB\_EXT),  
\$(OMROOT)\lib\\$(LIB\_PREFIX)WebServices\$(LIB\_POSTFIX)\$(LIB\_EXT), winsock.lib

NucleusPLUS-PPC \$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)

WebComponents \$(LIB\_POSTFIX)\$(LIB\_EXT),  
\$(OMROOT)\lib\\$(LIB\_PREFIX)WebServices\$(LIB\_POSTFIX)\$(LIB\_EXT)

QNXNeutrinoCW \$(OMROOT)\LangCpp\lib\

QNXCWWebComponents \$(CPU)\$(CPU\_SUFFIX)\$(LIB\_EXT),  
\$(OMROOT)\lib\QNXCWWebServices\$(CPU)\$(CPU\_SUFFIX)\$(LIB\_EXT), -lsocket

QNXNeutrinoGCC \$(OMROOT)\LangCpp\lib\QNXWebComponents\$(LIB\_EXT), \$(OMROOT)\lib\

QNXWebServices \$(LIB\_EXT), -lsocket

#### Solaris2

\$(OMROOT)\LangCpp\lib\sol2WebComponents\$(LIB\_EXT),\$(OMROOT)\lib\sol2WebServices\$(LIB\_EXT),  
-lsocket -lnsl

Solaris2GNU \$(OMROOT)\LangCpp\lib\

sol2WebComponentsGNU \$(LIB\_EXT), \$(OMROOT)\lib\sol2WebServicesGNU\$(LIB\_EXT),-lsocket  
-lnsl

VxWorks \$(OMROOT)\LangCpp\lib\vxWebComponents\$ CPU)\$(LIB\_EXT), \$(OMROOT)\lib\

vxWebServices \$(CPU)\$(LIB\_EXT)

*(Default = Empty string)*

## HasIDEInterface

The HasIDEInterface property is a Boolean value that specifies whether IDE support is enabled.

If IDE support is enabled (Checked), the IDEInterfaceDLL property points to an IDE adapter that provides connection to the IDE. If the property is set to Cleared, IDE support is disabled and IDE services are not attempted.

In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The default value for QNXNeutrinoCW is Cleared; for the other environments, the default value is Checked.

*(Default = Empty string)*

## IDEInterfaceDLL

The IDEInterfaceDLL property is a string that points to the IDE adapter DLL. You should not have any reason to modify this property. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody. The default values are as follows:

Environment Default Value

QNXNeutrinoCW \$OMROOT/DLLs/CodeWarriorIDE.dll

INTEGRITY Empty string

IntegrityESTL

VxWorks \$OMROOT/DLLs/TornadoIDE.dll

*(Default = Empty string)*



## InvokeCodeGeneration

The InvokeCodeGeneration property specifies the command line used by Rational Rhapsody to run an external code generator. The generator should implement the IRPEXternalCodeGenerator connection point. To use an external code generator, you need the appropriate license; an external generator license is part of the Rational Rhapsody Developer for Ada package license. The default values are as follows:

Metaclass Default Value

GNAT "\$OMROOT/etc/Executer.exe" "\$OMROOT/etc/invokeScriptor.bat"

MultiWin32

OBJECTADA

RAVEN\_PPC

SPARK

INTEGRITY "\$OMROOT/etc/Executer.exe" "\$OMROOT\etc\IntegrityAdaMake.bat \$makefile \$maketarget"

*(Default = Empty string)*

## InvokeMakeGenerator

The InvokeMakeGenerator specifies the path to the executable program for an external makefile generator. This external generator is started

each time you request a makefile generation. If the specified path is incorrect, Rational Rhapsody generates an error message. If you are using a full-featured external code generator, this property setting is ignored. The default values are as follows:

Environment Default Value

INTEGRITY \$OMROOT/etc/MultiMakefileGenerator.exe

MultiWin32

IntegrityESTL \$OMROOT/etc/IntegrityMakefileGenerator.bat

All others Empty string

*(Default = Empty string)*

## MainIncludes

The MainIncludes property is a string that specifies the files that need to be included in the main program generated for an application.

*(Default = Empty string)*

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*(Default = Empty string)*

## OMCPU

The OMCPU property is resolved in the MakeFileContent property as the CPU type. The QNXNeutrinoCW environment uses the custom keywords feature to enable you to select the CPU without modifying the makefile template.

*(Default = Empty string)*

## OMCPU\_SUFFIX

The OMCPU\_SUFFIX property is resolved in the MakeFileContent property as the CPU extension (which is required for PPC targets). The QNXNeutrinoCW environment uses the custom keywords feature to enable you to select the CPU without modifying the makefile template.

*(Default = Empty string)*

## OpenHTMLReports

The OpenHTMLReports property specifies whether to open the HTML reports when the examination is complete.

*(Default = Empty string)*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation. The default values are as follows:

Environment Default Value

Borland /

Linux

MultiWin32

PsosPPC PsosX86

QNXNeutrinoCW

QNXNeutrinoGCC

Solaris2

Solaris2GNU

JDK \

Microsoft

MicrosoftDLL

MSStandardLibrary

*(Default = Empty string)*

## **ProcessToKillAtStopExec**

The ProcessToKillAtStopExec property stops the running process of the Java application when you select Code > Stop Execution in the Rational Rhapsody GUI.

*(Default = Empty string)*

## **RCCompileCommand**

The RCCompileCommand property is a string that specifies the compilation command for the resource file.

In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The default values are as follows:

Environment Default Value

Microsoft \$(RC) /Fo"\$(TARGET\_MAIN).res" \$(TARGET\_MAIN)\$OMRCEExtension

MicrosoftDLL

MultiWin32 Empty string

*(Default = Empty string)*

## **RCEExtension**

The RCEExtension property is a string that specifies the extension for resource files. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*(Default = Empty string)*

## **ReusableStatechartSwitches**

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts. See the upgrade history on the support site for information on this change. The default values are as follows:

Environment Default Value

Borland -DOM\_REUSABLE\_STATECHART\_IMPLEMENTATION

Linux

NucleusPLUS-PPC

OsePPCDiab

OseSfk

PsosPPC

QNXNeutrinoCW

QNXNeutrinoGCC

Solaris2

Solaris2GNU

VxWorks

Microsoft /D "OM\_REUSABLE\_STATECHART\_IMPLEMENTATION"

MicrosoftDLL

MicrosoftWinCE.NET

MSStandardLibrary

INTEGRITY OM\_REUSABLE\_STATECHART\_IMPLEMENTATION

IntegrityESTL

MultiWin32

*(Default = Empty string)*

## **RPFrameWorkDll**

The RPFrameWorkDll property determines whether the configuration uses the DLL flavor of the framework libraries. To use OXF DLLs for the creation of COM ATL components, you must set this property to Checked before you generate code.

Rational Rhapsody COM ATL components use a DLL version of the OXF. This version of the OXF allows the use of multiple Rational Rhapsody-generated DLL/executable components confined to a single process.

There are three versions of the OXF DLL:

DLL Version Animation Enabled Trace Enabled

oxfdll.dll No No

oxfanimdll.dll Yes No

oxftracedll.dll No Yes

OXF DLLs are not a part of a typical Rational Rhapsody installation, and must be built from the OXF C++ sources. To obtain these sources, you can either perform a custom installation, or update the installation to add the sources to your existing installation. To rebuild the framework DLLs, run the following in your \$OMROOT\LangCpp folder:

```
nmake -f msoxfanimtracedll.mak CFG=oxfdll
```

```
nmake -f msoxfanimtracedll.mak CFG=oxfanimdll
```

```
nmake -f msoxfanimtracedll.mak CFG=oxftracedll
```

To use OXF DLLS for the creation of COM ATL components, set the following component configuration properties to True before you generate code:

ADA\_CG::Microsoft::RPFrameWorkDll

ADA\_CG::MicrosoftDLL::RPFrameWorkDll this is True by default

In addition, make sure that the following path are included in the system environment path:

OXF DLL path (\$OMROOT\LangCpp\lib)

The full path to regsrv32.exe

Without these settings, COM ATL components are not registered and cannot run. Limitations:

Rational Rhapsody components with different instrumentation settings (both Animation and Tracing) are not supported within a single process. However, you can mix instrumented and noninstrumented DLLs, as long as you use the instrumented DLL.

Mixing Rational Rhapsody components that link to the DLL and the library version of OXF is not supported within a single process.

*(Default = Empty string)*

## SpecFilesInDependencyRules

The SpecFilesInDependencyRules property specifies whether to include specification files in makefile dependency rules. The OSE makefile does not support specification files in the Dependency line. Therefore, the default for OSE is False. When this property is False, no .h files are added to the Dependency line of the makefile. The default value for GNAT is True; for OSE, the default value is False.

*(Default = Empty string)*

## SubSystem

The SubSystem property (ADA\_CG::Microsoft/MicrosoftDLL/MultiWin32) is a string that defines the type of the program for the Microsoft

linker. The possible values are as follows:

CONSOLE - Used for a Win32 character-mode application

WINDOWS - Used for an application that does not require a console

NATIVE - Applies device drivers for Windows NT

POSIX - Creates an application that runs with the POSIX subsystem in Windows NT

*(Default = Empty string)*

## TargetConfigurationFileName

The TargetConfigurationFileName property specifies the name of the target configuration file to be passed as an argument to the SPARK Examiner.

*(Default = Empty string)*

## UnixLineTerminationStyle

The UnixLineTerminationStyle property specifies whether generated files use the UNIX end-of-line style. If this property is set to False, the end-of-line style depends on the host type (for example, DOS style on Windows machines, and UNIX style on Solaris machines).

*(Default = Empty string)*

## UnixPathNameForOMROOT

The UnixPathNameForOMROOT property specifies whether the makefile must include UNIX-style path names. The pRISM compilers do not tolerate DOS-style path conventions. If you do not set this property correctly, there might be many compilation problems.

The default value for the following environments is Checked:

Linux

PsosPPC PsosX86

QNXNeutrinoCW

QNXNeutrinoGCC

Solaris2

Solaris2GNU

The default value for the following environments is Cleared:

JDK

OsePPCDia

OseSfk

*(Default = Empty string)*

## UseActorsCode

The UseActorsCode property specifies whether code is generated for actors. The value of the property should be synchronized

with the configuration "Generate Code For Actors" check box (located in the configuration Initialization tab).

*(Default = Empty string)*

## GNATVxWorks

This metaclass contains the properties that manipulate the GNATVxWorks operating system environment.

The properties are as follows:

- AdditionalReservedWords
- BuildCommandSet
- buildFrameworkCommand
- CompileCommand
- CompileSwitches
- CPPCompileDebug
- CPPCompileRelease
- DependencyRule
- EntryPoint
- ErrorMessageTokensFormat
- ExeExtension
- FileDependencies
- ImpExtension
- Include
- InvokeAnimatedExecutable
- InvokeExecutable
- InvokeMake
- IsFileNameShort
- LibExtension
- LinkDebug
- LinkRelease
- LinkSwitches
- MakeExtension
- MakeFileContent
- MakeFileNameForExe1
- MakeFileContentForExe1
- MakeFileNameForExe2
- MakeFileContentForExe2
- MakeFileNameForLib



- MakeFileContentForLib
- OSFileSystemCaseSensitive
- ObjCleanCommand
- ObjExtension
- ObjectName
- ObjectsDirectory
- ParseErrorMessage
- QuoteOMROOT
- RemoteHost
- SpecExtension
- UseNonZeroStdInputHandle
- UseRemoteHost
- VxWorksVariant

## AdditionalReservedWords

The AdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not

allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody. The property value is checked at run time when you name/rename an element, based on the active configuration environment setting. Note that this property affects the algorithm only when the active configuration is of the selected environment.

*(Default = Empty string)*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration. To change this property, use the Configuration window in the browser.

Important: Do NOT change it by using the Properties window or by modifying the site.prp file!

This property also affects the names of the framework libraries used in the link. The possible values are as follows:

Debug - Generate the debug command that is set in the makefile.

DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

Release - Generate the release command that is set in the makefile.

ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*(Default = Debug)*

## **buildFrameworkCommand**

*(Default = \"\$OMROOT/etc/Executer.exe\"  
\"\\\\\"\$OMROOT\\..\\Sodius\\RiA\_CG\\bin\\recompile\_GNAT\_VxW.bat\\\\\"")*

## **CompileCommand**

The CompileCommand property is a string that specifies a different compile command.

*(Default = )*

## **CompileSwitches**

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property.

*(Default = )*

## **CPPCompileDebug**

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*(Default = Empty string)*

## **CPPCompileRelease**

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*(Default = Empty string)*

## **DependencyRule**

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

For example, the following dependency rule lists the file

dependencies for a Windows application with a GUI, including bitmaps,

icons, and resource files:

`$OMFileObjPath : $OMFileImpPath "*.bmp" "*.ico" "*.rc2"`

*(Default = Empty string)*

## EntryPoint

The EntryPoint property specifies the name of the main program for a given environment.

See also the definition of the EntryPointDeclarationModifier property for more information.

*(Default = Empty string)*

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected

format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens has three parameters, each with an integer value:

TotalNumberOfTokens - The number of tokens in the regular expression

FileTokenPosition - The position of the file name token in the expression

LineTokenPosition - The position of the line number token in the expression

*(Default = TotalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2)*

## ExeExtension

The ExeExtension property specifies the extension that is appended to compiled executable components for a given environment.

*(Default = .exe)*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included

in model elements. The file inclusions are generated in the makefile.

*(Default = Empty string)*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment.

*(Default = .adb)*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*(Default = Empty string)*

## InvokeAnimatedExecutable

The InvokeExecutable property specifies the command used to run an animated executable file.

*(Default = \$executable \$port)*

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*(Default = \"\$executable\")*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT/etc/vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

*(Default = \"\$OMROOT/etc/Executer.exe\" | \"\\\"\$OMROOT\\etc\\GnatVxMake.bat\\\" \$makefile \$maketarget\")*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format.

If this is Checked:

The file name is not truncated.

If the FileName property is not blank, its value overrides any automatic file name synthesis.

If the file name is longer than eight characters, the Checker reports this prior to code generation.

*(Default = False)*

## **LibExtension**

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*(Default = .a)*

## **LinkDebug**

The LinkDebug property specifies the special link switches used to link in debug mode.

*(Default = Empty string)*

## **LinkRelease**

The LinkRelease property specifies the special link switches used to link in release mode.

*(Default = Empty string)*

## **LinkSwitches**

The LinkSwitches property specifies the standard link switches used to link in any mode.

*(Default = Empty string)*

## **MakeExtension**

The MakeExtension property specifies the extension that Rational Rhapsody appends to makefiles.

*(Default = .bat)*

## **MakeFileContent**

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

Target type

Compilation flags

Commands definitions

Generated macros

Predefined macros

Generated dependencies

Linking instructions

The following sections describe the contents of the makefile in detail.

Target Type

The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration. For example, the default content of the target type section of a C++ makefile for the Microsoft environment is as follows:

```
##### Target type (Debug/Release) #####  
  
#####  
  
CPPCompileDebug=$OMCPPCompileDebug  
  
CPPCompileRelease=$OMCPPCompileRelease  
  
LinkDebug=$OMLinkDebug  
  
LinkRelease=$OMLinkRelease  
  
BuildSet=$OMBuildSet  
  
SUBSYSTEM=$OMSubSystem  
  
COM=$OMCOM  
  
RPFrameWorkDll=$OMRPFrameWorkDll  
  
ConfigurationCPPCompileSwitches=  
  
$OMReusableStatechartSwitches $OMConfigurationCPPCompileSwitches
```

```
!IF "$(RPFrameWorkDll)" == "True"
```

```
ConfigurationCPPCompileSwitches=
```

```
$(ConfigurationCPPCompileSwitches) /D "FRAMEWORK_DLL"
```

```
!ENDIF
```

```
!IF "$(COM)" == "True"
```

```
SUBSYSTEM=/SUBSYSTEM:windows
```

```
!ENDIF
```

## Compilation Flags

The compilation flags section of the makefile contains the default compilation flags stored in the CompileSwitches property. For example, the default content of the compilation flags section of a C++ makefile for the Microsoft environment is as follows:

```
##### Compilation flags #####
```

```
#####
```

```
INCLUDE_QUALIFIER=/I
```

```
LIB_PREFIX=MS
```

## Commands Definitions

The commands definition section

of the makefile specifies programs to execute from the makefile. For example, the default commands definition section of a C++ makefile for the Microsoft environment is as follows:

```
##### Commands definition #####
```

```
#####
```

```
RMDIR = rmdir
```

```
LIB_CMD=link.exe -lib
```

```
LINK_CMD=link.exe
```

```
LIB_FLAGS=$OMConfigurationLinkSwitches
```

```
LINK_FLAGS=$OMConfigurationLinkSwitches $(SUBSYSTEM) /
```

```
MACHINE:I386
```

Generated Macros

The generated macros section of the makefile contains a variable that expands to the Rational Rhapsody-generated macros in the makefile. For example:

```
##### Generated macros #####  
#####
```

```
$OMContextMacros
```

```
OBJ_DIR=$OMObjectsDir
```

```
!IF "$ (OBJ_DIR)" != ""
```

```
CREATE_OBJ_DIR=if not exist $(OBJ_DIR) mkdir $(OBJ_DIR)
```

```
CLEAN_OBJ_DIR= if exist $(OBJ_DIR) $(RMDIR) $(OBJ_DIR)
```

```
!ELSE
```

```
CREATE_OBJ_DIR=
```

```
CLEAN_OBJ_DIR=
```

```
!ENDIF
```

The \$OMContextMacros keyword expands several macros in the makefile. Each makefile macro has its own keyword. You can use these keywords separately to customize the makefile. The \$OMContextMacros variable modifies target-specific variables. Replace the \$OMContextMacros line in the MakeFileContent

property with the following:

```
FLAGSFILE=$OMFlagsFile
```

```
RULESFILE=$OMRulesFile
```

```
OMROOT=$OMROOT
```

```
CPP_EXT=$OMImplExt
```

```
H_EXT=$OMSpecExt
```



```

OBJ_EXT=$OMObjExt

EXE_EXT=$OMExeExt

LIB_EXT=$OMLibExt

INSTRUMENTATION=$OMInstrumentation

TIME_MODEL=$OMTimeModel

TARGET_TYPE=$OMTargetType TARGET_NAME=$OMTargetName

$OMAllDependencyRule

TARGET_MAIN=$OMTargetMain LIBS=$OMLibs

INCLUDE_PATH=$OMIncludePath

ADDITIONAL_OBJS=$OMAdditionalObjs

OBJS= $OMObjs

Predefined Macros

```

The predefined macros section of the makefile contains other macros than the Rational Rhapsody-generated macros specified in the generated macros section. For example, the default predefined macros section of a C++ makefile for the Microsoft environment is as follows:

```

##### Predefined macros #####

#####

$(OBJS) : $(INST_LIBS) $(OXF_LIBS)

LIB_POSTFIX=

!IF "$(BuildSet)"=="Release"

LIB_POSTFIX=R

!ENDIF

!IF "$(TARGET_TYPE)" == "Executable"

LinkDebug=$(LinkDebug) /DEBUG

LinkRelease=$(LinkRelease) /OPT:NOREF

```

```

!ELSEIF "$(TARGET_TYPE)" == "Library"

LinkDebug=$(LinkDebug) /DEBUGTYPE:CV

!ENDIF


!IF "$(INSTRUMENTATION)" == "Animation"

INST_FLAGS=/D "OMANIMATOR"

INST_INCLUDES=/I $(OMROOT)\LangCpp\om /I

$(OMROOT)\LangCpp\om

!IF "$(RPFrameWorkDll)" == "True"

INST_LIBS=

OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfanimdll$(

LIB_POSTFIX)

$(LIB_EXT)

!ELSE

INST_LIBS=

$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomanim$(LIB_POSTFIX)

(LIB_EXT)

OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfinst$(LIB

POSTFIX)$(LIB_EXT)

$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX)$(LIB_EXT)

!ENDIF

SOCK_LIB=wsock32.lib


!ELSEIF "$(INSTRUMENTATION)" == "Tracing"

INST_FLAGS=/D "OMTRACER"

INST_INCLUDES=/I $(OMROOT)\LangCpp\om /I

```

```

$(OMROOT)\LangCpp\tom

!IF "$ (RPFrameWorkDll)" == "True"

INST_LIBS=

OXF_LIBS=

$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfracedll$(LIB_POST
FIX)$(LIB_EXT)

!ELSE

INST_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)tomtrace$
(LIB_POSTFIX)
$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomtrace$(LIB_POSTFIX)
$(LIB_EXT)
OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfinst$(LIB_POSTFIX)
(LIB_EXT) $(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX)$(LIB_EXT)
!ENDIF

SOCK_LIB=wsock32.lib

!ELSEIF "$ (INSTRUMENTATION)" == "None"

INST_FLAGS=

INST_INCLUDES=

INST_LIBS=

!IF "$ (RPFrameWorkDll)" == "True"

OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfdll$

```

```
(LIB_POSTFIX)$(LIB_EXT)
```

```
!ELSE
```

```
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$
```

```
(LIB_POSTFIX)$(LIB_EXT)
```

```
!ENDIF
```

```
SOCK_LIB=
```

```
!ELSE
```

```
!ERROR An invalid Instrumentation $(INSTRUMENTATION) is specified. !ENDIF
```

## Generated Dependencies

The generated dependencies section of the makefile contains a variable that expands to Rational Rhapsody-generated dependencies and compilation instructions. For example, the generated dependencies section of a C++ makefile for the Microsoft environment is as follows:

```
##### Generated dependencies #####
```

```
#####
```

```
$OMContextDependencies
```

```
$OMFileObjPath : $OMMainImplementationFile $(OBJS)
```

```
$(CPP) $(ConfigurationCPPCompileSwitches) /Fo"$OMFileObjPath" $OMMainImplementationFile
```

## Linking Instructions

The linking instructions section

of the makefile contains the predefined linking instructions. For example, the default linking instructions section of a C++ makefile for the Microsoft environment is as follows:

```
##### Linking instructions #####
```

```
#####
```

```
$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS)
```

```
$OMFileObjPath $OMMakefileName $OMModelLibs
```

```
@echo Linking $(TARGET_NAME)$(EXE_EXT)
```

```
$(LINK_CMD) $OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \
```

```

$(LIBS) \
$(INST_LIBS) \
$(OXF_LIBS) \
$(SOCK_LIB) \
$(LINK_FLAGS) /out:$(TARGET_NAME)$(EXE_EXT)
$(TARGET_NAME)$(LIB_EXT) : $(OBJS) $(ADDITIONAL_OBJS)
$OMMakefileName
@echo Building library $@
$(LIB_CMD) $(LIB_FLAGS) /out:$(TARGET_NAME)$(LIB_EXT)
$(OBJS) $(ADDITIONAL_OBJS)
clean:
@echo Cleanup
$OMCleanOBJS
if exist $OMFileObjPath erase $OMFileObjPath
if exist *$(OBJ_EXT) erase *$(OBJ_EXT)
if exist $(TARGET_NAME).pdb erase $(TARGET_NAME).pdb
if exist $(TARGET_NAME)$(LIB_EXT) erase
$(TARGET_NAME)$(LIB_EXT)
if exist $(TARGET_NAME).ilk erase $(TARGET_NAME).ilk
if exist $(TARGET_NAME)$(EXE_EXT) erase
$(TARGET_NAME)$(EXE_EXT)
$(CLEAN_OBJ_DIR)
(Default = )

```

## MakeFileNameForExe1

This property is used to specify the filename for the makefile generated by Rhapsody from the

corresponding MakeFileContentForExe property.

Note that this property specifies the entire filename, including the extension.

*(Default = gnat.adc)*

## **MakeFileContentForExe1**

This property defines the content of a file whose name is defined in the corresponding MakeFileNameForExe property.

The code generator will create the file and take the file content from the value of this property.

The value of the property can contain text and macros. Macro replacement depends on the code generator rules called by the macro. See the RiA CG User Guide for details about the list of macros.

This property is used for executable components, while the MakeFileContentForLib property is used for library components.

*(Default = \$AdaCGGnatAdc)*

## **MakeFileNameForExe2**

This property is used to specify the filename for the makefile generated by Rhapsody from the corresponding MakeFileContentForExe property.

Note that this property specifies the entire filename, including the extension.

*(Default = \$ComponentName\$MakeExtension)*

## **MakeFileContentForExe2**

This property defines the content of a file whose name is defined in the corresponding MakeFileNameForExe property.

The code generator will create the file and take the file content from the value of this property.

The value of the property can contain text and macros. Macro replacement depends on the code generator rules called by the macro. See the RiA CG User Guide for details about the list of macros.

This property is used for executable components, while the MakeFileContentForLib property is used for library components.

*(Default = \$AdaCGGnatMakefile)*

## **MakeFileNameForLib**

This property is used to specify the filename for the makefile generated by Rhapsody from the corresponding MakeFileContentForLib property.

Note that this property specifies the entire filename, including the extension.

*(Default = \$ComponentName\$MakeExtension)*

## **MakeFileContentForLib**

This property defines the content of a file whose name is defined in the corresponding MakeFileNameForLib property.

The code generator will create the file and take the file content from the value of this property.

The value of the property can contain text and macros. Macro replacement depends on the code generator rules called by the macro. See the RiA CG User Guide for details about the list of macros.

This property is used for library components, while the MakeFileContentForExe property is used for executable components.

*(Default = \$AdaCGGnatMakefile)*

## **OSFileSystemCaseSensitive**

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*(Default = False)*

## **ObjCleanCommand**

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*(Default = Empty string)*

## **ObjExtension**

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*(Default = .o)*

## **ObjectName**

The ObjectName property specifies an alternative name for the compiled object file in the generated

makefile.

*(Default = Empty string)*

## ObjectsDirectory

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*(Default = Empty string)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example:

(lineNumber), (fileName), and (Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*(Default = ([^:]+):?([0-9]+):?)*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*(Default = True)*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely.

In such configurations, the host is the machine running Rational Rhapsody, whereas the target is the machine running the application.

To run remotely, the UseRemoteHost property must be set to Checked. If UseRemoteHost is Checked and RemoteHost is blank, the current host name is used for the remote host. You can use this as a workaround if you have problems running animated applications on Windows 95.

The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*(Default = Empty string)*



## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*(Default = .ads)*

## UseNonZeroStdInputHandle

The UseNonZeroStdInputHandle property is a Boolean value that specifies whether to use a non-zero standard input handle.

*(Default = True)*

## UseRemoteHost

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*(Default = False)*

## VxWorksVariant

This property defines the variant of the VxWorks target.

*(Default = powerpc-wrs-vxworksae)*

# INTEGRITY

This metaclass contains the properties that manipulate the INTERGRITY operating system environment.

The properties are as follows:

- BLDDAdditionalOptions
- BLDDMainExecutableOptions
- BLDDMainLibraryOptions
- BLDDTarget
- BuildCommandSet
- buildFrameworkCommand
- CompileSwitches
- DebugSwitches

- ErrorMessageTokensFormat
- ExeExtension
- ImpExtension
- InvokeExecutable
- InvokeMake
- IsFileNameShort
- LibExtension
- LinkSwitches
- MakeExtension
- FilenameEntrypointBuildFileContent
- EntrypointBuildFileContent
- MakeFileNameForExe1
- MakeFileContentForExe1
- MakeFileNameForExe2
- MakeFileContentForExe2
- MakeFileNameForExe3
- MakeFileContentForExe3
- MakeFileNameForLib1
- MakeFileContentForLib1
- MakeFileNameForLib2
- MakeFileContentForLib2
- OSFileSystemCaseSensitive
- ParseErrorMessage
- QuoteOMROOT
- RemoteHost
- SpecExtension
- UseNonZeroStdInputHandle
- UseRemoteHost
- 

## BLDAdditionalOptions

The BLDAdditionalOptions property specifies additional compilation switches.

```
:optimizestrategy=space
```

```
:driver_opts=--diag_suppress=14
```

```
:driver_opts=--diag_suppress=550
```

*(Default = :optimizestrategy=space :driver\_opts=--diag\_suppress=14 :driver\_opts=--diag\_suppress=550)*

## BLDMainExecutableOptions

The BLDMainExecutableOptions property specifies the options generated in the main build file of the executable component of the model.

:target\_os=integrity

:ada\_library=full

:integrity\_option=dynamic

:staticlink=true

*(Default = :target\_os=integrity :ada\_library=full :integrity\_option=dynamic :staticlink=true)*

## BLDMainLibraryOptions

The BLDMainLibraryOptions property specifies the options generated in the main build file of the library component of the model.

*(Default = :target\_os=integrity)*

## BLDTarget

The BLDTarget property specifies the target BSP. For example, ":target=Win32". This property also affects the names of the framework libraries used in the link.

*(Default = sim800)*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration. To change this property, use the Configuration window in the browser - do not change it by using the Properties window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link. The possible values are as follows:

Debug - Generate the debug command that is set in the makefile.

DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

Release - Generate the release command that is set in the makefile.

ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*(Default = Debug)*

## buildFrameworkCommand

*(Default = \"\$OMROOT/etc/Executer.exe\" |\"|\$OMROOT\\etc\\IntegrityMake.bat|\" IntegrityBuild.bat  
buildLibs \$BLDTarget bld ada\")*

## CompileSwitches

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property.

*(Default = )*

## DebugSwitches

The DebugSwitches property sets the debug level used in debug switches.

*(Default = Default)*

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected

format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens has three parameters, each with an integer value:

TotalNumberOfTokens - The number of tokens in the regular expression

FileTokenPosition - The position of the file name token in the expression

LineTokenPosition - The position of the line number token in the expression

*(Default = TotalNumberOfTokens=3,FileTokenPosition=1,LineTokenPosition=3)*

## ExeExtension

The ExeExtension property specifies the extension that is appended to compiled executable components for a given environment.

*(Default = .elf)*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment.

*(Default = .adb)*

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*(Default = \"\$executable\")*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

*(Default = \"\$OMROOT/etc/Executer.exe\" |\"|\$\"\$OMROOT\etc\IntegrityMake.bat\|\" \$makefile \$maketarget\")*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format.

If this is Checked:

The file name is not truncated.

If the FileName property is not blank, its value overrides any automatic file name synthesis.

If the file name is longer than eight characters, the Checker reports this prior to code generation.

*(Default = False)*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*(Default = .a)*

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

*(Default = )*

## MakeExtension

The MakeExtension property specifies the extension that Rational Rhapsody appends to makefiles.

*(Default = .bld)*

## FilenameEntrypointBuildFileContent

The FilenameEntrypointBuildFileContent property defines the rule which will be called to create the names of the entry point files.

*(Default = \$AdaCGFilenameMULTIEntrypointBuildFile)*

## EntrypointBuildFileContent

The EntrypointBuildFileContent property defines the rules which will be called to generate entry points files.

The names of the files are determined by the rule defined in the FilenameEntrypointBuildContent property.

*(Default = \$AdaCGMULTIEntrypointBuildFile)*

## MakeFileNameForExe1

This property is used to specify the filename for the makefile generated by Rhapsody from the corresponding MakeFileContentForExe property.

Note that this property specifies the entire filename, including the extension.

*(Default = Main\$ComponentName\$MakeExtension)*

## MakeFileContentForExe1

This property defines the content of a file whose name is defined in the corresponding MakeFileNameForExe property.

The code generator will create the file and take the file content from the value of this property.

The value of the property can contain text and macros. Macro replacement depends on the code generator rules called by the macro. See the RiA CG User Guide for details about the list of macros.

This property is used for executable components, while the MakeFileContentForLib property is used for library components.

*(Default = \$AdaCGMultiEntryPoint)*

## **MakeFileNameForExe2**

This property is used to specify the filename for the makefile generated by Rhapsody from the corresponding MakeFileContentForExe property.

Note that this property specifies the entire filename, including the extension.

*(Default = sources\$MakeExtension)*

## **MakeFileContentForExe2**

This property defines the content of a file whose name is defined in the corresponding MakeFileNameForExe property.

The code generator will create the file and take the file content from the value of this property.

The value of the property can contain text and macros. Macro replacement depends on the code generator rules called by the macro. See the RiA CG User Guide for details about the list of macros.

This property is used for executable components, while the MakeFileContentForLib property is used for library components.

*(Default = \$AdaCGMultiSources)*

## **MakeFileNameForExe3**

This property is used to specify the filename for the makefile generated by Rhapsody from the corresponding MakeFileContentForExe property.

Note that this property specifies the entire filename, including the extension.

*(Default = \$ComponentName\$MakeExtension)*

## **MakeFileContentForExe3**

This property defines the content of a file whose name is defined in the corresponding `MakeFileNameForExe` property.

The code generator will create the file and take the file content from the value of this property.

The value of the property can contain text and macros. Macro replacement depends on the code generator rules called by the macro. See the RiA CG User Guide for details about the list of macros.

This property is used for executable components, while the `MakeFileContentForLib` property is used for library components.

*(Default = \$AdaCGMultiMakeFile)*

## **MakeFileNameForLib1**

This property is used to specify the filename for the makefile generated by Rhapsody from the corresponding `MakeFileContentForLib` property.

Note that this property specifies the entire filename, including the extension.

*(Default = sources\$MakeExtension)*

## **MakeFileContentForLib1**

This property defines the content of a file whose name is defined in the corresponding `MakeFileNameForLib` property.

The code generator will create the file and take the file content from the value of this property.

The value of the property can contain text and macros. Macro replacement depends on the code generator rules called by the macro. See the RiA CG User Guide for details about the list of macros.

This property is used for library components, while the `MakeFileContentForExe` property is used for executable components.

*(Default = \$AdaCGMultiSources)*

## **MakeFileNameForLib2**

This property is used to specify the filename for the makefile generated by Rhapsody from the corresponding `MakeFileContentForLib` property.

Note that this property specifies the entire filename, including the extension.

*(Default = \$ComponentName\$MakeExtension)*

## **MakeFileContentForLib2**



This property defines the content of a file whose name is defined in the corresponding MakeFileNameForLib property.

The code generator will create the file and take the file content from the value of this property.

The value of the property can contain text and macros. Macro replacement depends on the code generator rules called by the macro. See the RiA CG User Guide for details about the list of macros.

This property is used for library components, while the MakeFileContentForExe property is used for executable components.

*(Default = \$AdaCGMultiMakeFile)*

## **OSFileSystemCaseSensitive**

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*(Default = False)*

## **ParseErrorMessage**

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example:

(lineNumber), (fileName), and (Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*(Default = ([a-zA-Z][^:,]+)(,/: Error:/: Warning:) line ([0-9]+))*

## **QuoteOMROOT**

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*(Default = True)*

## **RemoteHost**

The RemoteHost property specifies the name of the host machine when you run an application remotely.

In such configurations, the host is the machine running Rational Rhapsody, whereas the target is the machine running the application.

To run remotely, the UseRemoteHost property must be set to Checked. If UseRemoteHost is Checked and

RemoteHost is blank, the current host name is used for the remote host. You can use this as a workaround if you have problems running animated applications on Windows 95.

The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*(Default = Empty string)*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*(Default = .ads)*

## UseNonZeroStdInputHandle

The UseNonZeroStdInputHandle property is a Boolean value that specifies whether to use a non-zero standard input handle.

*(Default = False)*

## UseRemoteHost

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*(Default = False)*

*(Default = Empty string)*

# INTEGRITY5

This metaclass contains the properties that manipulate the INTERGRITY5 operating system environment.

The properties are as follows:

- BLDAdditionalOptions
- BLDMainExecutableOptions
- BLDMainLibraryOptions
- BLDTarget
- BuildCommandSet

- buildFrameworkCommand
- CompileSwitches
- DebugSwitches
- ErrorMessageTokensFormat
- ExeExtension
- HasIDEInterface
- IDEInterfaceDLL
- ImpExtension
- IntegrityRoot
- InvokeExecutable
- InvokeMake
- IsFileNameShort
- LibExtension
- LinkSwitches
- MakeExtension
- FilenameEntrypointBuildFileContent
- EntrypointBuildFileContent
- MakeFileNameForExe1
- MakeFileContentForExe1
- MakeFileNameForExe2
- MakeFileContentForExe2
- MakeFileNameForExe3
- MakeFileContentForExe3
- MakeFileNameForLib1
- MakeFileContentForLib1
- MakeFileNameForLib2
- MakeFileContentForLib2
- OSFileSystemCaseSensitive
- ParseErrorMessage
- QuoteOMROOT
- RemoteHost
- SpecExtension
- UseNonZeroStdInputHandle
- UseRemoteHost

## **BLDAdditionalOptions**

The BLDAdditionalOptions property specifies additional compilation switches.

-Ospace

--diag\_suppress 14,550

*(Default = -Ospace --diag\_suppress 14,550)*

## BLDMainExecutableOptions

The BLDMainExecutableOptions property specifies the options generated in the main build file of the executable component of the model.

--ada\_library\_full

-dynamic

-non\_shared

*(Default = --ada\_library\_full -dynamic -non\_shared)*

## BLDMainLibraryOptions

The BLDMainLibraryOptions property specifies the options generated in the main build file of the library component of the model.

*(Default = -non\_shared)*

## BLDTarget

The BLDTarget property specifies the target BSP. For example, ":target=Win32". This property also affects the names of the framework libraries used in the link.

*(Default = sim800)*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration. To change this property, use the Configuration window in the browser - do not change it by using the Properties window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link. The possible values are as follows:

Debug - Generate the debug command that is set in the makefile.

DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

Release - Generate the release command that is set in the makefile.

ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*(Default = Debug)*

## buildFrameworkCommand

*(Default = \"\$OMROOT/etc/Executer.exe\" \"|\" \"\$OMROOT\\etc\\Integrity5Make.bat\\|\" IntegrityBuild.bat buildLibs \$BLDTarget ada\")*

## CompileSwitches

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property.

*(Default = )*

## DebugSwitches

The DebugSwitches property sets the debug level used in debug switches.

*(Default = --no\_debug)*

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected

format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens has three parameters, each with an integer value:

TotalNumberOfTokens - The number of tokens in the regular expression

FileTokenPosition - The position of the file name token in the expression

LineTokenPosition - The position of the line number token in the expression

*(Default = TotalNumberOfTokens=3,FileTokenPosition=1,LineTokenPosition=3)*

## ExeExtension

The ExeExtension property specifies the extension that is appended to compiled executable components for a given environment.

*(Default = .elf)*

## HasIDEInterface

The HasIDEInterface property is a Boolean value that specifies whether IDE support is enabled.

If IDE support is enabled (Checked), the IDEInterfaceDLL property points to an IDE adapter that provides connection to the IDE. If the property is set to Cleared, IDE support is disabled and IDE services are not attempted.

In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*(Default = False)*

## IDEInterfaceDLL

The IDEInterfaceDLL property is a string that points to the IDE adapter DLL. You should not have any reason to modify this property. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*(Default = Empty string)*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment.

*(Default = .adb)*

## IntegrityRoot

This property holds the value of the “os\_dir” parameter generated in build files. The default value of this property is set by the Rhapsody installer.

*(Default = <INTEGRITYROOT>)*

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*(Default = \"\$executable\")*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

```
(Default = "\"$OMROOT/etc/Executer.exe\" \"\\\"$OMROOT\\etc\\Integrity5Make.bat\\\" \"$makefile $maketarget\"")
```

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format.

If this is Checked:

The file name is not truncated.

If the FileName property is not blank, its value overrides any automatic file name synthesis.

If the file name is longer than eight characters, the Checker reports this prior to code generation.

*(Default = False)*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*(Default = .a)*

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

*(Default = )*

## MakeExtension

The MakeExtension property specifies the extension that Rational Rhapsody appends to makefiles.

*(Default = .gpj)*

## FilenameEntrypointBuildFileContent

The `FilenameEntrypointBuildFileContent` property defines the rule which will be called to create the names of the entry point files.

*(Default = `$AdaCGFilenameMULTIEntrypointBuildFile`)*

## EntrypointBuildFileContent

The `EntrypointBuildFileContent` property defines the rules which will be called to generate entry points files.

The names of the files are determined by the rule defined in the `FilenameEntrypointBuildContent` property.

*(Default = `$AdaCGMULTI4EntrypointBuildFile`)*

## MakeFileNameForExe1

This property is used to specify the filename for the makefile generated by Rhapsody from the corresponding `MakeFileContentForExe` property.

Note that this property specifies the entire filename, including the extension.

*(Default = `Main$ComponentName$MakeExtension`)*

## MakeFileContentForExe1

This property defines the content of a file whose name is defined in the corresponding `MakeFileNameForExe` property.

The code generator will create the file and take the file content from the value of this property.

The value of the property can contain text and macros. Macro replacement depends on the code generator rules called by the macro. See the RiA CG User Guide for details about the list of macros.

This property is used for executable components, while the `MakeFileContentForLib` property is used for library components.

*(Default = `$AdaCGMulti4EntryPoint`)*

## MakeFileNameForExe2

This property is used to specify the filename for the makefile generated by Rhapsody from the corresponding `MakeFileContentForExe` property.

Note that this property specifies the entire filename, including the extension.

*(Default = `sources$MakeExtension`)*



## MakeFileContentForExe2

This property defines the content of a file whose name is defined in the corresponding MakeFileNameForExe property.

The code generator will create the file and take the file content from the value of this property.

The value of the property can contain text and macros. Macro replacement depends on the code generator rules called by the macro. See the RiA CG User Guide for details about the list of macros.

This property is used for executable components, while the MakeFileContentForLib property is used for library components.

*(Default = \$AdaCGMulti4Sources)*

## MakeFileNameForExe3

This property is used to specify the filename for the makefile generated by Rhapsody from the corresponding MakeFileContentForExe property.

Note that this property specifies the entire filename, including the extension.

*(Default = \$ComponentName\$MakeExtension)*

## MakeFileContentForExe3

This property defines the content of a file whose name is defined in the corresponding MakeFileNameForExe property.

The code generator will create the file and take the file content from the value of this property.

The value of the property can contain text and macros. Macro replacement depends on the code generator rules called by the macro. See the RiA CG User Guide for details about the list of macros.

This property is used for executable components, while the MakeFileContentForLib property is used for library components.

*(Default = \$AdaCGMulti4MakeFile)*

## MakeFileNameForLib1

This property is used to specify the filename for the makefile generated by Rhapsody from the corresponding MakeFileContentForLib property.

Note that this property specifies the entire filename, including the extension.

*(Default = sources\$MakeExtension)*

## MakeFileContentForLib1

This property defines the content of a file whose name is defined in the corresponding MakeFileNameForLib property.

The code generator will create the file and take the file content from the value of this property.

The value of the property can contain text and macros. Macro replacement depends on the code generator rules called by the macro. See the RiA CG User Guide for details about the list of macros.

This property is used for library components, while the MakeFileContentForExe property is used for executable components.

*(Default = \$AdaCGMulti4Sources)*

## MakeFileNameForLib2

This property is used to specify the filename for the makefile generated by Rhapsody from the corresponding MakeFileContentForLib property.

Note that this property specifies the entire filename, including the extension.

*(Default = \$ComponentName\$MakeExtension)*

## MakeFileContentForLib2

This property defines the content of a file whose name is defined in the corresponding MakeFileNameForLib property.

The code generator will create the file and take the file content from the value of this property.

The value of the property can contain text and macros. Macro replacement depends on the code generator rules called by the macro. See the RiA CG User Guide for details about the list of macros.

This property is used for library components, while the MakeFileContentForExe property is used for executable components.

*(Default = \$AdaCGMulti4MakeFile)*

## OSFileSystemCaseSensitive

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*(Default = False)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example:

(lineNumber), (fileName), and (Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*(Default = ([a-zA-Z][^:,]+)(,/: Error:/: Warning:) line ([0-9]+))*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*(Default = True)*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely.

In such configurations, the host is the machine running Rational Rhapsody, whereas the target is the machine running the application.

To run remotely, the UseRemoteHost property must be set to Checked. If UseRemoteHost is Checked and RemoteHost is blank, the current host name is used for the remote host. You can use this as a workaround if you have problems running animated applications on Windows 95.

The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*(Default = Empty string)*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*(Default = .ads)*

## UseNonZeroStdInputHandle

The UseNonZeroStdInputHandle property is a Boolean value that specifies whether to use a non-zero standard input handle.

*(Default = False)*

## UseRemoteHost

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*(Default = False)*

## MultiWin32

This metaclass contains the properties that manipulate the MultiWin32 operating system environment.

The properties are as follows:

- BLDDAdditionalOptions
- BLDDMainExecutableOptions
- BLDDMainLibraryOptions
- BuildCommandSet
- buildFrameworkCommand
- CompileSwitches
- DebugSwitches
- ErrorMessageTokensFormat
- ExeExtension
- ImpExtension
- InvokeCodeGeneration
- InvokeExecutable
- InvokeMake
- IsFileNameShort
- LibExtension
- LinkSwitches
- MakeExtension
- FilenameEntrypointBuildFileContent
- EntrypointBuildFileContent
- MakeFileNameForExe1
- MakeFileContentForExe1
- MakeFileNameForExe2
- MakeFileContentForExe2
- MakeFileNameForExe3

- MakeFileContentForExe3
- MakeFileNameForLib1
- MakeFileContentForLib1
- MakeFileNameForLib2
- MakeFileContentForLib2
- OSFileSystemCaseSensitive
- ParseErrorMessage
- QuoteOMROOT
- RemoteHost
- SpecExtension
- UseNonZeroStdInputHandle
- UseRemoteHost

## BLDAdditionalOptions

The BLDAdditionalOptions property specifies additional compilation switches.

*(Default = )*

## BLDMainExecutableOptions

The BLDMainExecutableOptions property specifies the options generated in the main build file of the executable component of the model.

`:win32_threading=multiple`

`:ada_library=full`

*(Default = :win32\_threading=multiple :ada\_library=full)*

## BLDMainLibraryOptions

The BLDMainLibraryOptions property specifies the options generated in the main build file of the library component of the model.

*(Default = )*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration. To change this property, use the Configuration window in the browser - do not change it by using the Properties window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link. The possible values are as follows:

Debug - Generate the debug command that is set in the makefile.

DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

Release - Generate the release command that is set in the makefile.

ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*(Default = Debug)*

## buildFrameworkCommand

*(Default = \"\$OMROOT/etc/Executer.exe\"  
\"\\\"\$OMROOT\\..\\Sodius\\RiA\_CG\\bin\\Recompile\_AdaMultiWin32.bat\\\"\\\")*

## CompileSwitches

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property.

*(Default = )*

## DebugSwitches

The DebugSwitches property sets the debug level used in debug switches.

*(Default = Default)*

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected

format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens has three parameters, each with an integer value:

TotalNumberOfTokens - The number of tokens in the regular expression

FileTokenPosition - The position of the file name token in the expression

LineTokenPosition - The position of the line number token in the expression

*(Default = TotalNumberOfTokens=3,FileTokenPosition=1,LineTokenPosition=3)*

## ExeExtension

The ExeExtension property specifies the extension that is appended to compiled executable components for a given environment.

*(Default = .exe)*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment.

*(Default = .adb)*

## InvokeCodeGeneration

The InvokeCodeGeneration property specifies the command line used by Rational Rhapsody to run an external code generator. The generator should implement the IRPEXternalCodeGenerator connection point. To use an external code generator, you need the appropriate license; an external generator license is part of the Rational Rhapsody Developer for Ada package license.

*(Default = \" \$OMROOT/etc/Executer.exe\" \" \$OMROOT/etc/runScriptor.bat\" )*

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*(Default = \"\$executable\" )*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\" $OMROOT/etc/vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

```
"$OMROOT/etc/Executer.exe" "$OMROOT/etc\MultiWin32Make.bat $makefile $maketarget"
```

*(Default = \" \$OMROOT/etc/Executer.exe\" \" \" \"\$OMROOT/etc\MultiWin32Make.bat\" \" \$makefile*

*\$maketarget\')*

## **IsFileNameShort**

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format.

If this is Checked:

The file name is not truncated.

If the FileName property is not blank, its value overrides any automatic file name synthesis.

If the file name is longer than eight characters, the Checker reports this prior to code generation.

*(Default = False)*

## **LibExtension**

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*(Default = .lib)*

## **LinkSwitches**

The LinkSwitches property specifies the standard link switches used to link in any mode.

*(Default = )*

## **MakeExtension**

The MakeExtension property specifies the extension that Rational Rhapsody appends to makefiles.

*(Default = .bld)*

## **FilenameEntrypointBuildFileContent**

The FilenameEntrypointBuildFileContent property defines the rule which will be called to create the names of the entry point files.

*(Default = \$AdaCGFilenameMULTIEntrypointBuildFile)*

## **EntrypointBuildFileContent**



The EntrypointBuildFileContent property defines the rules which will be called to generate entry points files.

The names of the files are determined by the rule defined in the FilenameEntrypointBuildContent property.

*(Default = \$AdaCGMULTIEntrypointBuildFile)*

## **MakeFileNameForExe1**

This property is used to specify the filename for the makefile generated by Rhapsody from the corresponding MakeFileContentForExe property.

Note that this property specifies the entire filename, including the extension.

*(Default = Main\$ComponentName\$MakeExtension)*

## **MakeFileContentForExe1**

This property defines the content of a file whose name is defined in the corresponding MakeFileNameForExe property.

The code generator will create the file and take the file content from the value of this property.

The value of the property can contain text and macros. Macro replacement depends on the code generator rules called by the macro. See the RiA CG User Guide for details about the list of macros.

This property is used for executable components, while the MakeFileContentForLib property is used for library components.

*(Default = \$AdaCGMultiEntryPoint)*

## **MakeFileNameForExe2**

This property is used to specify the filename for the makefile generated by Rhapsody from the corresponding MakeFileContentForExe property.

Note that this property specifies the entire filename, including the extension.

*(Default = sources\$MakeExtension)*

## **MakeFileContentForExe2**

This property defines the content of a file whose name is defined in the corresponding MakeFileNameForExe property.

The code generator will create the file and take the file content from the value of this property.

The value of the property can contain text and macros. Macro replacement depends on the code generator rules called by the macro. See the RiA CG User Guide for details about the list of macros.

This property is used for executable components, while the `MakeFileContentForLib` property is used for library components.

*(Default = \$AdaCGMultiSources)*

## **MakeFileNameForExe3**

This property is used to specify the filename for the makefile generated by Rhapsody from the corresponding `MakeFileContentForExe` property.

Note that this property specifies the entire filename, including the extension.

*(Default = \$ComponentName\$MakeExtension)*

## **MakeFileContentForExe3**

This property defines the content of a file whose name is defined in the corresponding `MakeFileNameForExe` property.

The code generator will create the file and take the file content from the value of this property.

The value of the property can contain text and macros. Macro replacement depends on the code generator rules called by the macro. See the RiA CG User Guide for details about the list of macros.

This property is used for executable components, while the `MakeFileContentForLib` property is used for library components.

*(Default = \$AdaCGMultiMakeFile)*

## **MakeFileNameForLib1**

This property is used to specify the filename for the makefile generated by Rhapsody from the corresponding `MakeFileContentForLib` property.

Note that this property specifies the entire filename, including the extension.

*(Default = sources\$MakeExtension)*

## **MakeFileContentForLib1**

This property defines the content of a file whose name is defined in the corresponding `MakeFileNameForLib` property.

The code generator will create the file and take the file content from the value of this property.

The value of the property can contain text and macros. Macro replacement depends on the code generator rules called by the macro. See the RiA CG User Guide for details about the list of macros.

This property is used for library components, while the `MakeFileContentForExe` property is used for executable components.

*(Default = \$AdaCGMultiSources)*

## **MakeFileNameForLib2**

This property is used to specify the filename for the makefile generated by Rhapsody from the corresponding `MakeFileContentForLib` property.

Note that this property specifies the entire filename, including the extension.

*(Default = \$ComponentName\$MakeExtension)*

## **MakeFileContentForLib2**

This property defines the content of a file whose name is defined in the corresponding `MakeFileNameForLib` property.

The code generator will create the file and take the file content from the value of this property.

The value of the property can contain text and macros. Macro replacement depends on the code generator rules called by the macro. See the RiA CG User Guide for details about the list of macros.

This property is used for library components, while the `MakeFileContentForExe` property is used for executable components.

*(Default = \$AdaCGMultiMakeFile)*

## **OSFileSystemCaseSensitive**

The `OSFileSystemCaseSensitive` property specifies whether the OS file system for a given environment is case sensitive.

*(Default = False)*

## **ParseErrorMessage**

The `ParseErrorMessage` property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example:

(lineNumber), (fileName), and (Error|Fatal|Warning) Along with the `ErrorMessageTokensFormat` property, `ParseErrorMessage` specifies the expected format of error messages for a given environment.

These two properties retrieve the file name and line number of errors reported by the compiler.

*(Default = ([a-zA-Z][[:^:],]+)(,/: Error:/: Warning:) line ([0-9]+))*

## **QuoteOMROOT**

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*(Default = True)*

## **RemoteHost**

The RemoteHost property specifies the name of the host machine when you run an application remotely.

In such configurations, the host is the machine running Rational Rhapsody, whereas the target is the machine running the application.

To run remotely, the UseRemoteHost property must be set to Checked. If UseRemoteHost is Checked and RemoteHost is blank, the current host name is used for the remote host. You can use this as a workaround if you have problems running animated applications on Windows 95.

The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*(Default = Empty string)*

## **SpecExtension**

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*(Default = .ads)*

## **UseNonZeroStdInputHandle**

The UseNonZeroStdInputHandle property is a Boolean value that specifies whether to use a non-zero standard input handle.

*(Default = False)*

## **UseRemoteHost**

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

(Default = False)

## Multi4Win32

This metaclass contains the properties that manipulate the Multi4Win32 operating system environment.

The properties are as follows:

- BLDDAdditionalOptions
- BuildCommandSet
- buildFrameworkCommand
- BLDDMainExecutableOptions
- BLDDMainLibraryOptions
- CompileSwitches
- DebugSwitches
- ErrorMessageTokensFormat
- ExeExtension
- HasIDEInterface
- IDEInterfaceDLL
- ImpExtension
- InvokeExecutable
- InvokeMake
- IsFileNameShort
- LibExtension
- LinkSwitches
- MakeExtension
- FilenameEntrypointBuildFileContent
- EntrypointBuildFileContent
- MakeFileNameForExe1
- MakeFileContentForExe1
- MakeFileNameForExe2
- MakeFileContentForExe2
- MakeFileNameForExe3
- MakeFileContentForExe3
- MakeFileNameForLib1
- MakeFileContentForLib1
- MakeFileNameForLib2

- MakeFileContentForLib2
- OSFileSystemCaseSensitive
- ParseErrorMessage
- QuoteOMROOT
- RemoteHost
- SpecExtension
- UseNonZeroStdInputHandle
- UseRemoteHost

## BLDAdditionalOptions

The `BLDAdditionalOptions` property specifies additional compilation switches.

(Default = )

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration. To change this property, use the Configuration window in the browser - do not change it by using the Properties window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link. The possible values are as follows:

Debug - Generate the debug command that is set in the makefile.

DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

Release - Generate the release command that is set in the makefile.

**ReleaseNoExp** - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

(Default = Debug)

## buildFrameworkCommand

```
(Default = \"$OMROOT/etc/Executer.exe\"  
|\"||\"$OMROOT\\..\\Sodius\\RiA_CG\\bin\\Recompile_AdaMulti4Win32.bar|\"|\"")
```

## BLDMainExecutableOptions

The `BLDMainExecutableOptions` property specifies the options generated in the main build file of the executable component of the model.

`-threading=multiple`

`--ada_library_full`

*(Default = -threading=multiple --ada\_library\_full)*

## **BLDMainLibraryOptions**

The BLDMainLibraryOptions property specifies the options generated in the main build file of the library component of the model.

*(Default = )*

## **CompileSwitches**

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property.

*(Default = )*

## **DebugSwitches**

The DebugSwitches property sets the debug level used in debug switches.

*(Default = --no\_debug)*

## **ErrorMessageTokensFormat**

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected

format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens has three parameters, each with an integer value:

TotalNumberOfTokens - The number of tokens in the regular expression

FileTokenPosition - The position of the file name token in the expression

LineTokenPosition - The position of the line number token in the expression

*(Default = TotalNumberOfTokens=3,FileTokenPosition=1,LineTokenPosition=3)*

## **ExeExtension**

The ExeExtension property specifies the extension that is appended to compiled executable components for a given environment.

*(Default = .exe)*

## HasIDEInterface

The HasIDEInterface property is a Boolean value that specifies whether IDE support is enabled.

If IDE support is enabled (Checked), the IDEInterfaceDLL property points to an IDE adapter that provides connection to the IDE. If the property is set to Cleared, IDE support is disabled and IDE services are not attempted.

In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*(Default = False)*

## IDEInterfaceDLL

The IDEInterfaceDLL property is a string that points to the IDE adapter DLL. You should not have any reason to modify this property. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*(Default = Empty string)*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment.

*(Default = .adb)*

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*(Default = \"\$executable\")*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the



BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

```
(Default = \"$OMROOT/etc/Executer.exe\" \"\\\"$OMROOT\etc\\Multi4Win32Make.bat\\\" $makefile $maketarget\")
```

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format.

If this is Checked:

The file name is not truncated.

If the FileName property is not blank, its value overrides any automatic file name synthesis.

If the file name is longer than eight characters, the Checker reports this prior to code generation.

*(Default = False)*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*(Default = .lib)*

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

*(Default = )*

## MakeExtension

The MakeExtension property specifies the extension that Rational Rhapsody appends to makefiles.

*(Default = .gpj)*

## FilenameEntrypointBuildFileContent

The FilenameEntrypointBuildFileContent property defines the rule which will be called to create the names of the entry point files.

*(Default = \$AdaCGFilenameMULTIEntrypointBuildFile)*

## EntrypointBuildFileContent

The EntrypointBuildFileContent property defines the rules which will be called to generate entry points files.

The names of the files are determined by the rule defined in the FilenameEntrypointBuildContent property.

*(Default = \$AdaCGMULTI4EntrypointBuildFile)*

## MakeFileNameForExe1

This property is used to specify the filename for the makefile generated by Rhapsody from the corresponding MakeFileContentForExe property.

Note that this property specifies the entire filename, including the extension.

*(Default = Main\$ComponentName\$MakeExtension)*

## MakeFileContentForExe1

This property defines the content of a file whose name is defined in the corresponding MakeFileNameForExe property.

The code generator will create the file and take the file content from the value of this property.

The value of the property can contain text and macros. Macro replacement depends on the code generator rules called by the macro. See the RiA CG User Guide for details about the list of macros.

This property is used for executable components, while the MakeFileContentForLib property is used for library components.

*(Default = \$AdaCGMulti4EntryPoint)*

## MakeFileNameForExe2

This property is used to specify the filename for the makefile generated by Rhapsody from the corresponding MakeFileContentForExe property.

Note that this property specifies the entire filename, including the extension.

*(Default = sources\$MakeExtension)*

## MakeFileContentForExe2

This property defines the content of a file whose name is defined in the corresponding `MakeFileNameForExe` property.

The code generator will create the file and take the file content from the value of this property.

The value of the property can contain text and macros. Macro replacement depends on the code generator rules called by the macro. See the RiA CG User Guide for details about the list of macros.

This property is used for executable components, while the `MakeFileContentForLib` property is used for library components.

*(Default = \$AdaCGMulti4Sources)*

## **MakeFileNameForExe3**

This property is used to specify the filename for the makefile generated by Rhapsody from the corresponding `MakeFileContentForExe` property.

Note that this property specifies the entire filename, including the extension.

*(Default = \$ComponentName\$MakeExtension)*

## **MakeFileContentForExe3**

This property defines the content of a file whose name is defined in the corresponding `MakeFileNameForExe` property.

The code generator will create the file and take the file content from the value of this property.

The value of the property can contain text and macros. Macro replacement depends on the code generator rules called by the macro. See the RiA CG User Guide for details about the list of macros.

This property is used for executable components, while the `MakeFileContentForLib` property is used for library components.

*(Default = \$AdaCGMulti4MakeFile)*

## **MakeFileNameForLib1**

This property is used to specify the filename for the makefile generated by Rhapsody from the corresponding `MakeFileContentForLib` property.

Note that this property specifies the entire filename, including the extension.

*(Default = sources\$MakeExtension)*

## **MakeFileContentForLib1**

This property defines the content of a file whose name is defined in the corresponding `MakeFileNameForLib` property.

The code generator will create the file and take the file content from the value of this property.

The value of the property can contain text and macros. Macro replacement depends on the code generator rules called by the macro. See the RiA CG User Guide for details about the list of macros.

This property is used for library components, while the `MakeFileContentForExe` property is used for executable components.

*(Default = \$AdaCGMulti4Sources)*

## **MakeFileNameForLib2**

This property is used to specify the filename for the makefile generated by Rhapsody from the corresponding `MakeFileContentForLib` property.

Note that this property specifies the entire filename, including the extension.

*(Default = \$ComponentName\$MakeExtension)*

## **MakeFileContentForLib2**

This property defines the content of a file whose name is defined in the corresponding `MakeFileNameForLib` property.

The code generator will create the file and take the file content from the value of this property.

The value of the property can contain text and macros. Macro replacement depends on the code generator rules called by the macro. See the RiA CG User Guide for details about the list of macros.

This property is used for library components, while the `MakeFileContentForExe` property is used for executable components.

*(Default = \$AdaCGMulti4MakeFile)*

## **OSFileSystemCaseSensitive**

The `OSFileSystemCaseSensitive` property specifies whether the OS file system for a given environment is case sensitive.

*(Default = False)*

## **ParseErrorMessage**

The `ParseErrorMessage` property defines a regular expression that matches the format of error messages.

The regular expression can contain two or more tokens. For example:

(lineNumber), (fileName), and (Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*(Default = ([a-zA-Z][[:^,]]+)(,/: Error:/: Warning:) line ([0-9]+))*

## **QuoteOMROOT**

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*(Default = True)*

## **RemoteHost**

The RemoteHost property specifies the name of the host machine when you run an application remotely.

In such configurations, the host is the machine running Rational Rhapsody, whereas the target is the machine running the application.

To run remotely, the UseRemoteHost property must be set to Checked. If UseRemoteHost is Checked and RemoteHost is blank, the current host name is used for the remote host. You can use this as a workaround if you have problems running animated applications on Windows 95.

The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*(Default = Empty string)*

## **SpecExtension**

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*(Default = .ads)*

## **UseNonZeroStdInputHandle**

The UseNonZeroStdInputHandle property is a Boolean value that specifies whether to use a non-zero standard input handle.

*(Default = False)*

## UseRemoteHost

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

(Default = False)

## OBJECTADA

This metaclass contains the properties that manipulate the OBJECTADA operating system environment.

The properties are as follows:

- BuildCommandSet
- buildFrameworkCommand
- CompileSwitches
- DebugSwitches
- ErrorMessageTokensFormat
- ExeExtension
- ImpExtension
- InvokeAnimatedExecutable
- InvokeExecutable
- InvokeMake
- IsFileNameShort
- LibExtension
- LinkSwitches
- MakeExtension
- MakeFileNameForExe
- MakeFileContentForExe
- MakeFileNameForLib
- MakeFileContentForLib
- OSFileSystemCaseSensitive
- ParseErrorMessage
- QuoteOMROOT
- RemoteHost
- SpecExtension
- UseNonZeroStdInputHandle
- UseRemoteHost

- InvokeCodeGeneration

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration. To change this property, use the Configuration window in the browser - do not change it by using the Properties window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link. The possible values are as follows:

Debug - Generate the debug command that is set in the makefile.

DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

Release - Generate the release command that is set in the makefile.

**ReleaseNoExp** - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

(Default = Debug)

## buildFrameworkCommand

```
(Default = \"$OMROOT/etc/Executer.exe\"  
|\"|\"|$OMROOT\\..\\Sodius\\RiA_CG\\bin\\recompile_ObjectAda.bat||\"|\"")
```

## CompileSwitches

The `CompileSwitches` property specifies the compiler switches. This property replaces the `CPPCompileSwitches` property.

(Default = )

## DebugSwitches

The `DebugSwitches` property sets the debug level used in debug switches.

(Default = -ga)

## ErrorMessageTokensFormat

The `ErrorMessageTokensFormat`, working with the `ParseErrorMessage` property, specifies the expected

format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens has three parameters, each with an integer value:

TotalNumberOfTokens - The number of tokens in the regular expression

FileTokenPosition - The position of the file name token in the expression

LineTokenPosition - The position of the line number token in the expression

*(Default = TotalNumberOfTokens=3,FileTokenPosition=1,LineTokenPosition=3)*

## ExeExtension

The ExeExtension property specifies the extension that is appended to compiled executable components for a given environment.

*(Default = .exe)*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment.

*(Default = .adb)*

## InvokeAnimatedExecutable

The InvokeExecutable property specifies the command used to run an animated executable file.

*(Default = \$executable \$port)*

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*(Default = \"\$executable\")*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.



```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

*(Default = \"\$OMROOT/etc/Executer.exe\" \"\\\"\$OMROOT\\etc\\ObjectAdaMake.bat\\\" \$makefile \$maketarget\")*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format.

If this is Checked:

The file name is not truncated.

If the FileName property is not blank, its value overrides any automatic file name synthesis.

If the file name is longer than eight characters, the Checker reports this prior to code generation.

*(Default = False)*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*(Default = .lib)*

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

*(Default = )*

## MakeExtension

The MakeExtension property specifies the extension that Rational Rhapsody appends to makefiles.

*(Default = .bat)*

## MakeFileNameForExe

This property is used to specify the filename for the makefile generated by Rhapsody from the corresponding MakeFileContentForExe property.

Note that this property specifies the entire filename, including the extension.

*(Default = \$ComponentName\$MakeExtension)*

## **MakeFileContentForExe**

This property defines the content of a file whose name is defined in the corresponding MakeFileNameForExe property.

The code generator will create the file and take the file content from the value of this property.

The value of the property can contain text and macros. Macro replacement depends on the code generator rules called by the macro. See the RiA CG User Guide for details about the list of macros.

This property is used for executable components, while the MakeFileContentForLib property is used for library components.

AdaCGObjectAdaMakeFile

*(Default = \$AdaCGObjectAdaMakefile)*

## **MakeFileNameForLib**

This property is used to specify the filename for the makefile generated by Rhapsody from the corresponding MakeFileContentForLib property.

Note that this property specifies the entire filename, including the extension.

*(Default = \$ComponentName\$MakeExtension)*

## **MakeFileContentForLib**

This property defines the content of a file whose name is defined in the corresponding MakeFileNameForLib property.

The code generator will create the file and take the file content from the value of this property.

The value of the property can contain text and macros. Macro replacement depends on the code generator rules called by the macro. See the RiA CG User Guide for details about the list of macros.

This property is used for library components, while the MakeFileContentForExe property is used for executable components.

*(Default = \$AdaCGObjectAdaMakefile)*

## **OSFileSystemCaseSensitive**

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*(Default = False)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example:

(lineNumber), (fileName), and (Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*(Default = ([^:]+)[: ](Warning/Error)[:] line ([0-9]+))*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*(Default = True)*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely.

In such configurations, the host is the machine running Rational Rhapsody, whereas the target is the machine running the application.

To run remotely, the UseRemoteHost property must be set to Checked. If UseRemoteHost is Checked and RemoteHost is blank, the current host name is used for the remote host. You can use this as a workaround if you have problems running animated applications on Windows 95.

The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*(Default = Empty string)*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*(Default = .ads)*

## UseNonZeroStdInputHandle

The `UseNonZeroStdInputHandle` property is a Boolean value that specifies whether to use a non-zero standard input handle.

*(Default = False)*

## UseRemoteHost

The `UseRemoteHost` property specifies whether you intend to run an application remotely over a network by default in a given environment.

*(Default = False)*

## InvokeCodeGeneration

The `InvokeCodeGeneration` property specifies the command line used by Rational Rhapsody to run an external code generator. The generator should implement the `IRPEXternalCodeGenerator` connection point. To use an external code generator, you need the appropriate license; an external generator license is part of the Rational Rhapsody Developer for Ada package license. The default values are as follows:

Metaclass Default Value

GNAT "\$OMROOT/etc/Executer.exe" "\$OMROOT/etc/invokeScriptor.bat"

MultiWin32

OBJECTADA

RAVEN\_PPC

SPARK

INTEGRITY "\$OMROOT/etc/Executer.exe" "\$OMROOT\etc\IntegrityAdaMake.bat \$makefile \$maketarget"

*(Default = Empty string)*

## RAVEN\_PPC

This metaclass contains the properties that manipulate the `RAVEN_PPC` operating system environment.

The properties are as follows:

- `BuildCommandSet`
- `BSP_Libraries`

- CompileSwitches
- DebugSwitches
- ErrorMessageTokensFormat
- ExeExtension
- ImpExtension
- InvokeExecutable
- InvokeMake
- IsFileNameShort
- LibExtension
- LinkSwitches
- MakeExtension
- MakeFileNameForExe
- MakeFileContentForExe
- MakeFileNameForLib
- MakeFileContentForLib
- OSFileSystemCaseSensitive
- ParseErrorMessage
- QuoteOMROOT
- RemoteHost
- SpecExtension
- UseNonZeroStdInputHandle
- UseRemoteHost

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration. To change this property, use the Configuration window in the browser - do not change it by using the Properties window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link. The possible values are as follows:

Debug - Generate the debug command that is set in the makefile.

DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

Release - Generate the release command that is set in the makefile.

ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*(Default = Debug)*

## BSP\_Libraries

The BSP\_Libraries property specifies the default BSP libraries to link to.

```
"%RAVENROOT%/bsp/raven/standard_model" "%RAVENROOT%/bsp/system/simulator"  
"%RAVENROOT%/lib/extensions"
```

*(Default = \"%RAVENROOT%/bsp/raven/standard\_model\" \"%RAVENROOT%/bsp/system/simulator\"  
\"%RAVENROOT%/lib/extensions\")*

## CompileSwitches

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property.

*(Default = )*

## DebugSwitches

The DebugSwitches property sets the debug level used in debug switches.

*(Default = -ga)*

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected

format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens has three parameters, each with an integer value:

TotalNumberOfTokens - The number of tokens in the regular expression

FileTokenPosition - The position of the file name token in the expression

LineTokenPosition - The position of the line number token in the expression

*(Default = TotalNumberOfTokens=3,FileTokenPosition=1,LineTokenPosition=3)*

## ExeExtension

The ExeExtension property specifies the extension that is appended to compiled executable components for a given environment.

*(Default = .x)*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment.

*(Default = .adb)*

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*(Default = \"\$executable\")*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

*(Default = \"\$OMROOT/etc/Executer.exe\" \"\\\"\$OMROOT\etc\\ObjectAdaRavenPPCMake.bat\\\" \"\$makefile \$maketarget\")*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format.

If this is Checked:

The file name is not truncated.

If the FileName property is not blank, its value overrides any automatic file name synthesis.

If the file name is longer than eight characters, the Checker reports this prior to code generation.

*(Default = False)*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*(Default = .a)*

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

*(Default = )*

## MakeExtension

The MakeExtension property specifies the extension that Rational Rhapsody appends to makefiles.

*(Default = .bat)*

## MakeFileNameForExe

This property is used to specify the filename for the makefile generated by Rhapsody from the corresponding MakeFileContentForExe property.

Note that this property specifies the entire filename, including the extension.

*(Default = \$ComponentName\$MakeExtension)*

## MakeFileContentForExe

This property defines the content of a file whose name is defined in the corresponding MakeFileNameForExe property.

The code generator will create the file and take the file content from the value of this property.

The value of the property can contain text and macros. Macro replacement depends on the code generator rules called by the macro. See the RiA CG User Guide for details about the list of macros.

This property is used for executable components, while the MakeFileContentForLib property is used for library components.

*\$AdaCGObjectAdaMakefile*

*(Default = \$AdaCGObjectAdaMakefile)*



## MakeFileNameForLib

This property is used to specify the filename for the makefile generated by Rhapsody from the corresponding MakeFileContentForLib property.

Note that this property specifies the entire filename, including the extension.

*(Default = \$ComponentName\$MakeExtension)*

## MakeFileContentForLib

This property defines the content of a file whose name is defined in the corresponding MakeFileNameForLib property.

The code generator will create the file and take the file content from the value of this property.

The value of the property can contain text and macros. Macro replacement depends on the code generator rules called by the macro. See the RiA CG User Guide for details about the list of macros.

This property is used for library components, while the MakeFileContentForExe property is used for executable components.

*(Default = \$AdaCGObjectAdaMakefile)*

## OSFileSystemCaseSensitive

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*(Default = False)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example:

(lineNumber), (fileName), and (Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*(Default = ([^:]+)[:](Warning/Error)[:] line ([0-9]+))*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in

double quotation marks in the generated makefile.

*(Default = True)*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely.

In such configurations, the host is the machine running Rational Rhapsody, whereas the target is the machine running the application.

To run remotely, the UseRemoteHost property must be set to Checked. If UseRemoteHost is Checked and RemoteHost is blank, the current host name is used for the remote host. You can use this as a workaround if you have problems running animated applications on Windows 95.

The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*(Default = Empty string)*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*(Default = .ads)*

## UseNonZeroStdInputHandle

The UseNonZeroStdInputHandle property is a Boolean value that specifies whether to use a non-zero standard input handle.

*(Default = False)*

## UseRemoteHost

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*(Default = False)*

# SPARK

This metaclass contains the properties that manipulate the SPARK operating system environment.

The properties are as follows:

- BuildCommandSet
- BriefErrorMessages
- CompileSwitches
- DebugSwitches
- ErrorMessageTokensFormat
- ExeExtension
- ImpExtension
- InvokeExecutable
- InvokeMake
- IsFileNameShort
- LibExtension
- LinkSwitches
- MakeExtension
- MakeFileNameForExe
- MakeFileContentForExe
- MakeFileNameForLib
- MakeFileContentForLib
- OSFileSystemCaseSensitive
- OpenHTMLReports
- ParseErrorMessage
- QuoteOMROOT
- SpecExtension
- TargetConfigurationFileName
- UseNonZeroStdInputHandle

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration. To change this property, use the Configuration window in the browser - do not change it by using the Properties window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link. The possible values are as follows:

Debug - Generate the debug command that is set in the makefile.

DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

Release - Generate the release command that is set in the makefile.

ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag

(:cx\_option=exceptions).

(Default = Debug)

## BriefErrorMessages

The BriefErrorMessages property determines whether a /brief option is generated on SPARK Examiner calls.

(Default = True)

## CompileSwitches

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property.

(Default = )

## DebugSwitches

The DebugSwitches property sets the debug level used in debug switches.

(Default = )

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected

format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens has three parameters, each with an integer value:

TotalNumberOfTokens - The number of tokens in the regular expression

FileTokenPosition - The position of the file name token in the expression

LineTokenPosition - The position of the line number token in the expression

(Default = TotalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2)

## ExeExtension

The ExeExtension property specifies the extension that is appended to compiled executable components for a given environment.

*(Default = .exe)*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment.

*(Default = .adb)*

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*(Default = \"\$executable\")*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

*(Default = \"\$OMROOT/etc/Executer.exe\" \"|\" \"\$OMROOT\etc\SPARKMake.bat\|\" \$makefile \$maketarget\")*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format.

If this is Checked:

The file name is not truncated.

If the FileName property is not blank, its value overrides any automatic file name synthesis.

If the file name is longer than eight characters, the Checker reports this prior to code generation.

*(Default = False)*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*(Default = Empty string)*

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

*(Default = )*

## MakeExtension

The MakeExtension property specifies the extension that Rational Rhapsody appends to makefiles.

*(Default = .bat)*

## MakeFileNameForExe

This property is used to specify the filename for the makefile generated by Rhapsody from the corresponding MakeFileContentForExe property.

Note that this property specifies the entire filename, including the extension.

*(Default = \$ComponentName\$MakeExtension)*

## MakeFileContentForExe

This property defines the content of a file whose name is defined in the corresponding MakeFileNameForExe property.

The code generator will create the file and take the file content from the value of this property.

The value of the property can contain text and macros. Macro replacement depends on the code generator rules called by the macro. See the RiA CG User Guide for details about the list of macros.

This property is used for executable components, while the MakeFileContentForLib property is used for library components.

`$AdaCGSparkAdaMakefile`

*(Default = \$AdaCGSparkAdaMakefile)*

## MakeFileNameForLib

This property is used to specify the filename for the makefile generated by Rhapsody from the corresponding MakeFileContentForLib property.

Note that this property specifies the entire filename, including the extension.

*(Default = \$ComponentName\$MakeExtension)*

## MakeFileContentForLib

This property defines the content of a file whose name is defined in the corresponding MakeFileNameForLib property.

The code generator will create the file and take the file content from the value of this property.

The value of the property can contain text and macros. Macro replacement depends on the code generator rules called by the macro. See the RiA CG User Guide for details about the list of macros.

This property is used for library components, while the MakeFileContentForExe property is used for executable components.

*(Default = \$AdaCGSparkAdaMakefile)*

## OSFileSystemCaseSensitive

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*(Default = False)*

## OpenHTMLReports

The OpenHTMLReports property specifies whether to open the HTML reports when the examination is complete.

*(Default = True)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example:

(lineNumber), (fileName), and (Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment.

These two properties retrieve the file name and line number of errors reported by the compiler.

*(Default = ([^:]+)([:]([0-9]+)[:]?)*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*(Default = True)*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*(Default = .ads)*

## TargetConfigurationFileName

The TargetConfigurationFileName property specifies the name of the target configuration file to be passed as an argument to the SPARK Examiner.

*(Default = Empty string)*

## UseNonZeroStdInputHandle

The UseNonZeroStdInputHandle property is a Boolean value that specifies whether to use a non-zero standard input handle.

*(Default = False)*

## Event

The Event metaclass contains properties that control events.

The properties are as follows:

- AnimInstanceCreate
- DeclarationModifier
- DescriptionTemplate
- EnableDynamicAllocation



## AnimInstanceCreate

The AnimInstanceCreate property affects event creation. If you set the C\_CG::Event::NoDynamicAllocAnimCreate property to False, Rational Rhapsody does not generate the event creation method, effectively disabling the ability to inject the event in animation. To enable the injection of the event, you can specify a different method to obtain an instance of the event by setting this property to the name of the method to use.

*(Default = Empty string)*

## DeclarationModifier

The DeclarationModifier property adds a string to the class or event declaration. The string is displayed between the class keyword and the class name in the generated code. For example, for a class A, the DeclarationModifier would appear as follows:

class DeclarationModifier> A {...}; This property adds a modifier to the class declaration. For example, if you have a class myExportableClass that is exported from a DLL by using the MYDLL\_API macro, you can set the DeclarationModifier property to “MYDLL\_API.” The generated code would then be as follows:

```
class MYDLL_API myExportableClass
```

```
{ ...}; This property supports two keywords: $component and $class.
```

*(Default = Empty string)*

## DescriptionTemplate

The DescriptionTemplate property specifies how to generate the element description in the code. An empty MultiLine (the default value) tells Rational Rhapsody to use the default description generation rules. The property supports the following keywords:

\$Name - The element name

\$FullName - The full path of the element (P1::P2::C.a)

\$Description - The element description

Element-specific keywords, as shown in the following table:

Metatype Describes Additional Supported Keywords

Argument Arguments \$Type - The argument type

\$Direction - The argument direction (in, out, and so on)

Attribute Attributes \$Type - The attribute type

Class Classes, actors, objects, and blocks

Event Events \$Arguments - The description for the event argument

Operation Primitive operations, triggered operations, \$Arguments - The description for the operation argument

constructors, and destructors \$Signature - The operation signature

Package Packages

Relation Association ends \$Target - The other end of the association

Type Types \$Type - Applicable to Typedef types

Tag - The value of the specified the element tag

Property - The value of the element property with the specified name

The keywords are resolved in the following order:

Predefined keywords (such as \$Name)

Tag keywords

Property keywords

Note the following:

Keyword names can be written in parentheses. For example:

\$(Name)

If the value of a keyword is a MultiLine, each new line (except the first one) starts with the value of the ADA\_CG::Configuration::DescriptionBeginLine property; each line ends with the value of the ADA\_CG::Configuration::DescriptionEndLine property.

*(Default = Empty string)*

## **EnableDynamicAllocation**

The EnableDynamicAllocation property specifies whether to use dynamic memory allocation for objects. The possible values are as follows:

True - Dynamic allocation of events is enabled. Create() and Destroy() operations are generated for the object or object type.

False - Events are dynamically allocated during initialization, but not during run time. Create() and Destroy() operations are not generated for the object. This setting is recommended for static architectures that do not use dynamic memory management during run time.

(Default = Empty string)

## Transition

The Transition metaclass contains properties that control transitions.

The properties are as follows:

- DescriptionTemplate

### DescriptionTemplate

The DescriptionTemplate property specifies how to generate the element description in the code. An empty MultiLine (the default value) tells Rational Rhapsody to use the default description generation rules. The property supports the following keywords:

\$Name - The element name

\$FullName - The full path of the element (P1::P2::C.a)

\$Description - The element description

Element-specific keywords, as shown in the following table:

Metatype Describes Additional Supported Keywords

Argument Arguments \$Type - The argument type

\$Direction - The argument direction (in, out, and so on)

Attribute Attributes \$Type - The attribute type

Class Classes, actors, objects, and blocks

Event Events \$Arguments - The description for the event argument

Operation Primitive operations, triggered operations, \$Arguments - The description for the operation argument

constructors, and destructors \$Signature - The operation signature

Package Packages

Relation Association ends \$Target - The other end of the association

Type Types \$Type - Applicable to Typedef types

Tag - The value of the specified the element tag

Property - The value of the element property with the specified name

The keywords are resolved in the following order:

Predefined keywords (such as \$Name)

Tag keywords

Property keywords

Note the following:

Keyword names can be written in parentheses. For example:

\$(Name)

If the value of a keyword is a MultiLine, each new line (except the first one) starts with the value of the ADA\_CG::Configuration::DescriptionBeginLine property; each line ends with the value of the ADA\_CG::Configuration::DescriptionEndLine property.

(Default = )

## Generalization

The Generalization metaclass contains a property used to support generalization.

The properties are as follows:

- Animate

### Animate

The Animate property specifies whether animation code is generated for an element. You can specify your own animation function by using the CG::Attribute::AnimSerializeOperation property. The semantics of the Animate property is always in favor of the owner settings:

If a package Animate property is set to False, all the classes owned by the package are not animated, regardless of the class Animate settings.

If a class Animate property is set to False, all the elements in the class (attributes, operations, relations,

and so on) are not animated.

If an operation `Animate` property is set to `False`, all the arguments are not animated.

If the `AnimateArguments` property is set to `False`, all the arguments are not animated, regardless of the specific argument `Animate` property settings.

*(Default = Empty string)*

# ADA\_Roundtrip

---

The ADA\_Roundtrip subject contains metaclasses that contain properties that affect roundtripping.

## General

The General metaclass contains properties that control how changes to code are roundtripped in Rational Rhapsody.

### NotifyOnInvalidatedModel

*The NotifyOnInvalidatedModel property is a Boolean value that determines whether a warning window is displayed during roundtrip. This warning is displayed when information might be lost because the model was changed between the last code generation and the roundtrip operation. (Default = True)*

### ParserErrors

The ParserErrors property specifies the behavior of a round trip when a parser error is encountered. The possible values are as follows:

- Abort - Stop the round trip whenever there is a parser error in the code. No changes are applied to the model.
- AskUser - When Rational Rhapsody encounters an error, the program shows a message asking what you want to do. This option is available in Rational Rhapsody Developer for C++ only.
- AbortOnCritical - Stop the round trip if any critical parser errors are encountered in the code.
- Ignore - Continue the round trip, ignoring any parser errors that are encountered.

*Default = Abort*

### PredefineIncludes

The PredefineIncludes property specifies the predefined include path for roundtripping. The default value for C++ is an empty string. The default value for Java is as follows:  
\$OMROOT\LangJava\src,D:\jdk1.2.2\src

### PredefineMacros

The PredefineMacros property specifies the predefined macros for roundtripping. The default value is as follows:

```
DECLARE_META(class_0\,animClass_0), DECLARE_REACTIVE_META(class_0\,animClass_0),  
OMINIT_SUPERCLASS(class_0Super\,animClass_0Super),  
OMREGISTER_CLASS\,DECLARE_META_T(class_0\, ttype\,animClass_0),
```

```

DECLARE_REACTIVE_META_T(class_0\, ttype\,animClass_0),
DECLARE_META_SUBCLASS_T(class_0\, ttype\,animClass_0),
DECLARE_REACTIVE_META_SUBCLASS_T(class_0\, ttype\,animClass_0),
DECLARE_MEMORY_ALLOCATOR(CLASSNAME\,INITNUM),
IMPLEMENT_META(class_0\,Default\,FALSE),
IMPLEMENT_META_S(class_0\,FALSE\,class_1\,animClass_1\, animClass_0),
IMPLEMENT_META_M(class_0\, FALSE\, class_0Super\, 2\,animClass_0),
IMPLEMENT_REACTIVE_META(class_0\,Default\,FALSE),
IMPLEMENT_REACTIVE_META_S(class_0\,FALSE\,class_1\, animClass_1\,animClass_0),
IMPLEMENT_REACTIVE_META_M(class_0\, FALSE\, class_0Super\, 2 \,animClass_0),
IMPLEMENT_REACTIVE_META_SIMPLE(class_0\,Default\,FALSE),
IMPLEMENT_REACTIVE_META_S_SIMPLE(class_0\,FALSE\,class_1\ ,animClass_1\,animClass_0),
IMPLEMENT_REACTIVE_META_M_SIMPLE(class_0\, FALSE\, class_0Super\, 2 \,animClass_0),
IMPLEMENT_META_T(class_0\, Default\, FALSE\, animClass_0),
IMPLEMENT_META_S_T(class_0\,FALSE\,class_0Super\,animclas s_0Super\,animClass_0),
IMPLEMENT_META_M_T(class_0\, FALSE\, class_0Super\, 2 \,animClass_0),
IMPLEMENT_META_OBJECT(class_0\,class_type\,Default\, FALSE),
IMPLEMENT_META_S_OBJECT(class_0\,class_type\,FALSE\, class_1\,animClass_1\,animClass_0),
IMPLEMENT_META_M_OBJECT(class_0\,class_type\,FALSE\, class_0Super\, 2 \,animClass_0),
IMPLEMENT_REACTIVE_META_OBJECT(class_0\,class_type\, Default\,FALSE),
IMPLEMENT_REACTIVE_META_S_OBJECT(class_0\,class_type\,
FALSE\,class_1\,animClass_1\,animClass_0),
IMPLEMENT_REACTIVE_META_M_OBJECT(class_0\,class_type\, FALSE\, class_0Super\, 2
\,animClass_0), IMPLEMENT_REACTIVE_META_SIMPLE_OBJECT(class_0\,
class_type\,Default\,FALSE), IMPLEMENT_REACTIVE_META_S_SIMPLE_OBJECT(class_0\,
class_type\,FALSE\,class_1\,animClass_1\,animClass_0),
IMPLEMENT_REACTIVE_META_M_SIMPLE_OBJECT(class_0\, class_type\,FALSE\, class_0Super\,
2 \,animClass_0), IMPLEMENT_META_T_OBJECT(class_0\,class_type\, Default\, FALSE\,
animClass_0), IMPLEMENT_META_S_T_OBJECT(class_0\,class_type\,FALSE\,
class_0Super\,animclass_0Super\,animClass_0),
IMPLEMENT_META_M_T_OBJECT(class_0\,class_type\, FALSE\, class_0Super\, 2 \,animClass_0),
IMPLEMENT_MEMORY_ALLOCATOR(CLASSNAME\,INITNUM\,
INCREMENTNUM\,ISPROTECTED), DECLARE_META_PACKAGE(Default),
DECLARE_PACKAGE(Default), IMPLEMENT_META_PACKAGE(Default\,Default),
DECLARE_META_EVENT(event_0), DECLARE_META_SUBEVENT(event_0\,event_0Super\,
event_0SuperNamespace), IMPLEMENT_META_EVENT(event_0\,Default\,event_0),
IMPLEMENT_META_EVENT_S(words\, words\, baseWords),
DECLARE_OPERATION_CLASS(mangledName), DECLARE_META_OP(mangledName),
OM_OP_UNSER(type\, name), OP_UNSER(func\, name), OP_SET_RET_VAL(retVal),
OM_OP_SET_RET_VAL(retVal), IMPLEMENT_META_OP(animatedClassName\, mangledName\,
opNameStr\, isStatic\, signatureStr\, numOfArgs), IMPLEMENT_OP_CALL(mangledName\,
userClassName\, call\, retExp), STATIC_IMPLEMENT_OP_CALL(mangledName\, userClassName\,
call\, retExp), OMDefaultThread=0, NULL=0, OMDECLARE_GUARDED
OM_DECLARE_COMPOSITE_OFFSET

```

## ReportChanges

The ReportChanges property defines which changes are reported and displayed by the roundtrip operation. The possible values are as follows:

- None—No changes are shown in the output window.
- AddRemove—Only the elements added to, or removed from, the model are shown in the output window.

- UpdateFailures—Only unsuccessful changes to the model are shown in the output window.
- All—All changes to the model are shown in the output window.

*(Default = AddRemove)*

## RestrictedMode

The RestrictedMode property is a Boolean value that specifies whether restricted-mode roundtripping is enabled. This property can be modified on the configuration level. Restricted mode of full round trips roundtrips unusual usage of Rational Rhapsody elements, such as a class declaration in a user-defined type. Restricted mode has more limitations, but preserves the model from unexpected changes. The additional limitations for restricted mode are as follows:

- User-defined types cannot be removed or changed on round trips because Rational Rhapsody code generation adds the Ignore annotation for a user-defined type declaration.
- Relations cannot be removed or changed on roundtrip.
- New classes are not added to the model.

*(Default = False)*

## RoundtripScheme

The RoundtripScheme property specifies whether to perform a basic or full round trip. Batch and online round trips change their behavior according to the specified value.

*Default = Basic*

## CPP\_Roundtrip::Type

The Type metaclass contains a property that controls whether user-defined types are ignored during the roundtrip operation.

### Ignore

The Ignore property is a Boolean value that specifies whether to include user-defined types in a roundtrip operation. Types with the Ignore property set to True are generated with an Ignore annotation and does not change when a roundtrip operation is performed.

The default value of this property is True, which allows no deletion or change to be done on types.

*Setting this property to False reflects changes to the types declaration and deletion of types during round trips. Modifying the name of an existing type results in the addition of a new type, and removal of the model type (if the AcceptChanges property allows element removal), and the references for the model to the removed type is lost (such as appearance in diagrams, property settings, and so on). You can set this property either on the configuration or on specific elements in the model (which means the property*



*affects the elements and its aggregates). (Default = True)*

## Update

The Update metaclass contains a property that controls the update process used during roundtripping.

### AcceptChanges

The AcceptChanges property is an enumerated type that specifies which changes are applied to each CG element (attribute, operation, type, class, or package). You can apply separate properties to each type of CG element. The possible values are as follows:

- All—All the changes can be applied to the model element.
- NoDelete—All the changes except deletion can be applied to the model element. This setting prevents accidental removal of operations, constructors, attributes, relations, variables, instances, and functions.
- AddOnly—Apply only the addition of an aggregate to the model element. You cannot delete or change elements.
- NoChanges—Do not apply any changes to the model element.

Note that the value of the property is propagated to all the aggregates of an element. Therefore, if a package has the NoChanges value, no elements in that package is changed.

*(Default = NoDelete)*

# Ada2005Containers

---

The Ada2005Containers subject defines the property set to generate code for relations which need a container set. The container set used in this subject is provided by Ada2005 standard library. Each metaclass of this subject defines a specific implementation. If user needs it, he can create its own metaclass, for other kind of implementation. If done, the property CG.Relation.Implementation must be updated with the new metaclass.

The metaclasses are as follows:

- General
- StaticArray
- Bounded
- Unbounded
- BoundedQualified
- UnboundedQualified

## General

The General metaclass provides the properties used to generate some code in makefile.

The properties are as follows:

- GnatCompilerInclude
- ObjectAdaCompilerIncludeBounded
- ObjectAdaCompilerIncludeUnbounded
- ObjectAdaCompilerIncludeBoundedQualified
- ObjectAdaCompilerIncludeUnboundedQualified

## GnatCompilerInclude

This property is used to generate some compiler options in makefile for GNAT compiler.

(*Default* = )

## ObjectAdaCompilerIncludeBounded

This property is used to generate some compiler options in makefile for ObjectAda compiler.

It is used for bounded relation.

It is possible to create a new property for another kind of implementation by replacing "Bounded" suffix with the name of a new implementation.

*(Default = )*

## **ObjectAdaCompilerIncludeUnbounded**

This property is used to generate some compiler options in makefile for ObjectAda compiler.

It is used for Unbounded relation.

It is possible to create a new property for another kind of implementation by replacing "Unbounded" suffix with the name of a new implementation.

*(Default = )*

## **ObjectAdaCompilerIncludeBoundedQualified**

This property is used to generate some compiler options in makefile for ObjectAda compiler.

It is used for bounded qualified relation.

It is possible to create a new property for another kind of implementation by replacing "BoundedQualified" suffix with the name of a new implementation.

*(Default = )*

## **ObjectAdaCompilerIncludeUnboundedQualified**

This property is used to generate some compiler options in makefile for ObjectAda compiler.

It is used for unbounded qualified relation.

It is possible to create a new property for another kind of implementation by replacing "UnboundedQualified" suffix with the name of a new implementation.

*(Default = )*

## **StaticArray**

The StaticArray metaclass provides properties to generate code for relations which needs StaticArray container

The properties are as follows:

- WithClauseInSpec
- WithClauseInBody

- TypeDeclaration
- TypeDeclarationAfterClassRecord
- TypeDeclarationInBody
- TypeDeclarationInBodyRegular
- RelationType
- IndexType
- GetAtPosProcedure
- GetAtPosFunction
- Contains
- GetCount
- GetCountRegular
- Set
- Add
- AddAtPos
- Remove
- RemoveAtPos

## WithClauseInSpec

This property is used to generate with clause for the specified implementation in the specification file

*(Default = )*

## WithClauseInBody

This property is used to generate with clause for the specified implementation in the body file

*(Default = )*

## TypeDeclaration

This property is used to define some type's declaration or operation's declaration in specification file, before class record type.

*(Default = )*

## TypeDeclarationAfterClassRecord

This property is used to define some type's declaration or operation's declaration in specification file, after class record type.

*(Default = )*

## TypeDeclarationInBody

This property is used to define some type's declaration or operation's declaration in body file.

*(Default = )*

## TypeDeclarationInBodyRegular

This property is used to define some type's declaration or operation's declaration in body file. It is used is the value of the property Ada\_CG.relation.AccessTypeUsage is Regular or WideClass.

*(Default = )*

## RelationType

This property is used to generate the type of the relation inside the class record type.

*(Default = Empty string)*

## IndexType

This property is used to generate the specified type of the index in signature of some getter/setter which use an index. If this property does not exist, or is empty, CG will generate the type "Natural".

*(Default = Empty string)*

## GetAtPosProcedure

This property contains the code which will be generated in the implement of procedure of kind GetAtPos. This procedure returns the element which is located at the specified position in the list.

*(Default = )*

## GetAtPosFunction

This property contains the code which will be generated in the implementation of function of kind GetAtPos. This function returns the element which is location at the specified position in the list

*(Default = )*

## Contains

This property contains the code which will be generated in the implementation of function of kind

Contains. It returns true if the specified element is present in the list

*(Default = )*

## **GetCount**

This property contains the code which will be generated in the implementation of function of kind GetCount. It returns the number of element contained in the list

*(Default = )*

## **GetCountRegular**

This property contains the code which will be generated in the implementation of function of kind GetCount. It returns the number of element contained in the list. It is used is the value of the property Ada\_CG.relation.AccessTypeUsage is Regular or WideClass.

*(Default = )*

## **Set**

This property contains the code which will be generated in the implementation of function of kind Set. It puts an element at the specified position in the list.

*(Default = )*

## **Add**

This property contains the code which will be generated in the implementation of function of kind Add. It adds an element at the end of the list.

*(Default = )*

## **AddAtPos**

This property contains the code which will be generated in the implementation of function of kind AddAtPos. It adds an element at the specified position of the list.

*(Default = )*

## **Remove**

This property contains the code which will be generated in the implementation of function of kind Remove. It removes a specified element from the list.

(Default = )

## RemoveAtPos

This property contains the code which will be generated in the implementation of function of kind RemoveAtPos. It removes the element at specified position from the list.

(Default = )

## Bounded

The Bounded metaclass provides properties to generate code for relations which needs Bounded container

The properties are as follows:

- WithClauseInSpec
- WithClauseInBody
- TypeDeclaration
- RelationType
- IndexType
- GetAtPosProcedure
- GetAtPosFunction
- GetAtPosProcedureRegular
- GetAtPosFunctionRegular
- Contains
- GetCount
- Find
- Set
- Add
- AddAtPos
- Remove
- RemoveAtPos

## WithClauseInSpec

This property is used to generate with clause for the specified implementation in the specification file

(Default = )

## WithClauseInBody

This property is used to generate with clause for the specified implementation in the body file

*(Default = )*

## TypeDeclaration

This property is used to define some type's declaration or operation's declaration in specification file, before class record type.

*(Default = )*

## RelationType

This property is used to generate the type of the relation inside the class record type.

*(Default = Empty string)*

## IndexType

This property is used to generate the specified type of the index in signature of some getter/setter which use an index. If this property does not exist, or is empty, CG will generate the type "Natural".

*(Default = Empty string)*

## GetAtPosProcedure

This property contains the code which will be generated in the implementation of procedure of kind GetAtPos. This procedure returns the element which is located at the specified position in the list.

*(Default = )*

## GetAtPosFunction

This property contains the code which will be generated in the implementation of function of kind GetAtPos. This function returns the element which is location at the specified position in the list

*(Default = )*

## GetAtPosProcedureRegular

This property contains the code which will be generated in the implementation of procedure of kind



GetAtPos. This procedure returns the element which is located at the specified position in the list. It is used is the value of the property Ada\_CG.relation.AccessTypeUsage is Regular or WideClass.

*(Default = )*

## **GetAtPosFunctionRegular**

This property contains the code which will be generated in the implementation of function of kind GetAtPos. This function returns the element which is location at the specified position in the list. It is used is the value of the property Ada\_CG.relation.AccessTypeUsage is Regular or WideClass.

*(Default = )*

## **Contains**

This property contains the code which will be generated in the implementation of function of kind Contains. It returns true if the specified element is present in the list

*(Default = )*

## **GetCount**

This property contains the code which will be generated in the implementation of function of kind GetCount. It returns the number of element contained in the list

*(Default = )*

## **Find**

This property contains the code which will be generated in the implementation of function of kind Find.

*(Default = )*

## **Set**

This property contains the code which will be generated in the implementation of function of kind Set. It puts an element at the specified position in the list.

*(Default = )*

## **Add**

This property contains the code which will be generated in the implementation of function of kind Add. It adds en element at the end of the list.

(Default = )

## AddAtPos

This property contains the code which will be generated in the implementation of function of kind AddAtPos. It adds an element at the specified position of the list.

(Default = )

## Remove

This property contains the code which will be generated in the implementation of function of kind Remove. It removes a specified element from the list.

(Default = )

## RemoveAtPos

This property contains the code which will be generated in the implementation of function of kind RemoveAtPos. It removes the element at specified position from the list.

(Default = )

# Unbounded

The Unbounded metaclass provides properties to generate code for relations which needs Unbounded container

The properties are as follows:

- WithClauseInSpec
- WithClauseInBody
- TypeDeclaration
- RelationType
- IndexType
- GetAtPosProcedure
- GetAtPosFunction
- GetAtPosProcedureRegular
- GetAtPosFunctionRegular
- Contains
- GetCount

- Find
- Set
- Add
- AddAtPos
- Remove
- RemoveAtPos

## WithClauseInSpec

This property is used to generate with clause for the specified implementation in the specification file

*(Default = )*

## WithClauseInBody

This property is used to generate with clause for the specified implementation in the body file

*(Default = )*

## TypeDeclaration

This property is used to define some type's declaration or operation's declaration in specification file, before class record type.

*(Default = )*

## RelationType

This property is used to generate the type of the relation inside the class record type.

*(Default = Empty string)*

## IndexType

This property is used to generate the specified type of the index in signature of some getter/setter which use an index. If this property does not exist, or is empty, CG will generate the type "Natural".

*(Default = Empty string)*

## GetAtPosProcedure

This property contains the code which will be generated in the implementation of procedure of kind GetAtPos. This procedure returns the element which is located at the specified position in the list.

*(Default = )*

## **GetAtPosFunction**

This property contains the code which will be generated in the implementation of function of kind GetAtPos. This function returns the element which is location at the specified position in the list

*(Default = )*

## **GetAtPosProcedureRegular**

This property contains the code which will be generated in the implementation of procedure of kind GetAtPos. This procedure returns the element which is located at the specified position in the list. It is used is the value of the property Ada\_CG.relation.AccessTypeUsage is Regular or WideClass.

*(Default = )*

## **GetAtPosFunctionRegular**

This property contains the code which will be generated in the implementation of function of kind GetAtPos. This function returns the element which is location at the specified position in the list. It is used is the value of the property Ada\_CG.relation.AccessTypeUsage is Regular or WideClass.

*(Default = )*

## **Contains**

This property contains the code which will be generated in the implementation of function of kind Contains. It returns true if the specified element is present in the list

*(Default = )*

## **GetCount**

This property contains the code which will be generated in the implementation of function of kind GetCount. It returns the number of element contained in the list

*(Default = )*

## **Find**

This property contains the code which will be generated in the implementation of function of kind Find.

*(Default = )*

## Set

This property contains the code which will be generated in the implementation of function of kind Set. It puts an element at the specified position in the list.

*(Default = )*

## Add

This property contains the code which will be generated in the implementation of function of kind Add. It adds an element at the end of the list.

*(Default = )*

## AddAtPos

This property contains the code which will be generated in the implementation of function of kind AddAtPos. It adds an element at the specified position of the list.

*(Default = )*

## Remove

This property contains the code which will be generated in the implementation of function of kind Remove. It removes a specified element from the list.

*(Default = )*

## RemoveAtPos

This property contains the code which will be generated in the implementation of function of kind RemoveAtPos. It removes the element at specified position from the list.

*(Default = )*

## BoundedQualified

The BoundedQualified metaclass provides properties to generate code for relations which needs BoundedQualified container

The properties are as follows:

- WithClauseInSpec

- WithClauseInBody
- TypeDeclaration
- TypeDeclarationInBody
- RelationType
- GetAtPosProcedure
- GetAtPosFunction
- Contains
- GetCount
- Set
- Remove

## WithClauseInSpec

This property is used to generate with clause for the specified implementation in the specification file

*(Default = )*

## WithClauseInBody

This property is used to generate with clause for the specified implementation in the body file

*(Default = )*

## TypeDeclaration

This property is used to define some type's declaration or operation's declaration in specification file, before class record type.

*(Default = )*

## TypeDeclarationInBody

This property is used to define some type's declaration or operation's declaration in body file.

*(Default = )*

## RelationType

This property is used to generate the type of the relation inside the class record type.

*(Default = Empty string)*

## GetAtPosProcedure

This property contains the code which will be generated in the implementation of procedure of kind GetAtPos. This procedure returns the element which is located at the specified position in the list.

*(Default = )*

## GetAtPosFunction

This property contains the code which will be generated in the implementation of function of kind GetAtPos. This function returns the element which is location at the specified position in the list

*(Default = )*

## Contains

This property contains the code which will be generated in the implementation of function of kind Contains. It returns true if the specified element is present in the list

*(Default = )*

## GetCount

This property contains the code which will be generated in the implementation of function of kind GetCount. It returns the number of element contained in the list

*(Default = )*

## Set

This property contains the code which will be generated in the implementation of function of kind Set. It puts an element at the specified position in the list.

*(Default = )*

## Remove

This property contains the code which will be generated in the implementation of function of kind Remove. It removes a specified element from the list.

*(Default = )*

# UnboundedQualified

The UnboundedQualified metaclass provides properties to generate code for relations which needs UnboundedQualified container

The properties are as follows:

- WithClauseInSpec
- WithClauseInBody
- TypeDeclaration
- TypeDeclarationInBody
- RelationType
- GetAtPosProcedure
- GetAtPosFunction
- Contains
- GetCount
- Set
- Remove

## WithClauseInSpec

This property is used to generate with clause for the specified implementation in the specification file

*(Default = )*

## WithClauseInBody

This property is used to generate with clause for the specified implementation in the body file

*(Default = )*

## TypeDeclaration

This property is used to define some type's declaration or operation's declaration in specification file, before class record type.

*(Default = )*

## TypeDeclarationInBody



This property is used to define some type's declaration or operation's declaration in body file.

*(Default = )*

## **RelationType**

This property is used to generate the type of the relation inside the class record type.

*(Default = Empty string)*

## **GetAtPosProcedure**

This property contains the code which will be generated in the implementation of procedure of kind GetAtPos. This procedure returns the element which is located at the specified position in the list.

*(Default = )*

## **GetAtPosFunction**

This property contains the code which will be generated in the implementation of function of kind GetAtPos. This function returns the element which is location at the specified position in the list

*(Default = )*

## **Contains**

This property contains the code which will be generated in the implementation of function of kind Contains. It returns true if the specified element is present in the list

*(Default = )*

## **GetCount**

This property contains the code which will be generated in the implementation of function of kind GetCount. It returns the number of element contained in the list

*(Default = )*

## **Set**

This property contains the code which will be generated in the implementation of function of kind Set. It puts an element at the specified position in the list.

*(Default = )*

## Remove

This property contains the code which will be generated in the implementation of function of kind Remove. It removes a specified element from the list.

(*Default* = )

# Animation

---

The Animation subject contains metaclasses that contain properties that support black box animation and target monitoring.

## Activity

The activity feature shows the numbers of tokens on the animated active flows.

### ShowFlowToken

It will show the tokens on the active flows.

The values are:

- True - the tokens will be indicated on the active flows

*Default = False*

## ClassifierRole

The ClassifierRole metaclass contains properties that control black box animation.

### DisplayMessagesToSelf

The DisplayMessagesToSelf property determines whether messages-to-self are displayed during animation. The possible values are as follows:

- None - Do not display any messages-to-self.
- All - Display all messages-to-self.

For example, if lifeline L1 is mapped to objects O1 and O2, you can suppress the messages between O1 and O2 for this lifeline. Even if a lifeline is mapped to a single object, the messages the object sends to itself (for example, timeout events) can be suppressed.

*Default = All*

### MapVariationPointToVariant

If the MapVariationPointToVariant property is set to True for a lifeline in a diagram, then when the animated version of the diagram is created, the cloned lifeline will be set to represent the variant specified

for the corresponding variation point in the active component.

*Default = True*

## MappingPolicy

The MappingPolicy property specifies how lifelines are mapped during animation. You can set these properties for every lifeline in the diagram. The possible values are as follows:

- Smart - Rational Rhapsody decides the mapping policy.
- If the lifeline has a reference sequence diagram, the mapping is equivalent to ObjectAndDerivedFromRefSD; otherwise, the mapping is equivalent to ObjectAndItsParts (which, for an object without any parts, is the same as SingleObject).
- ObjectAndItsParts - The lifeline (classifier role) is mapped to the object that matches the name of the lifeline and all its parts (recursively), excluding parts that are explicitly shown in the diagram.
- This is the default value for a lifeline realized by a composite class that does not reference sequence diagrams (SDs) when Smart mode is used.
- SingleObject - The lifeline is mapped to a single, run-time object.
- ObjectAndDerivedFromRefSD - The lifeline is mapped to an object by its role name (if it exists) and to all derived objects from the reference SDs (according to their mapping rules).
- This is the default value for lifelines decomposed by reference SDs when Smart mode is used.

Note the following:

- If the lifeline can be mapped to a composite object, the compositional hierarchy is ignored in this mapping.
- Parts that are represented by other lifelines are excluded. For example, if two lifelines are decomposed to the same reference SD, one of them is completely ignored (arbitrarily).

*Default = SingleObject*

## General

The auto-open feature opens only activities generated when the value of the property Activity::General::SimulationMode is TokenOriented.

## AutoOpenBehavioralDiagrams

Ordinarily, statecharts and activity diagrams are not opened automatically during animation. You must first manually open the diagrams that you would like to follow during the animation.

The AutoOpenBehavioralDiagrams property can be used to specify that Rational Rhapsody should automatically open statecharts and activity diagrams as soon as they become relevant during the course of running an animated application.

If you set the value of the property to Always, diagrams will be automatically opened during animation even if you are using the Go button to run the application. Having various diagrams open at this kind of pace can result in situations where you find it difficult to follow the flow of the application. Therefore, this property provides an additional option - WhileStepping. When the property is set to WhileStepping, behavioral diagrams are only opened when they become relevant while you are using Go Step, Go Event, Go Action, or Go Idle, resulting in an easier-to-follow animation sequence.

*Default = Never*

## **AutoSetBreakpoints**

The AutoSetBreakpoints property determines whether breakpoints set for animation are saved so that you can reuse them later on in your Rational Rhapsody session or even in a new session.

When the value of the property is set to True, the breakpoints that you set are saved.

*Default = True*

## **EnableAutoScroll**

When viewing animated statecharts or activity diagrams that are larger than your current view, there may be situations where the applications progresses through states or actions that are not actually visible to you.

To prevent such situations, you can set the value of the EnableAutoScroll property to True. When the value is set to True, Rational Rhapsody will automatically adjust the portion of the diagram displayed so that the current state/action is always visible.

*Default = True*

## **EnableFragmentSimulation**

By default this feature is enabled only for SysML. By using this hidden property, you can also enable the feature for a non-SysML project.

Default value: By default the value is true for SysML projects, and false for other type projects.

## **FragmentSimulationScope**

This property is turned on by default in SysML. If the SysML profile is not loaded, it is off by default.

By default Rhapsody generates minimal code for the Model Fragment simulation. This means that global instances, functions and variables are not going to be generated. If you use them, you can change the above property to 'AllRelatedElements', to add those elements to the generation scope.

## Message

The Message metaclass contains properties related to the display of messages on animated sequence diagrams.

### DisplaySenderReceiverForEnvMsg

In a sequence diagram, if a message was drawn from or to the system border (ENV), then during animation of the diagram Rational Rhapsody will include in the message label the name of the actual sender or receiver of the message.

If you set the value of the DisplaySenderReceiverForEnvMsg property to False, the name of the actual sender or receiver of the message will not be displayed during animation.

*Default = True*

## TargetMonitoring

The TargetMonitoring metaclass contains properties that control target monitoring. Target monitoring is an animation instrumentation mode in which instrumentation code is optimized and limited, and the user cannot control the execution of the application on the target, only to view its status.

### ExecutionLog

The ExecutionLog property is a boolean property that indicates whether Rational Rhapsody should maintain a log file during target monitoring to record the execution time for doExecute functions and periodic operations.

*Default = Cleared*

### ExecutionLogBeginCode

The ExecutionLogBeginCode property is used to specify the logging-related instrumentation code added to the beginning of functions whose steps will be logged. This includes doExecute functions, periodic operations (in MicroC), and active operations (in AR3x\_BMT).

```
Default = #ifdef _OM_TARGET_MON /* Target Monitoring - Execution Log begin */ { timeUnit  
tgtMonTrceBeginTime = RiCTimerManager_getSystemTimerElapsedTime(); /* block closed at  
instrumentation end */ #endif
```

### ExecutionLogEndCode

The ExecutionLogEndCode property is used to specify the logging-related instrumentation code added to the end of functions whose steps will be logged. This includes doExecute functions, periodic operations (in MicroC), and active operations (in AR3x\_BMT)

```
Default = #ifdef _OM_TARGET_MON /* block started at Target Monitoring Execution Log Begin */
START_TGT_MON_PARAM_MSG(TGT_MON_RSRVD_TRACE_SEND, $me, NULL);
OM_TARGET_MON_SendShort((short)1); OM_TARGET_MON_SendShort((short)$TraceIdName);
OM_TARGET_MON_SendVoidP((void*)$me); OM_TARGET_MON_SendUnsignedLong((unsigned
long)tgtMonTrceBeginTime); OM_TARGET_MON_SendUnsignedLong((unsigned
long)RiCTimerManager_getSystemTimerElapsedTime()); END_TGT_MON_MSG(NULL); } /* Target
Monitoring - Execution Log End */ #endif
```

## ExecutionLogFileFormat

The ExecutionLogFileFormat property is used to specify the format of the target monitoring log file, in terms of what data should be logged and in what order. The value of this property is related to the values of the properties that specify the instrumentation code that is added in order to write the information to the log file: ExecutionLogBeginCode and ExecutionLogEndCode.

```
Default = TRACE_DATA:TRACE_ID_NUM:%r,InstanceId:0x%p,BeginTime:%l,EndTime:%l
```

## ExecutionLogFileName

The ExecutionLogFileName property is used to specify the name to use for the target monitoring log file if you have set the value of the ExecutionLog property to Checked. A complete path should be specified for the file.

```
Default = ?<IsConditionalProperty>$(ExecutablePath)$(ExecutableName).log
```

## OnHostMessageReaderArguments

The OnHostMessageReaderArguments property specifies arguments to be passed to a .dll file that reads messages sent from a target to Rational Rhapsody when using target monitoring.

Target monitoring is an animation instrumentation mode in which instrumentation code is optimized and limited, and the user cannot control the execution of the application on the target, only to view its status.

For TcpMessageReader.Dll, the string consists only of a single integer indicating the port to listen on (if no changes were made to the TargetMonitor.c file, the default port is 24816).

This property is used in conjunction with the Animation::TargetMonitoring::TargetProtocolBuildFlag and Animation::TargetMonitoring::OnHostMessageReaderDLL properties.

The value is using a conditional property and covers the relevant values available in the Animation::TargetMonitoring::TargetProtocolBuildFlag property.

```
Default = com1:,9600,8,n,1
```

## OnHostMessageReaderDLL

The OnHostMessageReaderDLL property specifies the path to a .dll file that reads messages sent from a target to Rational Rhapsody when using target monitoring.

Target monitoring is an animation instrumentation mode in which instrumentation code is optimized and limited, and the user cannot control the execution of the application on the target, only to view its status.

The .dll file receives the messages from the target and extracts the information from them. The extraction method depends on the selected protocol (TCP, RS232, and so on).

This property is used in conjunction with the Animation::TargetMonitoring::TargetProtocolBuildFlag and Animation::Animation::TargetMonitoring::OnHostMessageReaderArguments properties.

The value is using a conditional property and covers the relevant values available in the Animation::TargetMonitoring::TargetProtocolBuildFlag property.

*Default = (\$OMROOT)\DLLs\SerialMessageReader.dll*

## TargetProtocolBuildFlag

The TargetProtocolBuildFlag property specifies the compilation flags with which the application should be compiled in order to use the relevant target monitoring protocol.

Target monitoring is an animation instrumentation mode in which instrumentation code is optimized and limited, and the user cannot control the execution of the application on the target, only to view its status.

The application has an additional module (TargetMonitor in the aom) that defines the protocol in which the data is sent from the target to Rational Rhapsody.

This property is used in conjunction with the Animation::TargetMonitoring::OnHostMessageReaderArguments and Animation::TargetMonitoring::OnHostMessageReaderDLL properties.

The possible values are:

- \_OM\_TARGET\_MON\_WINDOWS\_TCP
- \_OM\_TARGET\_MON\_LINUX\_TCP
- \_OM\_TARGET\_MON\_WINDOWS\_RS232
- \_OM\_TARGET\_MON\_HCS12X\_RS232
- \_OM\_TARGET\_MON\_STM32CIRCLE\_USB

*Default = \_OM\_TARGET\_MON\_HCS12X\_RS232, which is for the RS232 protocol and HCS12X target*

## UserFileFormat

The UserFileFormat property specifies the format of the output file created by target monitoring for



messages sent by using the `TGT_MON_USER_DATA_SEND()` macro. Each time the `TGT_MON_USER_DATA_SEND()` macro is used, an instance of the format (after substituting the relevant formatting tokens) is appended to the file.

The parameters used when calling the `TGT_MON_USER_DATA_SEND()` macro must match the formatting specifiers (see the following example). You can use the `TGT_MON_USER_DATA_SEND()` macro anywhere in the code and as many times as required.

The format contains free text and can contain the following type format specifiers. Available types for formatting:

- `%d` = int
- `%c` = char
- `%f` = float
- `%s` = `RiCString`
- `%r` = short
- `%l` = long
- `%p` = pointer

Example for a format: "Element: %p, intVal = %d"

Example for code corresponding to this format:

```
TGT_MON_USER_DATA_SEND(TGT_MON_SEND_PTR_PARAM(me);TGT_MON_SEND_INT_PARAM(me->idx));
```

*Default = Empty MultiLine*

The full path name of the file is defined in the `Animation::TargetMonitoring::UserFileName` property.

## UserFileName

The `UserFileName` property specifies the file name of the output file created by target monitoring for messages sent by using the `TGT_MON_USER_DATA_SEND(...)` macro. (The file format for the file is defined in the `Animation::TargetMonitoring::UserFileFormat` property.)

*Default = Empty string*

## UseTargetMonitoring

The `UseTargetMonitoring` property enables target monitoring.

Target monitoring is an animation instrumentation mode in which instrumentation code is optimized and limited, and the user cannot control the execution of the application on the target, only to view its status.

The possible values are:

- Off
- On

*Default = Off*

# ATL

---

The ATL subject contains metaclasses that contain properties that control aspects of ATL classes.

## Class

The Class metaclass contains properties that control the behavior of ATL classes.

### Aggregation

The Aggregation property enables or disables aggregation for a <<COM ATL Class>>. Note that this property must be set for the aggregated (part) class, not the aggregate (whole) class.

*Default = Yes*

### ConnectionPoints

The ConnectionPoints property specifies whether an ATL class supports a connection point.

*Default = No*

### DeclarationModifier

The DeclarationModifier property is a class modifier for an ATL class that gets printed before the class name.

*Default = ATL\_NO\_VTABLE*

### DeclareClassFactory

The DeclareClassFactory property specifies a template for the generation of code for ATL classes. For example:

```
class $DeclarationModifier $class : public CComObjectRootEx$ThreadModel, public  
CCOMCoClass$class, CLSID_def, public IDispatchImplIdef, IID_Idef, LIBID_ALL_KIND_OF_ATLLib  
{ ... }
```

This template references information that is stored in the ATLRootClass - the ATLClassObject and the IDispatchImpl properties.

*Default = Empty string*

## FreeThreadedMarshaller

The FreeThreadedMarshaller property specifies whether an ATL class supports a free-threaded marshaller.

*Default = No*

## SupportErrorInfo

The SupportErrorInfo property specifies whether an ATL class supports the ISupportErrorInfo interface.

*Default = No*

## ThreadingModel

The ThreadingModel property specifies the mapping of Rational Rhapsody threads (and any other UML modeling construct) and the COM apartment/threading model for the class under design. Because there is no direct mapping, you can freely define the apartment model for every generated ATL class.

*Default = Apartment*

# Configuration

The Configuration metaclass contains properties that control the configuration of ATL classes.

## APPID

The APPID property specifies the APPID name. If you do not specify the APPID property at the ComponentConfiguration level, Rational Rhapsody generates it automatically.

*Default = Empty string*

## ATLCustomCPFireOperation

The ATLCustomCPFireOperation property is a template for the implementation of connection point fire operations for custom interfaces.

```
Default = $opRetType Fire_$opname($arguments) { $opRetType ret; T* pT = static_cast<T*>(this); int
nConnectionIndex; int nConnections = m_vec.GetSize(); for (nConnectionIndex = 0; nConnectionIndex <
nConnections; nConnectionIndex++) { pT->Lock(); CComPtr<IUnknown> sp =
m_vec.GetAt(nConnectionIndex); pT->Unlock(); $interface* p$interface =
reinterpret_cast<$interface*>(sp.p); if (p$interface != NULL) { ret =
p$interface->$opname($argumentlist); } } return ret; }
```

## ATLDispInterfaceCPFireOperation

The ATLDispInterfaceCPFireOperation property is a template for the implementation of connection point fire operations for dual interfaces.

```
Default = $opRetType Fire_$opname($arguments) { CComVariant varResult; T* pT =
static_cast<T*>(this); int nConnectionIndex; CComVariant* pvars = NULL ; int noOfArgs = $noOfArgs
; if( noOfArgs > 0 ) { pvars = new CComVariant[noOfArgs]; } int nConnections = m_vec.GetSize(); for
(nConnectionIndex = 0; nConnectionIndex < nConnections; nConnectionIndex++) { pT->Lock();
CComPtr<IUnknown> sp = m_vec.GetAt(nConnectionIndex); pT->Unlock(); IDispatch* pDispatch =
reinterpret_cast<IDispatch*>(sp.p); if (pDispatch != NULL) { VariantClear(&varResult); /*Add your
code to initilize pvars[..]*/ DISPPARAMS disp = { pvars, NULL, $noOfArgs, 0 }; pDispatch->Invoke($id,
IID_NULL, LOCALE_USER_DEFAULT, DISPATCH_METHOD, &disp, &varResult, NULL, NULL); } }
delete[] pvars; return varResult.scode; }
```

## ATLProxyClass

The ATLProxyClass property provides a template for generating the ATL proxy class for a connection point.

```
Default = #ifndef CP$interface_H #define CP$interface_H $import template <class T> class
CProxy$interface : public IConnectionPointImpl<T, &$IDinterface, CComDynamicUnkArray> { public:
$operations }; #endif
```

## CPP\_StandardInclude

The CPP\_StandardInclude property adds standard include files in all CPP files.

```
Default = stdafx.h
```

## InProcServerExports

The InProcServerExports property provides a template for the DEF file used during DLL creation.

```
Default =
```

```
EXPORTS DllCanUnloadNow @1 PRIVATE DllGetClassObject @2 PRIVATE DllRegisterServer @3
PRIVATE DllUnregisterServer @4 PRIVATE
```

## InProcServerMainLineTemplate

The InProcServerMainLineTemplate property provides a template for the Dllmain() function.

```
Default =
```

```
if (dwReason == DLL_PROCESS_ATTACH) { _Module.Init(ObjectMap, hInstance/*,
```

```
&LIBID_$PackageLib*); DisableThreadLibraryCalls(hInstance); } else if (dwReason ==
DLL_PROCESS_DETACH) { _Module.Term(); } return TRUE; // ok
```

## InProcServerMainModule

The InProcServerMainModule property provides a template for the declaration and definition of each of the COM methods exported in the DLL. The default exported methods are DllCanUnloadNow(), DllGetClassObject(), IRegisterServer(), and DllUnregisterServer().

*Default =*

```
CComModule _Module; #ifdef _ATL_STATIC_REGISTRY #include <statreg.h> #if (_MSC_VER <
1310) // Avoid in .NET 2003 #include <statreg.cpp> #endif #endif #if (_MSC_VER < 1310) // Avoid in
.NET 2003 #include <atimpl.cpp> #endif static BOOL aFlag = TRUE ;
//////////////////////////////////// // Used to determine whether the DLL can be
unloaded by OLE STDAPI DllCanUnloadNow(void) { if( _Module.GetLockCount()==0 ) { OXF::end();
aFlag = TRUE ; return S_OK ; } else return S_FALSE ; }
//////////////////////////////////// // Returns a class factory to create an object of the
requested type STDAPI DllGetClassObject(REFCLSID rclsid, REFIID riid, LPVOID* ppv) { if(aFlag) {
if(OXF::init(0, NULL, 6423)) OXF::start(TRUE); aFlag = FALSE ; } return
_Module.GetClassObject(rclsid, riid, ppv); } ////////////////////////////////// //
DllRegisterServer - Adds entries to the system registry STDAPI DllRegisterServer(void) { // registers
object, typelib and all interfaces in typelib RegisterApp(COMPAPPID, "$component") ; return
_Module.RegisterServer($RegTlb); } ////////////////////////////////// //
DllUnregisterServer - Removes entries from the system registry STDAPI DllUnregisterServer(void) {
return _Module.UnregisterServer($RegTlb); }
```

## InProcStdAfx

The InProcStdAfx property is a template for the COM InProcServer StdAfx.h header file.

*Default =*

```
#if _MSC_VER > 1000 #pragma once #endif // _MSC_VER > 1000 #define STRICT #ifndef
_WIN32_WINNT #define _WIN32_WINNT 0x0400 #endif #define _ATL_APARTMENT_THREADED
#include <atlbase.h> //You might derive a class from CComModule and use it if you want to override
//something, but do not change the name of _Module extern CComModule _Module; #include <atlcom.h>
#include "RhapRegistry.h"
```

## OutProcServerMainLineTemplate

The OutProcServerMainLineTemplate property provides a template for the Dllmain() function.

*Default =*

```
_Module.StartMonitor(); #if _WIN32_WINNT >= 0x0400 & defined(_ATL_FREE_THREADED) hRes
= _Module.RegisterClassObjects(CLSCTX_LOCAL_SERVER, REGCLS_MULTIPLEUSE |
REGCLS_SUSPENDED); _ASSERT(SUCCEEDED(hRes)); hRes = CoResumeClassObjects(); #else
hRes = _Module.RegisterClassObjects(CLSCTX_LOCAL_SERVER, REGCLS_MULTIPLEUSE);
```

```
#endif _ASSERTE(SUCCEEDED(hRes)); MSG msg; while (GetMessage(&msg, 0, 0, 0))
DispatchMessage(&msg); _Module.RevokeClassObjects(); Sleep(dwPause); //wait for any threads to
finish _Module.Term(); CoUninitialize();
```

## OutProcServerMainModule

The OutProcServerMainModule property provides a template for the declaration and definition of each of the COM methods exported from the DLL. The default exported methods are DllCanUnloadNow(), DllGetClassObject(), IRegisterServer(), and IUnregisterServer().

*Default =*

```
#ifdef _ATL_STATIC_REGISTRY #include <statreg.h> #if (_MSC_VER < 1310) // Avoid in .NET 2003
#include <statreg.cpp> #endif #endif #if (_MSC_VER < 1310) // Avoid in .NET 2003 #include
<atlimpl.cpp> #endif const DWORD dwTimeOut = 5000; // time for EXE to be idle before shutting down
const DWORD dwPause = 1000; // time to wait for threads to finish up // Passed to CreateThread to
monitor the shutdown event static DWORD WINAPI MonitorProc(void* pv) { CExeModule* p =
(CExeModule*)pv; p->MonitorShutdown(); return 0; } LONG CExeModule::Unlock() { LONG l =
CComModule::Unlock(); if (l == 0) { bActivity = true; SetEvent(hEventShutdown); // tell monitor that we
transitioned to zero } return l; } //Monitors the shutdown event void CExeModule::MonitorShutdown() {
while (1) { WaitForSingleObject(hEventShutdown, INFINITE); DWORD dwWait=0; do { bActivity =
false; dwWait = WaitForSingleObject(hEventShutdown, dwTimeOut); } while (dwWait ==
WAIT_OBJECT_0); // timed out if (!bActivity && m_nLockCnt == 0) { // if no activity, shut down #if
_WIN32_WINNT >= 0x0400 & defined(_ATL_FREE_THREADED) CoSuspendClassObjects(); if
(!bActivity && m_nLockCnt == 0) #endif break; } } CloseHandle(hEventShutdown);
PostThreadMessage(dwThreadId, WM_QUIT, 0, 0); } bool CExeModule::StartMonitor() {
hEventShutdown = CreateEvent(NULL, false, false, NULL); if (hEventShutdown == NULL) { return
false; } DWORD dwThreadId; HANDLE h = CreateThread(NULL, 0, MonitorProc, this, 0,
&dwThreadId); return (h != NULL); } CExeModule _Module; LPCTSTR FindOneOf(LPCTSTR p1,
LPCTSTR p2) { while (p1 != NULL && *p1 != NULL) { LPCTSTR p = p2; while (p != NULL && *p
!= NULL) { if (*p1 == *p) { return CharNext(p1); } p = CharNext(p); } p1 = CharNext(p1); } return
NULL; }
```

## OutProcServerRegistration

The OutProcServerRegistration property provides a registration template for a COM OutProcServer server. This is inserted into the OutProcServer main file (in the WinMain(...) function) for server registration.

*Default =*

```
lpCmdLine = GetCommandLine(); //this line necessary for _ATL_MIN_CRT #if _WIN32_WINNT >=
0x0400 & defined(_ATL_FREE_THREADED) HRESULT hRes = CoInitializeEx(NULL,
COINIT_MULTITHREADED); #else HRESULT hRes = CoInitialize(NULL); #endif
_ASSERTE(SUCCEEDED(hRes)); _Module.Init(ObjectMap, hInstance ); /*, &LIBID_$package);*/
_Module.dwThreadId = GetCurrentThreadId(); TCHAR szTokens[] = _T("-/"); int nRet = 0; BOOL bRun
= TRUE; LPCTSTR lpszToken = FindOneOf(lpCmdLine, szTokens); while (lpszToken != NULL) { if
(lstrcmpi(lpszToken, _T("UnregServer"))==0) { nRet = _Module.UnregisterServer($RegTlb); bRun =
FALSE; break; } if (lstrcmpi(lpszToken, _T("RegServer"))==0) { nRet =
_Module.RegisterServer($RegTlb); RegisterApp(COMAPPID, "$component" ); bRun = FALSE; break;
} lpszToken = FindOneOf(lpszToken, szTokens); } if(!bRun) { _Module.Term(); CoUninitialize(); return
```

```
nRet; }
```

## OutProcStdAfx

The OutProcStdAfx property is a template for COM OutProcServer StdAfx.h header file.

*Default =*

```
#if _MSC_VER > 1000 #pragma once #endif // _MSC_VER > 1000 #define STRICT #ifndef
_WIN32_WINNT #define _WIN32_WINNT 0x0400 #endif #define _ATL_APARTMENT_THREADED
#include <atlbase.h> //You might derive a class from CComModule and use it if you want to override
//something, but do not change the name of _Module class CExeModule : public CComModule { public:
LONG Unlock(); DWORD dwThreadID; HANDLE hEventShutdown; void MonitorShutdown(); bool
StartMonitor(); bool bActivity; }; extern CExeModule _Module; #include <atlcom.h> #include
"RhapRegistry.h"
```

## ProxyStubExports

The ProxyStubExports property specifies the content of the ProxyStub.dll file. You can modify this content.

*Default =*

```
EXPORTS DllGetClassObject @1 PRIVATE DllCanUnloadNow @2 PRIVATE DllRegisterServer @3
PRIVATE DllUnregisterServer @4 PRIVATE
```

## RegistrationModule

The RegistrationModule property specifies the name of the file that implements the ATL class registration.

*Default = RhapRegistry*

## ServerNonCreatableObjectMapEntry

The ServerNonCreatableObjectMapEntry property is a macro that specifies a non-creatable ALT class entry into the ALT server map.

*Default =*

```
OBJECT_ENTRY_NON_CREATEABLE($class)
```

The \$class keyword is replaced with the name of the ATL class.

## ServerObjectMapBegin



The ServerObjectMapBegin property sets the ATL server map macro to "begin."

*Default = BEGIN\_OBJECT\_MAP(ObjectMap)*

## ServerObjectMapEnd

The ServerObjectMapEnd property sets the ATL server map macro to "end."

*Default = END\_OBJECT\_MAP()*

## ServerObjectMapEntry

The ServerObjectMapEntry property is a template that specifies a creatable ATL class entry into the ATL server map.

*Default = OBJECT\_ENTRY(CLSID\_\$coclass, \$class)*

The \$coclass keyword is replaced with the name of the coclass; \$class is replaced with the name of the ATL class that implements the coclass.

## TypeLibImportFormat

The TypeLibImportFormat specifies the template used to generate COM TLB import statements.

*Default = #import "\$tlbPath" raw\_interfaces\_only, raw\_native\_types, no\_namespace, named\_guids*

# Macro

The Macro metaclass contains properties that act as templates for ATL classes and operations.

## ATL\_ErrorMethodBody

The ATL\_ErrorMethodBody property is a template that implements the SupportErrorInfo operation.

*Default =*

```
static const IID* arr[] = { $IID_implClass }; for (int i=0; i < sizeof(arr) / sizeof(arr[0]); i++) { if  
(InlineIsEqualGUID(*arr[i],riid)) { return S_OK; } } return S_FALSE;
```

## ATL\_FTMCreateBody

The ATL\_FTMCreateBody property is a template that causes a function in an ATL class to create a

free-threaded marshaller.

*Default = return CoCreateFreeThreadedMarshaler(GetControllingUnknown(), &m\_pUnkMarshaler.p);*

## **ATL\_FTMReleaseBody**

The ATL\_FTMReleaseBody property is a template that causes a function in an ATL class to release a free-threaded marshaller.

*Default = m\_pUnkMarshaler.Release();*

## **ATLClassObject**

The ATLClassObject property specifies the ATL class that implements a COM coclass.

*Default = CComCoClass<\$class, &CLSID\_\$coclass>*

The \$class keyword is replaced with the name of the ATL class that implements the coclass; \$coclass is replaced with the name of the coclass that exposes the COM interface.

## **ATLConnectionPointImpl**

The ATLConnectionPointImpl property specifies the ATL class that implements the IConnectionPointContainer interface.

*Default = IConnectionPointContainerImpl<\$interface>*

The \$interface keyword is replaced with the name of the interface being implemented.

## **ATLRootClass**

The ATLRootClass property specifies the ATL root class.

*Default = CComObjectRootEx<\$ThreadModel>*

*The \$ThreadModel keyword is replaced with the value of the ThreadingModel property (Default = Apartment).*

## **BeginConnectionPointMap**

The BeginConnectionPointMap property specifies the macro to start a connection point map for an ATL class.

*Default = BEGIN\_CONNECTION\_POINT\_MAP(\$interface)*

The \$interface keyword is replaced with the interface that contains the connection point map.

## BeginInterfaceMap

The BeginInterfaceMap property controls how macro templates are generated. The property specifies the start macro for a COM map of an ATL class.

*Default = BEGIN\_COM\_MAP(\$class)*

The \$class keyword is replaced with the name of the ATL class.

## ClassRegistration

The ClassRegistration property specifies the ATL class registration macro.

*Default =*

```
DECLARE_RHAPSODY_REGISTER(CLSID_$coclass, "$TypeName",  
"$VersionIndepProgID", "$ProgID", "$ThreadingModel", COMPAPPID )
```

The keywords are replaced with the appropriate information, as follows:

- \$coclass - Replaced with the name of the coclass that the ATL class implements.
- \$TypeName - Replaced with the value of the TypeName property, which specifies the declaration of the class type being registered (Default = \$class).
- \$VersionIndepProgID - Replaced with the value of the VersionIndepProgID property (Default = \$component.\$class).
- \$ProgID - Replaced with the value of the ProgID property (Default = \$component.\$class.1).
- \$ThreadModel - Replaced with the value of the ThreadingModel property (Default = Apartment).

## ConnectionPointMapEntry

The ConnectionPointMapEntry property specifies the macro for the connection point map entry.

*Default = CONNECTION\_POINT\_ENTRY(\$interface)*

The \$interface keyword is replaced with the interface that contains the connection point map.

## ConnectionPointProxyClass

The ConnectionPointProxyClass property specifies the proxy class for the connection point.

*Default = CProxy\$interface< \$class >*

The \$interface keyword is replaced with the name of the interface being implemented; \$class is replaced with the ATL class.

## DeclareControllingUnknown

The DeclareControllingUnknown property prints the DECLARE\_GET\_CONTROLLING\_UNKNOWN() macro into the ATL class. For more information on COM properties, see the MSDN Online Library.

*Default = DECLARE\_GET\_CONTROLLING\_UNKNOWN().*

## DeclareProtect

The DeclareProtect property specifies the macro that protects the ATL object from being deleted if, during FinalConstruct(), the nested object increments the reference count and then decrements the count to 0.

*Default = DECLARE\_PROTECT\_FINAL\_CONSTRUCT()*

## EndConnectionPointMap

The EndConnectionPointMap property specifies the macro to end a connection point map for an ATL class.

*Default = END\_CONNECTION\_POINT\_MAP()*

## EndInterfaceMap

The EndInterfaceMap property specifies the end macro for a COM map of an ATL class.

*Default = END\_COM\_MAP()*

## IDispatchImpl

The IDispatchImpl property provides support for animation.

*Default = IDispatchImpl<\$interface, &IID\_\$interface, &LIBID\_\$Package>*

The \$interface keyword is replaced with the name of the interface being animated; \$Package is replaced with the name of the COM library to which the interface belongs.

## InterfaceEntry

The InterfaceEntry property specifies the ATL macro that defines the COM map interface entry point.

*Default = COM\_INTERFACE\_ENTRY(\$interface)*

The \$interface keyword is replaced with the name of the interface being animated.

## InterfaceEntry2

The InterfaceEntry2 property specifies the ATL macro that defines the COM map interface entry point, to disambiguate two branches of inheritance.

*Default = COM\_INTERFACE\_ENTRY2(\$dupinterface,\$interface)*

The \$dupinterface keyword is replaced with the name of the duplicate interface; \$interface is replaced with the name of the interface being animated.

## InterfaceEntryAggr

The InterfaceEntryAggr property is a macro that enables an interface entry of an aggregated object in an ALT class interface map.

*Default = COM\_INTERFACE\_ENTRY\_AGGREGATE(IID\_\$interface, \$datamem)*

The \$interface keyword is replaced with the name of the interface being animated; \$datamem is replaced with the data member.

## NoAggregation

The NoAggregation property is a template for a macro that specifies that an object cannot be aggregated.

*Default = DECLARE\_NOT\_AGGREGATABLE(\$class)*

The \$class keyword is replaced with the name of the ATL class.

## OnlyAggregation

The OnlyAggregation property is a template for a macro that specifies that an object must be aggregated.

*Default = DECLARE\_ONLY\_AGGREGATABLE(\$class)*

The \$class keyword is replaced with the name of the ATL class.

## ProgID

The ProgID property is a standard MSDN COM property. This property returns the programmatic identifier (ProgID) for the specified OLE object. For more information on COM properties, see the MSDN Online Library (<http://msdn.microsoft.com/library/>).

*Default = \$component.\$class.I*

The \$component keyword is replaced with the name of the component; \$class is replaced with the name of the ATL class.

## ReturnSuccess

The ReturnSuccess property is a macro that specifies a successful return of an ATL class standard operation.

*Default = return S\_OK;*

## SupportAggregation

The SupportAggregation property is a template for a macro, which specifies that an object can be aggregated.

*Default = DECLARE\_AGGREGATABLE(\$class)*

The \$class keyword is replaced with the name of the ATL class.

## TypeName

The TypeName property specifies the declaration of the class type being registered, when you use the ClassRegistration property to specify the ATL class registration macro.

*Default = \$class class*

The \$class keyword is replaced with the name of the ATL class.

## VersionIndepProgID

The VersionIndepProgID property specifies the version-independent ID when you use the ClassRegistration property to specify the ATL class registration macro.

*Default = \$component.\$class*

The \$component keyword is replaced with the name of the component; \$class is replaced with the name of the ATL class.

## Operation

The Operation metaclass contains a property that specifies a standard ATL class operation.

## STDMETHOD

The STDMETHOD property specifies a standard ATL class operation.

*Default = Cleared*

# AutomaticTestGenerator

---

The AutomaticTestGenerator subject contains metaclasses that contain properties that affect the Automatic Test Generator (ATG) tool. This subject is available only if you have installed ATG.

## Settings

The Settings metaclass contains several properties with options for ATG.

### DisableTypeApplicabilityCheckInTracedMessages

The DisableTypeApplicabilityCheckInTracedMessages property controls if ATG applies applicability checks for types to be used in the interface.

If the property is checked, messages with arguments or return values of any type can be selected for trace generation. Any argument or return value with a not supported type will be recorded as don't care value in the generated traces.

If the property is cleared, messages with arguments or return values of types not supported by ATG cannot be selected for trace generation.

*Default = Cleared*

### ShowCallsOfPortRelayOperationsInScenarios

The ShowCallsOfPortRelayOperationsInScenarios property controls if calls of functions relaying messages via ports shall be visible in exported scenarios or not.

If the property is checked, calls of helper functions used for relaying messages via ports will be drawn into scenarios exported by ATG.

If the property is cleared, calls of helper functions used for relaying messages via ports will not be drawn into scenarios exported by ATG.

*Default = Cleared*

### StimulationInterfaceFilterMode

The StimulationInterfaceFilterMode property controls the list of classes offered for selection of the interface for trace generation.

If the value of the property is set to Parts, ATG filters the list of classes displayed for selection of available interfaces and shows only classes which are instantiated by the test context.



If the value of the property is set to None, ATG applies no filtering for the list of classes available for the interface selection.

*Default = Parts*

## **StimulationInterfaceMode**

The StimulationInterfaceMode property controls if the System Under Test shall be stimulated via ports or directly.

If the value of the property is set to PortBased, ATG automatically creates port relay functions in the test architecture for port interfaces available in the SUT. The helper functions will be used for stimulating the SUT via ports in the generated traces.

If the value of the property is set to Direct, ATG generates traces with direct stimulation of the SUT even if the SUT has port interfaces.

*Default = PortBased*

# AUTOSAR

---

The AUTOSAR subject contains metaclasses that contain properties that support Automotive Open System Architecture (AUTOSAR) in Rational Rhapsody.

## ARXML

The ARXML metaclass contains properties that support Automotive Open System Architecture (AUTOSAR) in Rational Rhapsody.

### GloballyMatchUuid

Prior to release 8.1.4, when importing ARXML files into Rhapsody, the UUID-matching mechanism looked for model elements with matching UUIDs only in the current hierarchy. Beginning in 8.1.4, the import process searches the entire project for matching UUIDs. If you want to use the pre-8.1.4 behavior, set the value of the property GloballyMatchUuid to False.

### TimerTickPeriod

Use the TimerTickPeriod property to set the time scaling multiplier for the "Period" values of a periodic Rational Rhapsody Implementation Block (RIMB) or an ActiveOperation. Time scaling means the ratio between one time unit in the model and one time unit in real-time (on a target). For example, in Rational Rhapsody 1 typically means 1 millisecond. If you would rather that 1 means seconds, then you can enter a value of 0.001 in the TimerTickPeriod property as the multiplier, so that 1 means 1 second for your model.

*Default = 0.001*

### UseModelOrderOnExportForMetaclasses

You can use the property OverrideOrdinalForMetaclasses to specify a list of metaclasses that should be exported to ARXML according to their current order in the model, rather than their relative order at the time they were imported into the model.

The value of OverrideOrdinalForMetaclasses should be a comma-separated list of metaclass names. The list can include both out-of-the-box metaclasses and "new terms".

This property should be set at the configuration level.

*Default = Argument, subElement, ImplementationDataTypeElement*

## CodeGeneration

The CodeGeneration metaclass contains properties relating to the use of the RIMB implementation to generate code for AUTOSAR software components.

## GenerateToSingleFileName

When using the RIMB implementation to generate code for an AUTOSAR software component, the property GenerateToSingleFileName can be used to customize the name used for the generated .c and .h files.

By default, the name of the software component is used.

See also the property: AUTOSAR::CodeGeneration::GenerateToSingleFileNameStyle.

*Default = \$<swcName>*

## GenerateToSingleFileNameStyle

When using the RIMB implementation to generate code for an AUTOSAR software component, the property GenerateToSingleFileName determines the name used for the generated .c and .h files.

You can further customize the names of the generated files by using the property GenerateToSingleFileNameStyle to specify that the filenames should be converted to all upper case or all lower case characters.

The possible values of this property are:

- NoChange - take the name directly from the property GenerateToSingleFileName.
- ToUpper - take the name from the property GenerateToSingleFileName and convert it to all upper case characters
- ToLower - take the name from the property GenerateToSingleFileName and convert it to all lower case characters

*Default = NoChange*

## RTE\_API

The RTE\_API metaclass contains properties that are used by the generation utility for the Automotive Open System Architecture Runtime Environment application programming interface (AUTOSAR RTE API).

## CandidatesNames

The CandidatesNames property contains a list of metaclasses to be handled by the generation utility for the Automotive Open System Architecture Runtime Environment application programming interface (AUTOSAR RTE API).

*Default =*  
*DataReadAccess,DataReceivePoint,DataSendPoint,DataWriteAccess,InterRunnableVariable,SynchronousServerCallPoint,A*

## ComponentsNames

The ComponentsNames property contains a list of metaclasses that represent Automotive Open System Architecture (AUTOSAR) Component types.

*Default =*  
*ApplicationSoftwareComponentType,ServiceComponentType,SensorActuatorSoftwareComponentType,ComplexDeviceDrive*

## GenerateRTEContractAPIs

This property controls the way AutosarRTEContract is generated.

Possible values:

- All - RTE contract is generated for all elements in the current configuration.
- PerUsage - RTE contract is generated Only for elements that are actually in use.

## Tokens

The Tokens property contains an expression that defines the templates to generate the various Automotive Open System Architecture Runtime Environment application programming interfaces (AUTOSAR RTE APIs).

*Default =*  
*<d>:dependency:l\_dataElement:nameDepandsOn~<o>:dependency:l\_operation:nameDepandsOn~<re>:owner:null:nameDepandsOn~<data>:dependency:l\_readVariable:type+nameDepandsOn~<p>:dependency:l\_port:nameDepandsOn*

# Browser

---

The Browser subject contains metaclasses that contain properties that enable you to modify the display of the Rational Rhapsody browser.

## Operation

The metaclass Operation contains properties that are used to control the way operations are displayed in the browser.

### ShowReturnTypeFromCG

The ShowReturnTypeFromCG property is used to specify that the signatures displayed for operations in the browser should include the return type that is generated in the code.

This property does not affect the browser display if you are working in Label mode.

*Default = Cleared*

## Settings

The Settings metaclass contains properties that control the display of the Rational Rhapsody browser.

### DeleteConfirmation

The DeleteConfirmation property specifies whether confirmation is required before deleting a graphical element from the model. Note that this property does not apply to statechart elements, which have a separate DeleteConfirmation property. The possible values are as follows:

- Always - Rational Rhapsody displays a confirmation window each time you try to delete an item from the model.
- Never - Confirmation is not required to delete an element.
- WhenNeeded - Rational Rhapsody displays a message asking for confirmation if there are references to the element (or for some other reason).

*Default = Always*

### DisplayMode

The DisplayMode property specifies the mode used to display the browser tree. The possible values are as follows:

- Meta-class - Display all the metaclass nodes (such as operations and objects).
- Flat - Do not display metaclass nodes.

*Default = Meta-class*

## **PreserveTreeNodeExpandState**

PreserveTreeNodeExpandState property allows you to specify whether or not Rational Rhapsody should remember the open or closed state of nested packages/folders in the browser for the duration of a Rational Rhapsody session.

To have Rational Rhapsody keep track of this information, set the value of the property to True.

*Default = Checked*

## **ShowAttributesCodeAsTooltip**

When you move your mouse over an attribute in the browser, a tooltip is displayed, showing the code that is generated for the attribute declaration. The ShowAttributes property is a Boolean property that CodeAsTooltip allows you to turn this behavior off or on.

*Default = Checked*

## **ShowContextForAssociation**

ShowContextForAssociation that allows you specify that for association ends, Rational Rhapsody should display the destination of the association in parentheses alongside the name of the association end in the browser.

This is particularly useful for situations where users might change the names of association ends, which by default refer to the destination (for example, itsClass\_3).

*Default = False*

## **ShowFeatures**

Reserved for future use.

*Default = Checked*

## **ShowImplementationArgument**

The ShowImplementationArgument property controls the way that operation arguments are displayed in the browser.

Ordinarily, the browser displays just the argument type and name.

However, if you change the value of this property to True, the browser displays the exact code that is generated for the arguments, for example, "getData(const Vehicle& currentVehicle)" instead of "getData(Vehicle currentVehicle)".

*Default = Cleared*

## ShowImplementationNameInTree

The ShowImplementationNameInTree property specifies whether to display the implementation (generated) name of operations instead of the design (user-assigned) name in the browser tree. The default is False (the design name is displayed). For example, in Rational Rhapsody Developer for C, if you create an operation named open() for an object named Valve, the design name for the operation is open(), but its implementation name is actually Valve\_open(). You must always use the implementation name for operations (and states) anywhere you write code. You can toggle the display of operation names in the browser tree between implementation names and design names by changing this property and then closing and reopening the object node to refresh the display of the operation names. Note that the implementation name is always displayed in the Operation window on the right side of the browser, regardless of this property setting.

*Default = Cleared*

## ShowLabels

The ShowLabels property specifies whether to display labels instead of names in the browser or diagrams, depending on which property is set.

*Default = Cleared*

## ShowMultipleStereotypes

Model elements can have one or more stereotypes applied to them. You can use the ShowMultipleStereotypes property to specify whether all of the applied stereotypes should be displayed alongside the name of the element in the model browser, or just the first stereotype.

*Default = True*

## ShowOrder

The ShowOrder property enables or disables the ability to reorder elements in the browser by enabling/disabling the up/down arrow controls. When the user selects View > Browser Display Options > Enable Ordering from the main menu, the property is assigned the Checked value. When the user deselects the Enable Ordering menu item, the property is assigned the Cleared value.

## ShowPredefinedPackage

The ShowPredefinedPackage property determines whether the PredefinedTypes package is displayed in the browser. When the property is set to Cleared, the package is hidden.

*Default = Checked*

## ShowSourceArtifacts

When you use reverse engineering and/or roundtripping, certain information that is necessary for the Rational Rhapsody code preserving feature is stored as SourceArtifact elements.

By default, these SourceArtifact elements are not displayed in the browser.

If the ShowSourceArtifacts property is set to True, then these elements are opened in the browser.

This property corresponds to the menu option View > Browser Display Options > Show Source Artifacts.

*Default = Cleared*

## ShowStereotypes

The ShowStereotypes property determines whether the browser displays the stereotypes applied to a model element, alongside the name of the element. It also controls how the stereotypes should be displayed. The possible values for this property are:

- No - stereotypes are not displayed
- Prefix - stereotypes are displayed to the left of the element name
- Suffix - stereotypes are displayed to the right of the element name

*Default = Prefix*

The property is set at the project level.

When you select View > Browser Display Options > Show Stereotype from the main menu, and select one of the stereotype display options, the value of this property is updated accordingly.

Note that the property Browser::Settings::ShowMultipleStereotypes can be used to specify whether all the applied stereotypes should be displayed or only the element's first stereotype.

## ShowTagValueFormat

Use the ShowTagValueFormat property to control the information that is displayed for tags in the model browser. The property supports the following keywords: \$name, \$type, \$value.

*Default = \$name = \$value*

## SortingPolicy

Prior to version 8.0.3, if you were using the Flat browser view (meaning, no category names such as Classes or Events), the model elements were displayed alphabetically. Beginning in version 8.0.3, when you use the Flat view, the elements are organized by metaclass, and alphabetically within each metaclass.



This behavior is controlled by the `SortingPolicy` property. If you want to restore the previous behavior, you can change the value of `SortingPolicy` to "ByBrowserSettings." Note that this property does not affect the display of elements if you have selected the `Enable Ordering` option.

*Default = MetaClass>NewTerm>Name*

The CG subject contains metaclasses that contain properties that control code generation aspects that are common to all languages:

## Argument

The Argument metaclass contains a property that controls the animation of a specific argument.

### Animate

The Animate property specifies whether animation code is generated for an element. You can specify your own animation function by using the `CPP_CG::Type::AnimSerializeOperation` property.

The semantics of the Animate property is always in favor of the owner settings:

- If a package Animate property check box is cleared, all the classes owned by the package are not animated, regardless of the class Animate settings.
- If a class Animate property check box is cleared, all the elements in the class (attributes, operations, relations, and so on) are not animated.
- If an operation Animate property check box is cleared, all the arguments are not animated.
- If the AnimateArguments property check box is cleared, all the arguments are not animated, regardless of the specific argument Animate property settings. (Default = Checked)

### UsageType

The UsageType property determines how an `#include` statement is generated for a type used as an argument of an operation. The possible values are:

- Existence - A forward declaration is generated in the specification file
- Implementation - A forward declaration is generated in the specification file, and an `#include` statement is generated in the implementation file
- Specification - An `#include` statement is generated in the specification file
- None - No `#include` statements or forward declarations are generated

The value of the property can be set at the level of individual arguments or higher (operation, class, package, project).

Note that for types, inner classes, and templates, an `#include` statement is always generated in the specification file unless the value of the property is set to None.

*Default = Implementation*

# Attribute

The Attribute metaclass contains properties for implementing attributes and methods that handle attributes.

## Accessor

The Accessor property specifies the format of the names of attribute accessors. The string `get_$attribute` means that if an accessor is generated for an attribute, it is called `get_attributeName`.

If you would like to shorten the length of the names of the accessors generated, you can specify the number of characters that should be taken from the name of the attribute. For example, if you set the value of the property to `get$attribute:3` and your model contains an attribute named "salary", the name of the accessor generated will be `getSal()`.

*Default = `get$attribute:c`*

## Animate

The Animate property specifies whether animation code is generated for an element. You can specify your own animation function by using the `CPP_CG::Type::AnimSerializeOperation` property.

The semantics of the Animate property is always in favor of the owner settings:

- If a package Animate property check box is cleared, all the classes owned by the package are not animated, regardless of the class Animate settings.
- If a class Animate property check box is cleared, all the elements in the class (attributes, operations, relations, and so on) are not animated.
- If an operation Animate property check box is cleared, all the arguments are not animated.
- If the AnimateArguments property check box is cleared, all the arguments are not animated, regardless of the specific argument Animate property settings.

*Default = `Checked`*

## AnimateAttributes

The AnimateAttributes property specifies whether to animate class member attributes in the animated browser.

During instrumentation, Rational Rhapsody displays the values of class member attributes in the animated browser by using the C++ stream output operator.

For this to work, the attribute must be of a built-in type that can be output to a stream, such as an `int` or a `char*`.

If, however, the attribute is of a complex, user-defined type that the compiler cannot serialize, trying to animate it causes compilation errors.

In VxWorks, trying to animate attributes of type long unsigned int also causes errors. If you want to animate attributes of user-defined types defined either inside or outside of Rational Rhapsody, you must do one of the following:

- Add to the framework an overloaded C++ stream output operator for the type, such as:

```
ostream operator(sometype);
```

- Instantiate the template string2X(T t) with the type.
- Disable the AnimateAttributes property by setting it to Cleared.

*Default = Checked*

## CorbaRealizingAccessor

The CorbaRealizingAccessor property is a format string that specifies the naming convention of an attribute getter that realizes a CORBAInterface attribute.

*Default = \$attribute*

## CorbaRealizingMutator

The CorbaRealizingMutator property is a format string that specifies the naming convention of an attribute setter that realizes a CORBAInterface attribute.

*Default = \$attribute*

## DataReceptionProcessingPolicy

The DataReceptionProcessingPolicy property defines the data reception processing policy for attributes.

- Disabled - No active operation will be generated.
- Immediate - The attribute will be handled immediately, meaning a triggered operation will be associated with it.
- Queued - The attribute handling will be queued, meaning a reception will be associated with it and an event will be generated (RiCGEN) and queued into the event queue for the Rational Rhapsody implementation block (RIMB).
- ByModel - The attribute will be handled according to the model. If there is a trigger operation associated with the attribute, processing happens immediately. If there is a reception associated with the attribute, it is added to a queue.

Note: Associating a trigger operation or a reception to an attribute is done by name: ev<p>\_<x> will match an attribute by the name <p>\_<x>.

*Default = ByModel*

## Generate

The Generate property specifies whether to generate code for a particular type of element.

*Default = Checked*

## Implementation

The Implementation property specifies how Rational Rhapsody should generate code for attributes or relations.

The various container property subjects, such as OMContainers, contain properties for generating code related to attributes and relations. When the value of the Implementation property is set to Default, Rational Rhapsody determines what category of properties to use (for example, Scalar or BoundedUnordered) on the basis of a number of criteria, including:

- the multiplicity specified
- whether or not the attribute was defined as ordered
- whether or not the attribute is of a predefined type

For example, if multiplicity is set to \* or 1..\*, code generation for the attribute or relation will be based on the values of the properties contained in the UnboundedUnordered metaclass. If the Ordered check box is also selected, then Rational Rhapsody will use the values of the properties in the UnboundedOrdered metaclass.

If you would prefer to manually specify what category of properties Rational Rhapsody should use, you can do so by selecting one of the other possible values for the Implementation property, for example, BoundedOrdered or Scalar.

The possible values for this property are:

- Default - Rational Rhapsody determines what category of properties to use
- Scalar - Implement a to-one relation as a pointer to a single object
- StaticArray - Implement a to-many relation as a static array of fixed size
- Fixed - Implement a to-many relation whose multiplicity is a number greater than 1, as a container of fixed size
- BoundedOrdered - Implement a to-many relation whose multiplicity is a number greater than 1 and whose "Ordered" check box is selected, as a list
- BoundedUnordered - Implement a to-many relation whose multiplicity is a number greater than 1 and whose "Ordered" check box is not selected, as a collection
- UnboundedOrdered - Implement a to-many relation with an upper limit of \* and whose "Ordered" check box is selected, as a list
- UnboundedUnordered - Implement a to-many relation with an upper limit of \* and whose "Ordered" check box is not selected, as either a list or a collection, depending on the language
- EmbeddedScalar - Implement a to-one composite relation as an embedded object (C and C++ only)
- EmbeddedFixed - Implement a to-many composite relation as an array of embedded objects (C and C++ only)

- User - This option instructs Rational Rhapsody to use the values you have provided for the properties under the "User" metaclass for the property subject that is relevant for you, for example OMContainers
- Qualified (for relations only) - Implement a to-many qualified relation as a map or hashtable, depending on the language

*Default = "Default"*

## IsConst

IsConst refers to the getter of the attribute (it does not indicate whether or not the attribute itself is a constant).

This property can take one of the following values:

Signature - This value means that the getter function is constant. For an int attribute, the generated code would be:

```
int getHeight() const;
```

Since this is the default behavior for attributes in C/C++, this is also the default value for the property.

None - This value means that you are allowing the attribute value to be changed in the getter before it is returned.

SignatureAndReturnValue - This value means that not only is the getter function constant, but also the object returned to the calling function is constant. The generated code would be:

```
const class_2* getInfo() const;
```

This value modifies the generated code if the attribute is a class, but for primitive types the generated code would be the same as for the value "Signature".

*Default = Signature*

## IsGuarded

The IsGuarded property specifies whether accessor and mutator operations are guarded. Guarded operations block if the object is not in a state in which it can be executed.

The possible values are as follows:

- none - Neither accessors nor mutators are guarded.
- mutator - Only mutators are guarded.
- all - Both accessors and mutators are guarded.

*Default = none*

## IterType

When get/set functions are generated for an attribute with multiplicity > 1 (array), they take the array index as a parameter. By default, the software generates an index of type int.

The IterType property allows you to specify a different type for the array index.

To use a different type, just enter the name of the type as the value for this property.

This property can be set for individual attributes, or at higher levels, such as class, package, or project.

*Default = Blank*

## Mutator

The Mutator property specifies the format of the names of mutator operations generated for attributes.

The default string, "set\_\$attribute:c", means that if a mutator is generated for an attribute, the generated method name is set\_attributeName().

*Default = set\_\$attribute:c*

## PeriodicRead

The PeriodicRead property specifies if an attribute should be read periodically or when a data-received-event activation is received.

Use this property in conjunction with CG::Attribute::ReadInterval.

*Default = Cleared*

## ReadInterval

The ReadInterval property specifies the interval between reads of an attribute.

Use this property in conjunction with CG::Attribute::PeriodicRead.

# CGGeneral

The CGGeneral metaclass contains a property that controls canonical operations.

## GeneratedCodeInBrowser

The GeneratedCodeInBrowser property specifies whether canonical operations (get/set) are added to the model and displayed in the browser. The possible values are as follows:

- Checked - Display automatically generated operations in the browser tree.
- Cleared - Do not display canonical operations.

*(Default = Cleared)*

## IgnoreGuardedOperationVisibility

The IgnoreGuardedOperationVisibility property is a Boolean value that specifies whether the visibility of a guarded operation is ignored. This property is available at the project level.

This property was added for upgrade reasons only. It should not be used directly, and should not exist in models created with Rational Rhapsody version 4.0 and higher.

*(Default = Checked)*

## InstallLayoutAs2.3

The InstallLayoutAs2.3 property specifies whether to use the generated file layout from Rational Rhapsody version 2.3.

Set this property at the project level.

*(Default = Checked)*

## Class

The Class metaclass contains properties for implementing classes and objects.

## ActiveMessageQueueSize

The ActiveMessageQueueSize property specifies the size of the message queue allocated for active classes. The possible values are as follows:

- Any integer - Specifies the message queue size for an active class.
- An empty string (blank) - If not specified, the value is set in an operating system-specific manner, based on the value of the ActiveMessageQueueSize property for the framework.

*Default = Empty string*

## ActiveStackSize

The ActiveStackSize property specifies the size of the stack allocated for active objects. The possible values are as follows:



- Any integer - Specifies that a stack of that size is allocated for active objects.
- An empty string (blank) - If not specified, the stack size is set in an operating system-specific manner, based on the value of the `ActiveStackSize` property for the framework.

*(Default = empty string)*

## ActiveThreadName

The `ActiveThreadName` property indicates the real OS task or thread name. This property has an affect only when the class is set to active. This facilitates debugging in complex environments in which many threads are constantly being created and deleted on-the-fly. This property is effective for all targets. All strings entered must be enclosed in quotation marks (" ").

The possible values are as follows:

- A string - Names the active thread.
- An empty string (blank) - If not specified, the value is set in an operating system-specific manner, based on the value of the `ActiveThreadName` property for the framework.

*Default = empty string (OS selects thread name)*

## ActiveThreadPriority

The `ActiveThreadPriority` property specifies the priority of active class threads.

The value of the property should be an integer or the name of a variable or constant that represents an integer value.

If left blank, the priority will be taken from the value of the `CPP_CG::Framework::ActiveThreadPriority` property.

*Default = Blank*

## AdditionalCleanupCode

*The `AdditionalCleanupCode` property is a `MultiLine` value that specifies additional cleanup code. (Default = empty `MultiLine`)*

## AdditionalInitializationCode

*The `AdditionalInitializationCode` property is a multiline value that specifies additional initialization code. (Default = empty `MultiLine`)*

## AttributesAutoArrange

*The `AttributesAutoArrange` property is a Boolean UI helper property. You should not modify this property directly. (Default = Checked)*

## CallUserInitRelations

The CallUserInitRelations property is a Boolean value that determines whether to call a user-defined initRelations() method. In Rational Rhapsody, initRelations() is called by the generated code (in the class constructors) even if you created your own initRelations() method.

Set this property to Cleared to disable the call to the user-defined initRelations().

*(Default = Checked)*

## ComplexityForInlining

The ComplexityForInlining property provides you with a certain degree of control over the degree to which Rhapsody uses inlining in place of function calls in the code generated for statecharts.

The property takes integer values. The higher you make the value, the more Rhapsody uses inlining. If you set the value to 0, Rhapsody never uses inlining in place of function calls in the code generated.

Increasing the value results in increased code size but can result in shorter code execution time.

This property applies only to the Flat implementation scheme for statecharts.

*Default = 3*

## Concurrency

The Concurrency property specifies the concurrency of a class.

The CG::Operation::Concurrency property lets you specify that an operation should be protected by the object mutex (that is, only one access to the operation is allowed at any specific time). When a class has one or more operations with concurrency guarded, the class becomes a guarded class.

The possible values for CG::Class::Concurrency are as follows:

- sequential - A sequential class maintains the single-threaded sequential protocol.
- active - An active class creates its own thread around which its operations are executed.

The possible values for CG::Operation::Concurrency are as follows:

- sequential - The operation is not guarded; access is sequential.
- guarded - The operation is protected by the object mutex.

Note that the Concurrency property cannot be used to guard static operations or the constructor of a class.

*Default = sequential*

## CreateImplicitDependencies

When the property `CreateImplicitDependencies` is set to `True`, Rational Rhapsody analyzes the types and classes used as types for the attributes and relations in the class, or as arguments or return types of operations in the class, and generates the required `#include` directives or forward declarations (or "import" statements for Java code).

*Default = True*

## DeleteGlobalInstance

The `DeleteGlobalInstance` property specifies whether to delete a global instance of the class. Global instances are not deleted by default because reaching a termination connector for a deleted global instance could cause an unexpected shut down.

*(Default = Cleared)*

## EmptyMemoryPoolCallback

The `EmptyMemoryPoolCallback` property specifies a name for the callback function that allocates more memory if the static pool is exhausted. This property provides support for static architectures.

*(Default = empty string)*

## EmptyMemoryPoolMessage

*The `EmptyMemoryPoolMessage` property specifies whether a message is output when the static memory pool is empty. This property provides support for static architectures. (Default = Checked)*

Note: This property is only active during animation.

## FileName

The `FileName` property specifies the name of the file to which code is generated for a class or package. This property uses a different name than the actual element name. For example, this feature can be of benefit if the class or package name is too long on 8.3 file systems.

The file name string can be either:

- A file name of your choice (including spaces)
- `$name:n`, where `n` is any digit between 0 and 9

The variable `$name:n` uses the name of the element as the basis for generating file names. The `:n` modifier specifies the length of the name. For example, if you specify `$name:8`, a class named `YosemiteNationalPark` would be generated into the files `Yosemite.cpp` and `Yosemite.h`. If you use `$name:0`, then no package file is generated (file name length = 0). Note also that when the `CPP_CG::Package::GenerateDirectory` property is set to `Cleared`, then no additional package directory is generated.

If the `FileName` property is blank and the `CG::Environment::IsFileNameShort` property is set to `Checked`,

the code generator creates 8.3 file names by truncating the class name.

If the FileName property is defined as a file name longer than eight characters and the IsFileNameShort property is set to Checked, the Checker reports an error. The long file name must be fixed prior to code generation. Each file name must be unique within the context of a single configuration.

If more than one class or package are generated to the same file name, they overwrite each other, causing compilation errors. To prevent this, the Checker determines whether multiple packages are directed to the same file before code is generated.

Setting the FileName property for an external base class (for example, to "BaseClass" without the ".h") generates an #include of the base class in the specification file of the subclass.

You use the UseAsExternal property to mark a class as an external reference class.

If the FileName property is not defined, you must add the appropriate code to the SpecificationProlog property for the subclass.

*Default = Empty string*

## **ForceInDeclarationForNestedTemplate**

Prior to version 8.0.3 of Rational Rhapsody, when code was generated for nested templates, inline operations would be generated in the declaration regardless of the value of the property CPP\_CG::Operation::Inline. This issue was fixed in 8.0.3. To preserve the previous code generation behavior for pre-8.0.3 models, the CG::Class::ForceInDeclarationForNestedTemplate property was added to the backward compatibility settings for C++ with a value of True.

## **ForceReactive**

The ForceReactive property is a Boolean value that specifies whether to generate a class as reactive, regardless of any other reactive criteria. A reactive class, that has a base class with this property check box checked, does not inherit directly from the framework reactive base class.

*(Default = Cleared)*

## **GenerateDependencyToExternals**

Prior to version 8.0, if you created a dependency from a model element to an element defined as external, the generated code would not contain an #include statement for the external element. Beginning in version 8.0, #include statements are generated for dependencies on external elements. To preserve the previous code generation behavior for pre-8.0 models, the CG::Class::GenerateDependencyToExternals property was added to the backward compatibility settings for C and C++ with a value of False.

## **GenerateImplicitConstructors**

The GenerateImplicitConstructors property is a Boolean value that specifies whether to generate implicit

constructors. When the property check box is cleared, The software generates only user-specified constructors.

*(Default = Checked)*

## GenerateImplicitDependencies

When code is generated, the file generated for a class automatically contains certain #include directives for elements such as:

- the package that contains the class
- base classes
- types for any attributes and associations in the class
- types for any arguments and return types of operations in the class

If you set the value of GenerateImplicitDependencies to False, these #include directives will not be automatically generated in the class file.

When the property is set to False, you can add dependencies with the Usage stereotype to have such #include directives generated selectively.

*Default = True*

## GuardDestruction

If the GuardDestruction property is set to Checked, then the event queue cannot be destroyed in the middle of handling an event, and events cannot be consumed from the queue if the active/reactive class instance is in the process of being destroyed.

This property can be overridden or set at the site or project level to ensure that it is the default behavior.

*(Default = Cleared)*

## ImplementStatechart

The ImplementStatechart property specifies whether to generate behavioral code for a reactive object. To use a statechart as documentation of behavior only (without generating behavioral code), do the following:

- Create a statechart for the object.
- Set the ImplementStatechart property for the object to Cleared.
- Override the OMReactive::processEvent() method (in C++), or the RiCReactive.processEvent() function (in C), which implements the statechart.

This is one way of optimizing your statechart code. The default value for Rational Rhapsody Developer for Ada is Cleared; for all other languages, the default value is checked.

## ImplicitDependencyToPackage

The ImplicitDependencyToPackage property is a Boolean value that determines whether the dependency from a class to its package is automatically generated.

*(Default = Checked)*

## IncomeSignalMap

This property is used to map incoming SDL model signals to their SDL entry channels. The value of this property should not be modified by the user.

## InitCleanUpRelations

The InitCleanUpRelations property specifies whether to generate initRelations() and cleanUpRelations() operations for sets of related global instances. This property applies only to composites and global relations.

*(Default = Checked)*

## InitializerValue

*The InitializerValue property initializes statically allocated objects and singletons. For example, for a singleton object A, if the InitializerValue property is set to {1,2,"abc"}, the resulting code is: struct A\_t A = {1,2,"abc"}; You can access this property directly from the constructor window in the browser. (Default = empty string)*

## IsCompletedForAllStates

The IsCompletedForAllStates property specifies whether you can use the IS\_COMPLETED(state) macro for all types of states. This macro is generally used in activity diagrams, but can also be used in statecharts.

The possible values are as follows:

- Checked - The IS\_COMPLETED(state) macro can be used for all types of states.
- Cleared - The IS\_COMPLETED(state) macro can be used only for states that have a Final state.

*(Default = Cleared)*

## MaximumPendingEvents

*Reserved for future use. (Default = -1)*

## OperationsAutoArrange

OperationsAutoArrange property that determines the order in which operations are generated in the code. If set to checked, the order of the operations in the generated code is based on the Rational Rhapsody default order for the programming language being used.

If set to Cleared, the order of the operations in the generated code is the order specified by the user.

This means the order in which the user added the operations, unless the user overrode this order by making changes in the Edit Operations Order window or by using the up/down buttons on the browser.

*(Default = Checked)*

## OutcomeSignalMap

This property is used to map outgoing SDL model signals to their SDL exit channels. The value of this property should not be modified by the user.

## ProtectStaticMemoryPool

The ProtectStaticMemoryPool property specifies whether to protect the static memory pool by using an operating system mutex. This property helps support static architectures.

*(Default = Checked)*

## ReactiveSimpleComposites

The ReactiveSimpleComposites property was added for compatibility with earlier versions. See the upgrade history on the support site for more information on this property.

## RelationsAutoArrange

*The RelationsAutoArrange property is a Boolean UI helper property. You should not modify this property directly. (Default = Checked)*

## StandardOperations

The StandardOperations property adds template-based code (based on resolution of keywords) to a class or event.

Every standard operation is associated with a logical name. You define the logical name of a standard operation by overriding the StandardOperations property to add a comma-separated list of the names of the standard operations you want to define.

For every standard operation defined, you also need to specify an operation declaration and definition.

This is done by adding the following two properties to the site.prp file to specify the necessary function templates:

- LogicalName>Declaration - Specifies a template for the operation declaration
- LogicalName>Definition - Specifies a template for the operation implementation

For example, for a logical name of myOp, you would define the following property (by using the site.prp file or the COM API(VBA)) :

```
Subject CG Metaclass Class Property myOpDeclaration MultiLine "" Property myOpDefinition MultiLine "" end end
```

You add all of the properties to be associated with a standard operation to the site.prp file under their respective CG subject and metaclasses. All of these properties should have a type of MultiLine.

## UseAsExternal

The UseAsExternal property specifies whether an object is referenced as an external object (one that was not generated in Rational Rhapsody). This property references an external object in the model without generating code for it.

To prevent compilation errors with instrumentation when inheriting from an external base class (in C++), set the UseAsExternal property for the external base class to checked.

This prevents serialization code from being added to the base class. Similarly, the serialization operations are not called in the subclass.

Setting the FileName property for the base class (for example, to “BaseClass” without the “.h”) generates an #include of the base class in the specification file of the subclass. If the FileName property is not defined, you must add the appropriate code to the SpecificationProlog property for the subclass.

You can also use the UseAsExternal property to create template (parameterized) classes as follows:

- Create the template class outside of Rational Rhapsody.
- In the model, define a template class (for example, DataStore) as a placeholder for the external code defining the class.
- Set the UseAsExternal property for DataStore in the Rational Rhapsody model to checked.

Setting UseAsExternal to checked for the class prevents code from being generated for the external class when you generate code. You can still use the class in an object model diagram and inherit from it. If you want to inherit from an external template class, you can specify the external base class in the SpecificationProlog property for the subclass.

*(Default = Cleared)*

## Component

The Component metaclass contains properties for implementing components.



## CalculatePackageEventBaseId

The CalculatePackageEventBaseId property specifies how Rational Rhapsody calculates each package base event ID.

The possible values are as follows:

- OnCodeGeneration - Calculate the package base event ID during code generation.
- OnCreatePackage - Calculate the package event base ID when the package is created.

*Default = OnCreatePackage*

## ComponentsSearchPath

The ComponentsSearchPath property specifies the name of related components. Adding the name of a component here causes the code generator to search the component for the file names of any related model elements that are not found in the original component. They are also added to the makefile search path.

Related components must have the same configuration name.

*(Default = empty string)*

## GenerateComponentCleanup

The GenerateComponentCleanup property determines whether Rational Rhapsody creates a \_Cleanup operation for components. If the value of the property is set to Never, the operation is not generated in the code. If the property is set to Smart, the operation is generated where relevant.

*Default = Smart*

## GenerateComponentInitialization

The GenerateComponentInitialization property determines whether Rational Rhapsody creates an \_Init operation for components. If the value of the property is set to Never, the operation is not generated in the code. If the property is set to Smart, the operation is generated where relevant.

*Default = Smart*

## InitializationScheme

The InitializationScheme property specifies how relations are initialized. This has implications for how relations are initialized across packages.

The possible values are as follows:

- ByPackage - Each package is responsible for its own initialization. The only responsibility for the component is to declare an attribute for each package in the class.

- **ByComponent** - The component is responsible for initializing all global relations declared in all packages. This initialization is done with explicit calls in the component-class constructor for each package that uses `initRelations()` code (specified by the `AdditionalInitialization` property) and `startBehavior()`.

*Default = ByComponent*

## **PackageCtrlDPMC**

The `PackageCtrlDPMC` property was added for compatibility with earlier versions. See the upgrade history on the support site for more information on this property.

## **PackageEventBaseIdAlgorithm**

The `PackageEventBaseIdAlgorithm` property specifies the algorithm Rational Rhapsody uses to calculate the package base event ID.

Note that this property value is only in effect when `CG::Component::CalculatePackageEventBaseId` is set to `OnCreatePackage`. The possible values are as follows:

- **Fixed** - The ID is based on random numbers.
- **Hash** - The ID is based on the name of the package.

*(Default = Hash)*

## **RelatedComponentsIncludePathInMakefile**

*The `RelatedComponentsIncludePathInMakefile` property was added for compatibility with earlier versions. See the upgrade history on the support site for more information on this property. (Default = Cleared)*

## **UseDefaultNameForUnmappedElements**

This `UseDefaultNameForUnmappedElements` property supports the use of an improved file name decision algorithm for naming files in code generation.

This property influences how file names are generated for elements that are out of the normal code generation scope of a component. By default, `#include` statements are not generated for elements that are out of the scope of the active component and its related components.

This prevents including non-existing files, which would cause compilation errors. The `UseDefaultNameForUnmappedElements` property provides control over how elements that are out of the known scope are generated.

The possible values are as follows:

- **Checked** - Use the default name for the element.
- **Cleared** - The name is an empty string. The result is that no `#include` statement can be generated for

that file.

*(Default = Checked)*

## Configuration

The Configuration metaclass contains properties for implementing configurations.

### AddExplicitInitialInstancesToScope

The AddExplicitInitialInstancesToScope property includes explicit initial instances as part of the scope for code generation. Beginning with version 5.0, Rational Rhapsody does not include explicit initial instances as part of the scope.

In other words, in explicit mode, code is not generated for a class just because it is in the list of initial instances for the configuration.

For existing models, Rational Rhapsody sets the CG::Configuration::AddExplicitInitialInstancesToScope property to checked at the project level to maintain the old behavior. This change uses the list of initial instances to create instances that their classes defined in related components (libraries).

*(Default = Checked)*

### AllowCollisionWithComponentName

The AllowCollisionWithComponentName property enables a check that prevents code generation when a class, actor, event, or global variable within the component scope has the same name as the component. In Rational Rhapsody Developer for Java, the check prevents generation of a class with the name Main "component."

This check was added because the software generates a class for the component.

When the model has global instances, multiple definitions of the same class are generated, one for the user class and the other for the generated component class. This means that if your model has elements and component with the same name, you must modify the class name, or the component name, in order to generate code.

*You can disable the check by setting the AllowCollisionWithComponentName property to checked. However, if you do this, Rational Rhapsody protects you from redefinition and name collision at the code level. (Default = Cleared)*

### AnimateAllPackages

In release 7.4.1 of Rhapsody, animation code was generated even for packages that were not included in the animated scope. This was corrected in release 7.5.

To allow users to maintain the previous behavior for older models, if they so desire, the pre-75 compatibility settings for C and C++ include the property `AnimateAllPackages` with a value of `True`.

## CaseStmtPutBreakLast

The `CaseStmtPutBreakLast` property exists in models created before Rational Rhapsody 7.5. If the property is set to `Checked`, the break statement is always the last generated statement in the case part of the switch statement. For example:

```
switch (rootState_active) { case fifo_has_packets: { .... break; } case: ... }
```

If the property is set to `Cleared`, the break statement is put inside of the block (parentheses). For example:

```
switch (rootState_active) { case fifo_has_packets: { .... } break; case: ... }
```

*Default = Cleared*

## CodeGeneratorTool

The `CodeGeneratorTool` property specifies which code generation tool to use for the given configuration.

The possible values are as follows:

- External - Use the registered, external code generator.
- Internal - Use the software internal code generator.

The default value for Rational Rhapsody Developer for Ada is `External`; for all other languages, the default value is `Internal`.

## CodeUpdate

The `CodeUpdate` property is responsible for the selective code generation used in code-centric mode.

In code-centric mode, the code-generation behavior is based on the premise that if you add any code-related elements to your model, you would prefer that Rational Rhapsody make as few changes as possible to your code.

*Default = True (in code-centric settings)*

## ExternalGenerationTimeout

The `ExternalGenerationTimeout` property specifies how long, in seconds, Rational Rhapsody waits for the each class in the configuration scope to complete so you can once again make changes to the model. This property applies to both the full-featured external generator and makefile generator.

For example, if you set this property to 2 and you have 10 classes, Rational Rhapsody sets a timeout of 20. If the external code generator does not complete generation in this timeframe, Rational Rhapsody displays

a message in the output window saying that the generator is not responding, and you are allowed to make changes to the model.

If you set this property to 0, Rational Rhapsody does not time out the generation session and waits for the code generator to complete its task - even if it takes forever. Rational Rhapsody waits for a notification from the full-featured external code generator or for the process termination of a makefile generator.

(Default = 0)

## ExternalGeneratorFileMappingRules

The `ExternalGeneratorFileMappingRules` property specifies whether the external code generator uses the same file mapping and naming scheme (mapping rules) as Rational Rhapsody.

If the mapping rules are different, the external generator must implement handlers to the `GetFileName`, `GetMainFileName`, and `GetMakefileName` events that Rational Rhapsody runs to get a requested file name and path.

The possible values are as follows:

- `AsRhapsody` - The external generator uses the same mapping rules as Rational Rhapsody.
- `DefinedByGenerator` - The external generator has its own mapping rules.

The default value for Ada is `DefinedByGenerator`; for all other languages, the default value is `AsRhapsody`.

## GenerateDirectoryPerModelComponent

The `GenerateDirectoryPerModelComponent` property specifies whether to generate a separate directory for each package in the component. The possible values are:

- `Checked` - Rational Rhapsody creates a separate directory for each package in the component. (This is the default.)
- `Cleared` - A separate directory is not created for each package.

## GeneratorExtraPropertyFiles

The `GeneratorExtraPropertyFiles` property opens the default Text Editor.

## GenerateForwardDeclarations

The `GenerateForwardDeclarations` property is a Boolean property that specifies whether forward declarations are generated.

When set to `True`, the generation of forward declarations is carried out according to the value of the property `CG::Dependency::UsageType`.

When set to `False`, the specification file will not contain a forward declaration even if the value of

UsageType is set to Existence or Implementation.

*Default = True*

## GeneratorRulesSet

The GeneratorRulesSet property specifies your own rules set.

*(Default = empty MultiLine)*

## GeneratorScenarioName

*The GeneratorScenarioName property specifies the scenario name for the rule, if you write your own set of code generation rules. (Default = empty string)*

## IncludeRequirementsAsComments

The IncludeRequirementsAsComments property determines the policy of generating requirements comments in code. The possible values are:

- UseDescriptionTemplate - The requirements comment is generated according to the DescriptionTemplate property (if it contains the \$Requirements keyword) for a metaclass.
- Always - The requirements comments is generated for all relevant (has DescriptionTemplate property) metaclass including states.

*Default = UseDescriptionTemplate*

## LineWrapLength

The LineWrapLength property specifies the length of the code line generated during code generation. For example, if this property has the value 250, the generated code lines are wrapped to 250 characters. A value of 0 means that lines of code are not wrapped.

Note that code for the following elements is not wrapped, regardless of this property setting:

- User code such as statechart actions and operation bodies
- The comments generated for element descriptions
- Element annotations
- Makefiles

*Default = 0*

## MainGenerationScheme

The MainGenerationScheme property controls how the main procedure is generated. This property is required for compliance with MISRA (Motor Industry Software Reliability Association) rules. The

possible values are as follows:

- Full - The main procedure is generated as usual.
- UserInitializationOnly - The main contents generation is switched off and is replaced with only the initialization code field. This means that users can rewrite the main exactly as they want and then have to add any code that would normally be generated automatically by Rational Rhapsody.
- For example, you would have to add the code for DefaultComponent\_Init():

```
int main(int argc, char* argv[]) { /*#[ configuration DefaultComponent::DefaultConfig */ // This is the
initialization code added by the user /*#]*/ }
```

*(Default = Full)*

## MainOpSingleExitPoint

The MainOpSingleExitPoint property specifies that the main function is generated in MISRA-compliant style if this property is set to Checked. For example:

```
int main(int argc, char* argv[]) { int status = 0; if(OXF::initialize(argc, argv, 6423)) { ... OXF::start();
status = 0; } else { status = 1; } return status; }
```

Otherwise, it contains two return statements, as shown in the following example:

```
int main(int argc, char* argv) { if(OXF::initialize(argc, argv, 6423)) { ... OXF::start(); return 0; } else {
return 1; } }
```

*Default = Cleared*

## NotifyNeedOfModelCodeSync

The NotifyNeedOfModelCodeSync property is an enumerated type that controls whether Rational Rhapsody displays the message “Do you want to regenerate?” when you try to build a configuration. The possible values are as follows:

- Never - Never display the message.
- OnDynamicModelCodeAssociativity - Display the message only when the code generation is sensitive to changes (dynamic model-code associativity is on).
- Always - Always display the message.

*(Default = OnDynamicModelCodeAssociativity)*

## PostFrameworkThreadSegment

The PostFrameworkThreadSegment property is a free text property added to the main() after the call to the framework start() (OXF::start() in Rational Rhapsody Developer for C++).

In order for this code to be executed, you must set the CG::Configuration::StartFrameworkInMainThread property to Cleared. The default value for C and Java is an empty MultiLine; the default value for Ada and

C++ is an empty string.

## PreFrameworkInitCode

The PreFrameworkInitCode property specifies text that is added to the generated main() before the call to the framework initialization (OXF::init() in Rational Rhapsody Developer for C++).

*This property was added to support additional customization features. For example, you could use this property to change the names of the OXFInit method arguments, argc and argv. (Default = empty MultiLine)*

## RemoveWhiteSpacesInBuildFile

The RemoveWhiteSpacesInBuildFile property determines whether spaces are removed from MULTI and INTEGRITY makefiles generated for models created before Rational Rhapsody 6.0.

In models created by using Rational Rhapsody 6.0, spaces are not removed and this property is not available: the property is created automatically when you load a pre-Rational Rhapsody 6.0 model.

*You can add this property to the site.prp file, or directly change the factory.prp file, so you can access it for new models. (Default = Checked)*

## SortAssociationEndsAndPartsTogether

Prior to release 8.1.3, the browser allowed you to edit the order of parts and associations together, but in the generated code the association ends were sorted separately from the parts.

Beginning in release 8.1.3, the association ends and parts are sorted together in the generated code, so any changes that you make to the order of association ends and parts are reflected in the generated code as well.

In order to preserve the previous code generation behavior for pre-8.1.3 models, the property CG::Configuration::SortAssociationEndsAndPartsTogether was added to the backward compatibility settings for C and C++, with a value of False.

## StartFrameworkInMainThread

*The StartFrameworkInMainThread property is a Boolean value that determines whether the framework default event dispatcher should run in the main thread. If this is Cleared, the event dispatcher runs in a new thread. (Default = Checked)*

## StrictExternalElementsGeneration

The StrictExternalElementsGeneration property is a Boolean value that specifies whether to take advantage of information in modeled external elements during code generation. This property is used for compatibility with earlier versions (refer to the upgrade history on the support site for more information).



By default, this property check box is checked for models created before Rational Rhapsody 5.2. For more information on modeling external elements, refer to the Rational Rhapsody help.

*(Default = Checked)*

## **SupportExternalElementsInScope**

The SupportExternalElementsInScope property is a Boolean value that specifies whether to include external elements in the component scope. This property is used for compatibility with earlier versions (refer to the upgrade history on the support site for more information).

By default, this property check box is cleared for models created before Rational Rhapsody 5.2. For more information on modeling external elements, refer to the Rational Rhapsody help.

*(Default = Cleared)*

## **TrailingElseClause**

The TrailingElseClause property can be used to add an else clause following an if/else if construct. The default value for this property is an empty string. If you replace this with a different string, the text you entered is placed within the braces of the else clause when the code is generated.

## **UnicodeEnvironment**

The UnicodeEnvironment property makes the code generator assume that it is working in a Unicode environment (such as some Japanese versions). This property was added so non-English comments are generated correctly.

*Default = Cleared*

## **UseDescriptionTemplates**

The UseDescription property is a Boolean value that is used to enable template-based element descriptions. When enabled, you can define description templates by using the DescriptionTemplate property provided for various types of model elements. When code is generated, the tags used in the description template definitions are expanded in the element description.

For Rational Rhapsody Developer for Java, the JavaDocProfile is loaded automatically for newly created Java projects with the default behavior to generate JavaDoc comments. To change the default for new Java projects, clear the check box for the UseDescriptionTemplates property. To disable the feature for a specific project, clear the Generate JavaDoc Comments check box on the Settings tab of the Configuration window. To enable JavaDoc comments on existing projects, load the JavaDocProfile.

*Default = Checked*

# Dependency

The Dependency metaclass contains properties for implementing dependencies.

## AutoGenAnnotationsForImplicitDependencies

Prior to release 8.0.4, there were cases where an `#include` directive was generated for an implicit dependency but the annotation preceding the `#include` directive just said "auto\_generated" rather than indicating the actual reason that the directive was generated. This issue was fixed in 8.0.4.

To preserve the previous code generation behavior for pre-8.0.4 models, the `CG::Dependency::AutoGenAnnotationsForImplicitDependencies` property was added to the backward compatibility settings for C++ with a value of `False`.

## ConfigurationDependencies

In Rational Rhapsody, you can define a number of configurations for each component.

If you specify a dependency from component A to component B, the code-generation behavior is based on the assumption that component B has a configuration with the same name as the active configuration of component A. For example, configuration "debug" in component A would be assumed to be dependent upon configuration "debug" in component B.

Since you may not always have configurations with the same name in the two components, the `ConfigurationDependencies` property allows you to specify that a dependency is between configurations with different names in the two components.

The value for this property should be a comma-separated list of configuration pairs (one configuration from the first component and one configuration from the second component). If component A is dependent on component B, then in each pair, the first configuration should be the relevant configuration in component A, and the second configuration should be the configuration in component B on which component A is dependent. For example:

```
configOneInA:configTwoInB, configA2:configB3
```

*Default = Blank*

## ForwardDeclarationPlacement

The `ForwardDeclarationPlacement` property allows positioning of the declaration either "before elements" or "at the top of the file."

(Default=BeforeElements)

## GenerateRelationWithActors

The GenerateRelationWithActors property is an enumerated type that controls the generation of the dependency or relation to an actor (if it is a dependency or relation to an actor).

Control over generation of actors is done by the scope and by the appropriate check box in the Configuration Initialization tab.

The possible values for this property are as follows:

- Never - Never generate the dependency or relation.
- WhenActorIsGenerated - Generate the dependency or relation when the actor is generated.
- Always - Always generate the dependency or relation.

*(Default = WhenActorIsGenerated)*

## GenerateRequirement

When you apply the "trace" stereotype to a dependency from an operation or function to a requirement, the Features window includes a "Show in" field which allows you to specify whether the dependency should be generated in the specification file, the implementation file, or both of these files.

The property GenerateRequirement reflects the value of the "Show in" field.

*Default = Specification+Implementation*

## PropagateImplementationToDerivedClasses

When the CG::Dependency::UsageType property is set to "Implementation", the .cpp file of the dependent element contains an #include to the .h file of the element on which it depends. If the dependent element is a base class, then the .cpp files of its derived classes must also contain this #include. If you do not want this #include to be propagated to derived classes, set the value of the property PropagateImplementationToDerivedClasses to False.

*(Default = Checked)*

## UsageType

When you model a usage dependency, you can use the UsageType property to control how the dependency is reflected in the generated code. The possible values are:

- Existence - a forward declaration is generated in the specification file
- Specification - an #include statement is generated in the specification file
- Implementation - a forward declaration is generated in the specification file, and an #include statement is generated in the implementation file

Note that the code generated also depends on the value of the property

[lang]\_CG::Dependency::GenerateForwardDeclarations. If GenerateForwardDeclarations is set to True, code is generated as described above. However, if GenerateForwardDeclarations is set to False, then the specification file will not contain a forward declaration even if the value of UsageType is set to Existence or Implementation.

*Default = Specification*

## UsePre80Dependencies

Prior to version 8.0.3 of Rational Rhapsody, there were situations where an #include directive was generated in both the specification and implementation file. This issue was fixed in 8.0.3. To preserve the previous code generation behavior for pre-8.0.3 models, the CG::Dependency::UsePre80Dependencies property was added to the backward compatibility settings for C++ with a value of True.

## Event

The Event metaclass contains properties for implementing events.

## AdditionalNumberOfInstances

The AdditionalNumberOfInstances property is a string that specifies the size of the local heap allocated for events when the current pool is full. Triggered operations use the event properties.

This property provides support for static architectures found in hard real-time and safety-critical systems without memory management capabilities during run time. All events are dynamically allocated during initialization.

Once allocated, the event queue for a thread remains static in size.

The possible values are as follows:

- An empty string (blank) - No additional memory is to be allocated when the initial memory pool is exhausted.
- n (a positive integer) - Specifies the size of the array allocated for additional instances. (Default = empty string)

## Animate

The Animate property specifies whether animation code is generated for an element. You can specify your own animation function by using the CPP\_CG::Type::AnimSerializeOperation property.

The semantics of the Animate property is always in favor of the owner settings:

- If a package Animate property check box is cleared, all the classes owned by the package are not animated, regardless of the class Animate settings.

- If a class Animate property check box is cleared, all the elements in the class (attributes, operations, relations, and so on) are not animated.
- If an operation Animate property check box is cleared, all the arguments are not animated.
- If the AnimateArguments property check box is cleared, all the arguments are not animated, regardless of the specific argument Animate property settings. (Default = Checked)

## AnimateArguments

The AnimateArguments property specifies whether to animate arguments to events or operations in animated sequence diagrams.

During instrumentation, Rational Rhapsody displays the values of actual parameters passed as arguments to events or operations in animated sequence diagrams by using the C++ stream output operator.

For this to work, the argument must be of a built-in type that can be output to a stream, such as an int or a char\*. If, however, the argument is of a complex, user-defined type that the compiler cannot serialize, then trying to animate it causes compilation errors. In VxWorks, trying to animate arguments of type long unsigned int also causes errors.

If you want to animate arguments of user-defined types defined either inside or outside of Rational Rhapsody, you must add to the framework an overloaded C++ stream output operator for the type.

*(Default = Checked)*

## BaseNumberOfInstances

The BaseNumberOfInstances property is a string that specifies the size of the local heap memory pool allocated for instances of an event.

This property provides support for static architectures found in hard real-time and safety-critical systems without memory management capabilities during run time. All instances of events are dynamically allocated during initialization.

Once allocated, the event queue for a thread remains static in size.

Triggered operations use the properties defined for events.

When the memory pool is exhausted, an additional amount, specified by the AdditionalNumberOfInstances property, is allocated.

The possible values are as follows:

- An empty string (blank) - Memory is always dynamically allocated.
- n (positive integer) - An array is allocated in this size for instances.

## DeleteAfterConsumption

The DeleteAfterConsumption property creates an event that cannot be deleted after it has been consumed.

Such events are usually generated by interrupt handlers (see also `GEN_ISR(event)`).

The following function is an example of an interrupt handler function that uses this type of event: `void handler() { // Called upon an interrupt static Emergency e[10]; // Assumption: 10 events are enough static int i = 0; e[i].severity = 1; // Setting the event parameter itsSafetyController-gen(e[i]); // Do not use GEN here !! i=(i+1)% 10; // Incrementing the buf pointer }`

The possible values are as follows:

- **Default** - The default policy is to delete the event after consumption. However, using this value allows for possible future changes to the default policy. For example, events could be deleted some times, and not other times.
- **Checked** - The event is always deleted after consumption.
- **Cleared** - The event is never deleted after consumption.

*(Default = Default)*

## EmptyMemoryPoolCallback

The `EmptyMemoryPoolCallback` property specifies a name for the callback function that allocates more memory if the static pool is exhausted.

*This property provides support for static architectures. (Default = empty string)*

## EmptyMemoryPoolMessage

The `EmptyMemoryPoolMessage` property specifies whether a message is output when the static memory pool is empty. This property provides support for static architectures.

*(Default = Checked)*

Note: This property is only active during animation.

## Generate

*The Generate property specifies whether to generate code for a particular type of element. (Default = Checked)*

## GenerateCleanupOp

The property `GenerateCleanupOp` can be used to control the generation of clean-up code for events. If you do not want to have this code generated, set the value of the property to `False`.

*Default = True*

## GenerateInitOp

The property `GenerateInitOp` can be used to control the generation of initialization code for events. If you do not want to have this code generated, set the value of the property to `False`.

*Default = True*

## GenEventIdAssignment

*This property generates code for a specific event assignment. (Default = Checked)*

## Id

The `Id` property is a string that assigns your own identification number to an event or triggered operation. The `Id` property assigns a permanent ID number to the event or triggered operation.

This supports the development of distributed systems in which an event or triggered operation must have the same ID in different parts of the system.

If a permanent ID is not assigned by using this property, the number might change (for example, if the system has multiple components).

*A check is performed before code is generated to verify that there are no conflicts between user-defined IDs and generated IDs. (Default = empty string)*

## IdNameScheme

The `IdNameScheme` property determines how the `#define` for the event is generated. The possible values are `Full` and `Short`.

If `Full` is selected, the `#define` for the event includes the entire hierarchy of the packages, for example:  
`#define P1_P2_P3_Ev 2`.

If `Short` is selected, the `#define` for the event only includes the name of the event and the name of the package it is under, for example: `#define P3_Ev 2` The default value is `Full`. (Rhapsody versions prior to 6.1 MR-1 used this format.)

## ProtectStaticMemoryPool

*The `ProtectStaticMemoryPool` property specifies whether to protect the static memory pool by using an operating system mutex. This property helps support static architectures. Triggered operations use the properties for the event. (Default = Checked)*

## StandardOperations

The `StandardOperations` property adds template-based code (based on resolution of keywords) to a class or event. Every standard operation is associated with a logical name. You define the logical name of a standard operation by overriding the `StandardOperations` property to add a comma-separated list of the

names of the standard operations you want to define.

For every standard operation defined, you also need to specify an operation declaration and definition. This is done by adding the following two properties to the site.prp file to specify the necessary function templates:

- Declaration - Specifies a template for the operation declaration
- Definition - Specifies a template for the operation implementation

For example, for a logical name of myOp, you would define the following property (by using the site.prp file or the COM API(VBA)) : Subject CG Metaclass Class Property myOpDeclaration MultiLine ""  
Property myOpDefinition MultiLine "" end

You add all of the properties to be associated with a standard operation to the site.prp file under their respective CG subject and metaclasses. All of these properties should have a type of MultiLine.

## File

The File metaclass contains properties for implementing files.

### AddToMakefile

The AddToMakefile property specifies whether a file is added to the makefile (and, therefore, built). This property supports modeling of flat source files (text without model elements).

This property supersedes the GenerateInMakefileOnly property.

To control whether code is generated for a file, use the CG::File::Generate property.

In general, these two properties - Generate and AddToMakeFile - operate independently. Note, however, that if Generate is set to True, then the file will be referenced in the makefile even if the property AddToMakeFile is set to False.

*Default = Checked*

### Footer

The Footer property specifies a multiline footer that is added to the end of generated files. The default footer template is as follows:

```
/****** File
Path:$FullCodeGeneratedFileName *****/
```

Footer format strings can contain any of the following keywords:

- \$ProjectName - The project name.



- `$ComponentName` - The component name.
- `$ConfigurationName` - The configuration name.
- `$ModelElementName` - The name of the element mapped to the file. If there is more than one, this is the name of the first element.
- `$FullModelElementName` - The name of the element mapped to the file, including the full path. If there is more than one, this is the name of the first element.
- `$CodeGeneratedDate` - The generation date.
- `$CodeGeneratedTime` - The generation time.
- `$RhapsodyVersion` - The version of Rational Rhapsody that generated the file.
- `$Login` - The user who generated the file.
- `$CodeGeneratedFileName` - The name of the generated file.
- `$FullCodeGeneratedFileName` - The full file name.
- `$Tag` - The value of the specified the element tag.
- `$Property` - The value of the element property with the specified name.

To avoid redundant compilation, Rational Rhapsody avoids unnecessary changes to specific lines prefixed with a special string, defined by the property `lang_CG::File::DiffDelimiter`. The default `DiffDelimiter` value is `“//!”`.

## Generate

*The `Generate` property specifies whether to generate code for a particular type of element. (Default = Checked)*

## Header

The Header property specifies a multiline header that is added to the top of all generated files. The default header template is as follows: `/******`

```
Rhapsody : $RhapsodyVersion Login : $Login Component : $ComponentName Configuration :
$ConfigurationName Model Element : $FullModelElementName //! Generated Date :
$CodeGeneratedDate File Path : $FullCodeGeneratedFileName
*****/ For example:
/***** Rhapsody : 4.0 Login : pie_man
Component : BakedGood Configuration : DefaultConfig Model Element : Fairgoer::Simon //! Generated
Date : Mon, 27, Nov 01 File Path : d:\projects\DefaultConfig\simple.h
*****/ Header format strings can contain
any of the following keywords:
```

- `$ProjectName` - The project name.
- `$ComponentName` - The component name.
- `$ConfigurationName` - The configuration name.
- `$ModelElementName` - The name of the element mapped to the file. If there is more than one, this is the name of the first element.
- `$FullModelElementName` - The name of the element mapped to the file, including the full path. If there is more than one, this is the name of the first element.

- `$CodeGeneratedDate` - The generation date.
- `$CodeGeneratedTime` - The generation time.
- `$RhapsodyVersion` - The version of Rational Rhapsody that generated the file.
- `$Login` - The user who generated the file.
- `$CodeGeneratedFileName` - The name of the generated file.
- `$FullCodeGeneratedFileName` - The full file name.
- `$Tag` - The value of the specified the element tag.
- `$Property` - The value of the element property with the specified name.

To avoid redundant compilation, Rational Rhapsody avoids unnecessary changes to specific lines prefixed with a special string, defined by the property `lang_CG::File::DiffDelimiter`. The default `DiffDelimiter` value is `“//!”`.

## HeaderDirectivePattern

The `HeaderDirectivePattern` property controls the format of the `#ifndef` pattern in the generated `.h` files that prevents multiple definitions of the class.

The `$FULLFILENAME_H` keyword expansion is related to the actual directory structure and reflects the directory structure of the generated file (starting from the configuration directory).

When you use the keyword `$FILENAME_H`, the directory generated for the package is not reflected in the `#ifndef` and `#define` statements; when you use the `$FULLFILENAME_H` keyword, the directory is reflected in the statements.

Consider the following example: you have a package, `P`, that has a class, `A`. The `C_CG/ CPP_CG/JAVA_CG::Package::GenerateDirectory` is set to `Checked`.

If you set the class property to `$FILENAME_H`, the following code is generated in the specification file for `A`:

```
#ifndef A_H #define A_H class A { ... }; #endif
```

If the property is set to `$FULLFILENAME_H`, the following code is generated in the specification file for `A`:

```
#ifndef P_A_H #define P_A_H class A { ... };
```

This property value is also used during roundtrip to ignore `#define` statements that match the pattern.

*(Default = `$FULLFILENAME_H`)*

## ImpExtension

The `ImpExtension` property specifies the extension that Rational Rhapsody appends to generated and external physical files for files in packages, components, and class/object/block/actor elements.

Note the following:

- This property is visible for files in packages, components, and class/object/block/actor elements.
- There is an `ImpExtension` property under `lang_CG::Environment`. However, the properties in the `File` metaclass have higher priority than those in the `Environment` metaclass.

*(Default = empty string)*

## IncludeScheme

The `IncludeScheme` property specifies whether the path information is included in the file name in an `#include` statement. The possible values are as follows:

- `RelativeToConfiguration` - Include the relative path from the configuration directory in the include files.
- `LocalOnly` - The name of the include file is generated without the path information.

The property is first evaluated by the target file in the component model, if the file exists. If the file does not exist, the value is taken from the active configuration. This behavior specifies the generation defaults for elements without files.

Note that this property affects only the files defined in the component, not the class. Therefore, you should define this property on the project, component, configuration, or file.

The property does not work correctly if you define for the class (even if the class is an element of the file, defined in the component).

*(Default = RelativeToConfiguration)*

## InvokePostProcessor

The `InvokePostProcessor` property runs a post-processing utility on the code that is generated by Rational Rhapsody. For example, you could run a “beautify” program to get a specific coding style.

When this property value is not empty, Rational Rhapsody runs a process by using the specified command.

You can specify the post-processing command on a single file or higher (folder, configuration, component, project, or site). You can specify the following keywords as part of the command:

- `$file` - The name of the generated file
- `$projectPath` - The current project root directory

Rational Rhapsody generates code by using the following sequence of events:

- Rational Rhapsody generates code into a temporary file.
- If the target file already exists (because of a previous build), Rational Rhapsody compares the temporary file to the target file.
- If there are differences, the target file is replaced with the temporary file.

- If you specified a post-processor command, Rational Rhapsody runs the post processor on the temporary files. Any messages from the post-processor are displayed in the Output window.
- The temporary files are copied to the final location.

## SpecExtension

The SpecExtension property specifies the extension that Rational Rhapsody appends to generated and external physical files for files in packages, components, and class/object/block/actor elements.

When you generate code for a dependency to a file with a non-standard extension, the generated #include includes the correct file extension, taken from this property.

Note that there is a SpecExtension property under lang\_CG::Environment. However, the properties in the File metaclass have higher priority than those in the Environment metaclass.

*See also the ImpExtension property. (Default = empty string)*

## Framework

### MicroCOxfCleanup

Use this property to set whether to do clean up on dynamically allocated elements.

*Default = Cleared*

### RapidPorts

The RapidPorts property specifies whether the MicroC Execution Framework (MXF) provides services for using Rapid Ports. This property configures the way the MXF is built.

### RiCCollection

The RiCCollection property specifies whether the MicroC Execution Framework (MXF) provides services for using the collection: RiCCollection. This property configures the way the MXF is built.

### RiCList

The RiCList property specifies whether the MicroC Execution Framework (MXF) provides services for using the collection: RiCList. This property configures the way the MXF is built.

### RiCMap

The RiCMap property specifies whether the MicroC Execution Framework (MXF) provides services for using the collection: RiCMap. This property configures the way the MXF is built.

## RiCReactiveGenMacros

The RiCReactiveGenMacros property specifies whether the MicroC Execution Framework (MXF) provides services for using event generation-related macros (for example, RiCGEN). This property configures the way the MXF is built. The possible values are:

- Include - Include event generation-related macros in MXF compilation.
- Exclude - Exclude event generation-related macros from MXF compilation.

*Default = Include*

## RiCReactiveStateMacros

The RiCReactiveStateMacros property specifies whether the MicroC Execution Framework (MXF) provides services for using state-related macros (for example, IS\_IN, IS\_COMPLETED). This property configures the way the MXF is built. The possible values are:

- Include – Include state-related macros in MXF compilation.
- Exclude – Exclude state-related macros from MXF compilation.

*Default = Include*

## RiCStack

The RiCStack property specifies whether the MicroC Execution Framework (MXF) provides services for using the collection: RiCStack. This property configures the way the MXF is built.

## General

The General metaclass contains a property used to support incremental code generation. See the Rational Rhapsody help for more information on incremental code generation.

## AbortOnModelChecker

The AbortOnModelChecker property specifies when to stop code generation, based on checks. The possible values are as follows:

- Errors - Stop code generation when errors are found prior to code generation. This is the default value.
- Warnings - Stop code generation when warnings or errors are found prior to code generation.

Most warnings are not started prior to code generation; the few that are started are related to problems that can result in incorrect behavior at run time.

*(Default = Errors)*

## BuildErrorHandling

The BuildErrorHandling property allows forcing Rational Rhapsody to treat all compiler errors as Code errors (versus element errors).

You can select the following choices from the drop-down list:

- "Model" means highlighting a model element on double-click compilation error on build.
- "Code" means highlighting a line in the code editor on double-click compilation error on build.

Note that "Model" sometimes highlights code lines if the model element cannot be found.

*Default = Model*

## EnableProgressDialog

EnableProgressDialog is a Boolean property that allows you to specify whether or not Rational Rhapsody should display a progress window while code is being generated.

*Default = Cleared*

## IncrementalCodeGenAcrossSession

The IncrementalCodeGenAcrossSession property is a Boolean value that controls the scope of the incremental code generation for the Rational Rhapsody session (either between sessions or only within the session).

If you set this to Cleared at the project level, code generation time stamps are not stored in the repository, and incremental code generation works only within a session.

*(Default = Checked)*

## ParallelCodeGeneration

Rational Rhapsody's default behavior is to use parallel processing in order to improve code generation performance. The ParallelCodeGeneration property is used to activate this feature and specify the criterion for determining how many parallel processes should be launched. Note that regardless of the value of this property, parallel processing is used only when you select the "with dependencies" or "entire project" code generation options.

The possible values are:

- BasedOnNumberOfCores - the number of processes launched matches the number of cores the computer has
- UserDefinedNumberOfProcesses - the number of processes is determined by the user by providing a

value for the property UserDefinedParallelProcesses

- Disabled - parallel processing is not used

*Default = BasedOnNumberOfCores*

## ParallelCodeGenerationCommand

The ParallelCodeGenerationCommand property specifies the command that should be carried out for code generation when parallel code generation is used. (Activation of parallel code generation is controlled by the property ParallelCodeGeneration.)

When parallel code generation is enabled, Rational Rhapsody launches multiple RhapsodyCL processes, and specifies which component should be handled by each such code generation process.

The ParallelCodeGenerationCommand property is provided to allow you to distribute these processes as you see fit. For example, you may want to have each such process run on a separate computer.

If you set the value of this property to anything other than RhapsodyCL.exe, Rational Rhapsody will repeatedly call the script you specified rather than calling RhapsodyCL. The arguments ordinarily sent to RhapsodyCL will instead be sent to your script.

This means that in addition to any code for distributing calls to RhapsodyCL, your script must handle the arguments received and pass them on to the RhapsodyCL call in the script.

*Default = RhapsodyCL.exe*

## ReportToOutputWindow

This property disables some messages printed to the Output window during code generation. This property can improve performance during code generation particularly on Linux systems if the Output window is slow.

The values are Basic and Detailed. Detailed prints all messages, and Basic does not print the "Generating..." messages.

*(Default = Detailed)*

## ShowLogViewAfterBuild

The ShowLogViewAfterBuild property allows you to specify that the Log tab should be brought to the front of the Output window after the completion of a build action, rather than the Build tab.

If this property is set to Checked, the Log tab is shown for all environments regardless of the value set for the property [lang]\_CG::[environment]::UseNewBuildOutputWindow for the individual environments.

*Default = Cleared*

## UserDefinedParallelProcesses

If the ParallelCodeGeneration property is set to UserDefinedNumberOfProcesses, then the UserDefinedParallelProcesses property is used to specify how many parallel code generation processes should be launched.

*Default = 0*

## Generalization

The Generalization metaclass contains a property used to support generalization. See the Rational Rhapsody help for more information on generalization.

### Generate

The Generate property specifies whether to generate code for a particular type of element.

*(Default = Checked)*

## ReportToOutputWindow

This property disables some messages printed to the Output window during code generation. This property can improve performance during code generation particularly on Linux systems if the Output window is slow.

The values are Basic and Detailed. Detailed prints all messages, and Basic does not print the "Generating..." messages.

*(Default = Detailed)*

## Operation

The Operation metaclass contains properties for implementing operations.

### Animate

The Animate property specifies whether animation code is generated for an element. You can specify your own animation function by using the property `<lang>__CG::Type::AnimSerializeOperation`.

The semantics of the Animate property is always in favor of the owner settings:



- If a package Animate property check box is cleared, all the classes owned by the package are not animated, regardless of the class Animate settings.
- If a class Animate property check box is cleared, all the elements in the class (attributes, operations, relations, and so on) are not animated.
- If an operation Animate property check box is cleared, all the arguments are not animated.
- If the AnimateArguments property check box is cleared, all the arguments are not animated, regardless of the specific argument Animate property settings.

*Default = Checked*

## AnimateArguments

The AnimateArguments property specifies whether to animate arguments to events or operations in animated sequence diagrams.

During instrumentation, Rational Rhapsody displays the values of actual parameters passed as arguments to events or operations in animated sequence diagrams by using the C++ stream output operator.

For this to work, the argument must be of a built-in type that can be output to a stream, such as an int or a char\*.

If, however, the argument is of a complex, user-defined type that the compiler cannot serialize, then trying to animate it causes compilation errors.

In VxWorks, trying to animate arguments of type long unsigned int also causes errors.

If you want to animate arguments of user-defined types defined either inside or outside of Rational Rhapsody, you must do one of the following:

- Add to the framework an overloaded C++ stream output operator for the type, such as: ostream operator(sometype);
- Instantiate the template string2X(T t) with the type.
- Disable the AnimateArguments property by setting it to Cleared.

*(Default = Checked)*

## Concurrency

The Concurrency property specifies the concurrency of a class. The CG::Operation::Concurrency property lets you specify that an operation should be protected by the object mutex (that is, only one access to the operation is allowed at any specific time).

When a class has one or more operations with concurrency guarded, the class becomes a guarded class.

The possible values for CG::Class::Concurrency are as follows:

- sequential - A sequential class maintains the single-threaded sequential protocol.
- active - An active class creates its own thread around which its operations are executed.

The possible values for `CG::Operation::Concurrency` are as follows:

- sequential - The operation is not guarded; access is sequential.
- guarded - The operation is protected by the object mutex.

Note that the `Concurrency` property cannot be used to guard static operations or the constructor of a class.

*Default = sequential*

## EnableInMethodBroker

The `EnableInMethodBroker` property specifies whether the operation should be considered in or excluded from the TestConductor MethodBroker.

Rational Rhapsody does not allow you to call operations from outside of your application during model execution because it assumes that an application is triggered by external events or signals.

If you want to test subsystems or classes that do not react to external events, you must build a test environment in your model. A MethodBroker receives events and calls operations that need to be driven.

Rational Rhapsody animation shows these operation calls as coming from the system border.

See the TestConductor documentation for more information about the MethodBroker.

You can exclude all operations of a class or package from the MethodBroker by setting the `EnableInMethodBroker` property at the class or package level.

*(Default = Checked)*

## Generate

The `Generate` property allows you to specify what type of code should be generated for the operation. The possible values are:

- Full - Rhapsody generates both specification and implementation code
- Specification - Rhapsody generates only the specification code
- None - Rhapsody does not generate any code for the element

Note that if you reverse engineer code that contains only specification code for an operation, the value of this property changes to `Specification` for that operation.

*Default = Full*

## TriggeredOperationDefaultReturnValue

The `TriggeredOperationDefaultReturnValue` property allows you to define a default return value for a triggered operation. This default value is returned by the operation if, for some reason, the user code that sets the return value is not run.

*Default = Blank*

## UseDefaultAttributeValues

The UseDefaultAttributeValues property is a Boolean value that specifies whether a constructor should initialize attributes according to the default value for the attribute.

Set this property to Cleared to make a constructor ignore the default value for the attribute.

*(Default = Checked)*

## VariableLengthArgumentList

The VariableLengthArgumentList property specifies whether a variable length argument list is to be added to the argument list for an operation.

If this is checked, a variable length argument list is added as the last argument for the operation.

For example, if the operation void f(int i) has this property set to checked, its generated declaration is void f(int i, ...).

*(Default = Cleared)*

# Package

The Package metaclass contains properties for implementing packages.

## AdditionalInitialization

The AdditionalInitializationCode property is a multiline value that specifies additional initialization code.

*(Default = empty MultiLine)*

## CallUserInitRelations

The CallUserInitRelations property is a Boolean value that determines whether to call a user-defined initRelations() method.

In Rational Rhapsody, initRelations() is called by the generated code (in the package constructors) even if you created your own initRelations() method. Set this property to Cleared to disable the call to the user-defined initRelations().

*(Default = Checked)*

## FileName

The FileName property specifies the name of the file to which code is generated for a class or package. This property uses a different name than the actual element name. For example, this feature can be of benefit if the class or package name is too long on 8.3 file systems.

The file name string can be either:

- A file name of your choice (including spaces)
- \$name:n, where n is any digit between 0 and 9

The variable \$name:n uses the name of the element as the basis for generating file names. The :n modifier specifies the length of the name. For example, if you specify \$name:8, a class named YosemiteNationalPark would be generated into the files Yosemite.cpp and Yosemite.h. If you use \$name:0, then no package file is generated (file name length = 0). Note also that when the CPP\_CG::Package::GenerateDirectory property is set to Cleared, then no additional package directory is generated.

If the FileName property is blank and the CG::Environment::IsFileNameShort property is set to Checked, the code generator creates 8.3 file names by truncating the class name.

If the FileName property is defined as a file name longer than eight characters and the IsFileNameShort property is set to Checked, the Checker reports an error. The long file name must be fixed prior to code generation. Each file name must be unique within the context of a single configuration.

If more than one class or package are generated to the same file name, they overwrite each other, causing compilation errors. To prevent this, the Checker determines whether multiple packages are directed to the same file before code is generated.

Setting the FileName property for an external base class (for example, to "BaseClass" without the ".h") generates an #include of the base class in the specification file of the subclass.

You use the UseAsExternal property to mark a class as an external reference class.

If the FileName property is not defined, you must add the appropriate code to the SpecificationProlog property for the subclass.

*Default = Empty string*

## GenerateImplicitDependencies

When code is generated, the file that is generated to represent a package automatically contains certain #include directives for elements such as:

- the parent of the package
- the classes in the package
- the types for any variables in the package

If you set the value of GenerateImplicitDependencies to False, these #include directives will not be automatically generated in the package file.

When the property is set to False, you can add dependencies with the Usage stereotype to have such #include directives generated selectively.

*Default = True*

## GeneratePackageCleanup

The GeneratePackageCleanup property determines when the system should perform a clean-up operation after generating a Package. The possible values are Never and Smart.

## GeneratePackageCode

The GeneratePackageCode property specifies whether to generate package code. This property supports “smart generation” of the package files. This property is ignored for COM/CORBA and Animation. The possible values are as follows:

- Always - Always generate package files.
- Never - Never generate package files.
- Smart - Generate package files only when the package contains elements that produce meaningful code.

*(Default = Smart)*

## GeneratePackageInitialization

The GeneratePackageInitialization property determines when the system should generate package initialization code. The possible values are:

- Never - the package initialization code is never generated
- Smart - Rational Rhapsody generates the package initialization code when it is required. The decision whether to generate this code is based on a number of factors, such as the values of the properties CG::Component::InitializationScheme and CG::Package::GeneratePackageCode, and whether the package contains a part that is a composite part.

*Default = Smart*

## GenerateWithAggregates

The GenerateWithAggregates property determines whether all classes owned by a package are added along with the package when you add a new package to a particular scope.

*(Default = Checked)*

## ImplicitDependencyToPackage

The ImplicitDependency propertyToPackage allows you to suppress the generation of an #include to a

parent package of a package. If set to False, Rational Rhapsody does not generate an #include to the parent package.

*Default = Checked*

## InitCleanUpRelations

The InitCleanUpRelations property specifies whether to generate initRelations() and cleanUpRelations() operations for sets of related global instances. This property applies only to composites and global relations.

*(Default = Checked)*

## InstancesAutoArrange

*The InstancesAutoArrange property is a Boolean UI helper property. You should not modify this property directly. (Default = Checked)*

## SelfInit

*The SelfInit property specifies whether to automatically initialize global instance variables. (Default = Cleared)*

## SynthesizeClassDependencies

The SynthesizeClassDependencies property controls when Rational Rhapsody generates forward declarations and include statements. The possible values are as follows:

- All - By default, Rational Rhapsody generates forward declarations to the classes owned by a package in the package specification file, and generates the includes in the package implementation file.
- ByUsage - Rational Rhapsody generates forward declarations and include files only for the classes to which the package has an explicit dependency (with the Usage stereotype).

*(Default = All)*

## UseAsExternal

The UseAsExternal property specifies whether an object is referenced as an external object (one that was not generated in Rational Rhapsody). This property references an external object in the model without generating code for it.

To prevent compilation errors with instrumentation when inheriting from an external base class (in C++), set the UseAsExternal property for the external base class to checked. This prevents serialization code from being added to the base class.

Similarly, the serialization operations are not called in the subclass. Setting the FileName property for the

base class (for example, to "BaseClass," without the ".h") generates an #include of the base class in the specification file of the subclass.

If the FileName property is not defined, you must add the appropriate code to the SpecificationProlog property for the subclass. You can also use the UseAsExternal property to create template (parameterized) classes as follows:

- Create the template class outside of Rational Rhapsody.
- In the Rational Rhapsody model, define a template class (for example, DataStore) as a placeholder for the external code defining the class.
- Set the UseAsExternal property for DataStore in the Rational Rhapsody model to checked.

Setting UseAsExternal to checked for the class prevents code from being generated for the external class when you generate code. You can still use the class in an object model diagram and inherit from it.

*If you want to inherit from an external template class, you can specify the external base class in the SpecificationProlog property for the subclass. Note that when you set this property at the package level, all the aggregates automatically become external as well. (Default = Cleared)*

## Relation

The Relation metaclass contains properties for implementing relations.

### Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter: \$cname-push\_back(\$item)

For maps (qualified relations), the following command inserts an item into the map based on a key:

*\$cname-insert(map\$keyType,\$target\*::value\_type( \$keyName,\$item))(Default = add\$cname:c)*

### AddComponentHelpersGenerate

The AddComponentHelpersGenerate property is a Boolean value that specifies whether to generate the helper method for a symmetric composite-part relationship, where the part multiplicity is greater than 1.

This property controls the helper method (such as \_addItsX()), which is used to connect a part X to its composite.

*(Default = Checked)*

### AddGenerate

The `AddGenerate` property specifies whether to generate an `Add()` operation for relations. Setting this property to `Cleared` is one way to optimize your code for size.

*(Default = Checked)*

## AddHelpersGenerate

The `AddHelpersGenerate` property is an enumerated type that controls the relation helper methods, such as `_addItsX()` and `__addItsX()`.

The possible values are as follows:

- `True` - Generate the helpers whenever code generation analysis determines that the methods are needed.
- `False` - Never generate the helpers.
- `FromModifier` - Generate the helpers based on the value of the `CG::Relation::AddGenerate` property.

*(Default = True)*

## Clear

The `Clear` property specifies the name of an operation that removes all items from a relation.

*For example, by using the boilerplate `clear$name:c` for a relation called `itsServer`, Rational Rhapsody would generate a public operation with the following signature: `void clearItsServer();` (Default = `clear$name:c`)*

## ClearGenerate

*The `ClearGenerate` property specifies whether to generate a `Clear()` operation for relations. Setting this property to `Cleared` is one way to optimize your code for size. (Default = Checked)*

## ClearHelpersGenerate

The `ClearHelpersGenerate` property is an enumerated type that controls the relation helper methods, such as `_clearItsX()` and `__clearItsX()`. The possible values are as follows:

- `True` - Generate the helpers whenever code generation analysis determines that the methods are needed.
- `False` - Never generate the helpers.
- `FromModifier` - Generate the helpers based on the value of the `CG::Relation::ClearGenerate` property.

*(Default = True)*

## Containment

The `Containment` property specifies how to represent relational objects as members of classes. The possible values are as follows:



- Value - The container actually lives inside the class. For example: `OMCollectionServer* itsServer;`
- Reference - The class maintains a pointer to the collection. For example: `OMCollectionServer** itsServer;`

*See also the `ContainerType::RelationType::CreateStatic` and `ContainerType::RelationType::InitStatic` properties. (Default = Value)*

## CreateComponent

The `CreateComponent` property specifies the name of an operation that creates a new component in a composite class.

For example, by using the boilerplate `new$cname:c` for a composite class with a component called `Server`, Rational Rhapsody generates a public operation with the following signature:

```
Server* newSv();
```

*(Default = new\$cname:c)*

## CreateComponentGenerate

The `CreateComponentGenerate` property specifies whether to generate a `CreateComponent` operation for composite objects. Setting this property to `Cleared` is one way to optimize your code for size.

*(Default = Checked)*

## CreateComponentUsingIndex

This property is included in the compatibility profile for version 6.1 MR-1 of Rational Rhapsody (`CGCompatibilityPre61M1[lang]`).

In version 6.1 MR-1 of Rational Rhapsody, improvements were made to the initialization code that is generated for parts with bounded multiplicity implemented as `StaticArray` (for example, `C* itsC[5]`).

These improvements avoid the redundant search for free locations in the array inside the composite create operation (for example, `newItsC()`). This is done by passing the index to the create operation from the external loop in `initRelations()`.

Since this represented a change to the create operation signature and behavior (for example, `newItsC()` replaced by `newItsC(int i)`), the change was disabled for pre-6.1 MR-1 models. This was accomplished by setting the value of the `CG::Relation::CreateComponentUsingIndex` property to `False`.

The property applies to `RiC`, `RiC++`, and `RiJ`.

## DeleteComponent

The `DeleteComponent` property specifies the name of an operation that deletes a component from a

composite class.

For example, by using the boilerplate `delete$name:c` for a composite class with a component called `Server`, Rational Rhapsody generates a public operation with the following signature:

```
void deleteSv(Server* p_Server);
```

*(Default = delete\$name:c)*

## DeleteComponentGenerate

The `DeleteComponentGenerate` property specifies whether to generate a `DeleteComponent()` operation for composite objects. Setting this property to `Cleared` is one way to optimize your code for size.

*(Default = Checked)*

## Dependency

The `Dependency` property specifies where to include the specification file (.h) of the dependent class in a relationship between two classes. The possible values are as follows:

- **Weak** - The specification file of the dependent class is included in the implementation file of the independent class with a forward declaration in the specification file.
- **Strong** - The specification file of the dependent class is included in the (.h) file of the independent class.

*(Default = Weak)*

## FilledDiamondInitializationScheme

The `FilledDiamondInitializationScheme` specifies how a filled-diamond relation is initialized. The possible values are as follows:

- **Automatic** - from the default
- **ByUser** - manually by the user

*(Default = ByUser)*

## FilledDiamondScheme

The `FilledDiamondScheme` property specifies how a filled-diamond relation is implemented. The possible values are as follows:

- **Composition** - Implement the relation as a composition. The automatically generated code creates and destroys the relation instances.
- **Aggregation** - Implement the relation as an aggregation. You are responsible for creating and deleting the relation elements.

For models that were created in versions prior to Rational Rhapsody 4.0, this property is automatically set to Aggregation when you load the model into Rational Rhapsody 4.0. This is due to the fact that only aggregation was supported in the previous versions of Rational Rhapsody.

*Default = Composition*

## Find

The Find property specifies the name of an operation that locates an item among relational objects.

For example, by using the boilerplate `find$name:c` for a relation called `itsServer`, Rational Rhapsody generates a public operation with the following signature:

```
int findItsServer(Server* p_Server)const;
```

*(Default = find\$name:c)*

## FindGenerate

*The FindGenerate property specifies whether to generate a Find() operation for relations. Setting this property to Cleared is one way to optimize your code for size. (Default = Cleared)*

## Generate

The Generate property specifies whether to generate code for a particular type of element. To optimize space, do not implement links to global or singleton objects by setting this property to Cleared.

*(Default = Checked)*

## GenerateRelationWithActors

The GenerateRelationWithActors property is an enumerated type that controls the generation of the dependency or relation to an actor (if it is a dependency or relation to an actor).

Control over generation of actors is done by the scope and by the appropriate check box in the Configuration Initialization tab.

The possible values for this property are as follows:

- Never - Never generate the dependency or relation.
- WhenActorIsGenerated - Generate the dependency or relation when the actor is generated.
- Always - Always generate the dependency or relation.

*(Default = WhenActorIsGenerated)*

## Get

The Get property specifies the name of an operation that retrieves the relation currently pointed to by the iterator.

For example, by using the boilerplate `get$name:c` for a relation called `itsServer`, Rational Rhapsody generates an operation with the following signature: `OMIteratorServer* getItsServer()const`; The Get property under a container metaclass (for example, under `RiCContainers::Scalar`) specifies the template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: `$me$name` The keyword `$me` is replaced with the object context variable as specified by the `Me` property. The variable `$name` is replaced with the name of the container, which is the role name for the relation.

*(Default = `get$name:c`)*

## GetEnd

The GetEnd property specifies the name of an operation that points the iterator to the last item in a collection. This property is used with the Standard Template Library.

For example, by using the boilerplate `get$name:cEnd` for a relation named `itsServer`, Rational Rhapsody generates a public operation with the following signature: `vectorServer*::const_iterator Server::getItsServerEnd() const`; This function points the iterator to the last item in the collection by testing for the following condition: `iter == getItsServerEnd()`

*(Default = `get$name:cEnd`)*

## GetEndGenerate

The GetEndGenerate property specifies whether to generate a `GetEnd()` operation for relations when you use STL. Setting this property to `Cleared` is one way to optimize your code for size.

*(Default = `Checked`)*

## GetGenerate

The GetGenerate property specifies whether to generate accessor operations for relations. Setting this property to `Cleared` is one way to optimize your code for size.

*(Default = `Checked`)*

## IgnoreQualifierOnBlackDiamond

If this the `IgnoreQualifierOnBlackDiamond` property is selected, it instructs the system to ignore the qualifier on a black diamond.

*(Default = `Cleared`)*

## Implementation

The Implementation property specifies how Rational Rhapsody should generate code for attributes or relations.

The various container property subjects, such as OMContainers, contain properties for generating code related to attributes and relations. When the value of the Implementation property is set to Default, Rational Rhapsody determines what category of properties to use (for example, Scalar or BoundedUnordered) on the basis of a number of criteria, including:

- the multiplicity specified
- whether or not the attribute was defined as ordered
- whether or not the attribute is of a predefined type

For example, if multiplicity is set to \* or 1..\*, code generation for the attribute or relation will be based on the values of the properties contained in the UnboundedUnordered metaclass. If the Ordered check box is also selected, then Rational Rhapsody will use the values of the properties in the UnboundedOrdered metaclass.

If you would prefer to manually specify what category of properties Rational Rhapsody should use, you can do so by selecting one of the other possible values for the Implementation property, for example, BoundedOrdered or Scalar.

The possible values for this property are:

- Default - Rational Rhapsody determines what category of properties to use
- Scalar - Implement a to-one relation as a pointer to a single object
- StaticArray - Implement a to-many relation as a static array of fixed size
- Fixed - Implement a to-many relation whose multiplicity is a number greater than 1, as a container of fixed size
- BoundedOrdered - Implement a to-many relation whose multiplicity is a number greater than 1 and whose "Ordered" check box is selected, as a list
- BoundedUnordered - Implement a to-many relation whose multiplicity is a number greater than 1 and whose "Ordered" check box is not selected, as a collection
- UnboundedOrdered - Implement a to-many relation with an upper limit of \* and whose "Ordered" check box is selected, as a list
- UnboundedUnordered - Implement a to-many relation with an upper limit of \* and whose "Ordered" check box is not selected, as either a list or a collection, depending on the language
- EmbeddedScalar - Implement a to-one composite relation as an embedded object (C and C++ only)
- EmbeddedFixed - Implement a to-many composite relation as an array of embedded objects (C and C++ only)
- User - This option instructs Rational Rhapsody to use the values you have provided for the properties under the "User" metaclass for the property subject that is relevant for you, for example OMContainers
- Qualified (for relations only) - Implement a to-many qualified relation as a map or hashtable, depending on the language

*Default = "Default"*

## InlineRelationForcesAnInclude

Prior to version 8.0.3 of Rational Rhapsody, when the value of the property `CPP_CG::Relation::Inline` was set to `in_header` or `in_declaration`, an `#include` directive was generated for the relation rather than a forward declaration. This issue was fixed in 8.0.3. To preserve the previous code generation behavior for pre-8.0.3 models, the `CG::Relation::InlineRelationForcesAnInclude` property was added to the backward compatibility settings for C++ with a value of `True`.

## InstanceBasedLinking

The `InstanceBasedLinking` property connects a relation with variant multiplicity between instances.

Rational Rhapsody can connect a relation between instances based on the object model diagram information. It connects instances based on the following principles:

- The instances to connect are owned by the same package or composite class.
- The multiplicity of the relation and instances match (that is, all the instances is connected based on the relation). As shown in the following figure, the match is calculated as  $n * M = N * M$ , where in a directional relation from A to B, N is set to 1.

In previous versions of Rational Rhapsody, the instances were connected only when M and N were constants. You can connect an instance even if the relation multiplicity is unconstrained (\*) or variant (such as 0..5).

Connecting instances over a relation with variant multiplicity is based on the following principles:

- The `InstanceBasedLinking` property check box is checked.
- If  $M = *$ , M is considered to have the same value as m.
- If  $M = x..y$ :
  - m must be greater than x.
  - M is considered to have the same value as  $\min(y, m)$ .

## IsConst

The `IsConst` property specifies whether accessor operations are const member functions. It can also be used to specify whether the return type of such an operation is a const. The possible values are as follows:

- None - not a const member function
- Signature - is a const member function
- SignatureAndReturnType - is a const member function, and the return type is a const

The default value is `Signature`.

## IsGuarded

The `IsGuarded` property specifies whether accessor and mutator operations are guarded. Guarded

operations block if the object is not in a state in which it can be executed.

The possible values are as follows:

- none - Neither accessors nor mutators are guarded.
- mutator - Only mutators are guarded.
- all - Both accessors and mutators are guarded.

*(Default = none)*

## Ordered

*The Ordered property specifies whether relations are ordered. (Default = Cleared)*

## Remove

The Remove property specifies the name of an operation that removes an item from a relation.

For example, by using the boilerplate `remove$name:c` for a relation called `itsServer`, Rational Rhapsody generates a public operation with the following signature: `void removeItsServer(Server* p_Server);`

*(Default = remove\$name:c)*

## RemoveComponentHelpersGenerate

The `RemoveComponentHelpersGenerate` property is a Boolean value that controls the relation helper methods, such as `_removeItsX()` and `__removeItsX()`.

*(Default = Checked)*

## RemoveGenerate

The `RemoveGenerate` property specifies whether to generate a `Remove()` operation for relations. Setting this property to `Cleared` is one way to optimize your code for size.

*(Default = Checked)*

## RemoveHelpersGenerate

The `RemoveHelpersGenerate` property is an enumerated type that controls the relation helper methods, such as `removeItsX()` and `__removeItsX()`.

The possible values are as follows:

- True - Generate the helpers whenever code generation analysis determines that the methods are needed.

- False - Never generate the helpers.
- FromModifier - Generate the helpers based on the value of the CG::Relation::RemoveGenerate property.

*(Default = True)*

## SafeInitScalar

The SafeInitScalar property specifies whether to initialize scalar relations as null pointers.

Consider the following C example: For a relation between an object A and a single object B, the pointer to itsB is initialized to NULL in the initializer for A, as follows:

```
A_Init() { me-itsB = NULL; initRelations(me); }
```

*(Default = Checked)*

## Set

The Set property specifies the name of the mutator generated for scalar relations.

For example, by using the template set\$name:c for a scalar relation called itsServer, Rational Rhapsody generates a public operation with the following signature:

```
void setItsServer(Server* p_Server);
```

*(Default = set\$name:c)*

## SetComponentHelpersGenerate

The SetComponentHelpersGenerate property is a Boolean value that controls the relation helper methods, such as \_setItsX() and \_\_setItsX().

*(Default = Checked)*

## SetGenerate

The SetGenerate property specifies whether to generate mutators for relations. Setting this property to Cleared is one way to optimize your code for size.

*(Default = Checked)*

## SetHelpersGenerate

The SetHelpersGenerate property is an enumerated type that controls the relation helper methods, such as \_setItsX() and \_\_setItsX().



The possible values are as follows:

- True - Generate the helpers whenever code generation analysis determines that the methods are needed.
- False - Never generate the helpers.
- FromModifier - Generate the helpers based on the value of the `CG::Relation::SetGenerate` property.

*(Default = True)*

## UsePre806AssociationCG

Prior to release 8.0.6, if you specified code to initialize an association in the initializer of the constructor, the body of the constructor would still assign a value of NULL to the association. This issue was fixed in 8.0.6.

To preserve the previous code generation behavior for pre-8.0.6 models, the `CG::Relation::UsePre806AssociationCG` property was added to the backward compatibility settings for C++ with a value of True.

## UsePre82RelationsCG

Prior to release 8.2, certain code for connecting links was generated in the `[package name]_initRelations` operation. In order to ensure that this code is run even when the `initRelations` operation is not used, in 8.2 this code was moved to the `[package name]_OMConnectRelations` operation.

To preserve the previous code generation behavior for pre-8.2 models, the property `CG::Relation::UsePre82RelationsCG` was added to the backward compatibility settings for C, C++, and Java with a value of True.

## Statechart

The Statechart metaclass contains a property to control statecharts.

## EventConsumptionScheme

The `EventConsumptionScheme` property controls whether multiple exits from `state_dispatchEvent` operations are allowed. An exit is needed only for AND state `dispatchEvent` functions.

This property is required for compliance with MISRA (Motor Industry Software Reliability Association) rules.

The possible values are as follows:

- Default - Multiple exits are allowed. This is normal pre-V4.2 behavior.
- SingleExitPoint - There is a single exit point.

For example, the following code shows the normal pre-V4.2 behavior (Default):

```
if(operation_dispatchEvent(me, id) 0) { res = eventConsumed; if(!IS_IN(me, Dishwasher_active)) return res; } if(service_dispatchEvent(me, id) 0) { res = eventConsumed; if(!IS_IN(me, Dishwasher_active)) return res; } return res; The following example shows the code that is generated when this property is set to SingleExitPoint: RiCBoolean dispatchDone = FALSE; if (operation_dispatchEvent(me, id) 0) { res = eventConsumed; if(!IS_IN(me, Dishwasher_active)) { dispatchDone = TRUE; } } if (!dispatchDone) { if (service_dispatchEvent(me, id) 0) { res = eventConsumed; if(!IS_IN(me, Dishwasher_active)) { dispatchDone = TRUE; } } } return res;
```

## FlatStateType

The FlatStateType property specifies another data type to use when defining attributes for flat statecharts.

*By default, Rational Rhapsody creates these attributes by using int values. You can set this property at or above the statechart level. (Default = empty string)*

## GenerateActionsAfterJunctionForDefaultTransition

Prior to version 7.6.1 of Rational Rhapsody, a bug prevented code from being generated for transition actions between a junction connector and the next state if one of the transitions entering the junction connector was the default transition.

This problem was corrected in version 7.6.1.

To preserve the previous code generation behavior for pre-7.6.1 models, the CG::Statechart::GenerateActionsAfterJunctionForDefaultTransition property was added to the backward compatibility settings for C++ and Java with a value of False.

## GenerateMisraCompliantEventProcessingCode

In version 8.0, changes were made to the statechart event-processing code in order to increase the compliance of the code with MISRA. To preserve the previous code generation behavior for pre-8.0 models, the CG::Statechart::GenerateMisraCompliantEventProcessingCode property was added to the backward compatibility settings for C and C++ with a value of False.

## IgnoreGuardsAfterJunctionConnector

Prior to version 8.0, in cases where a single trigger triggered multiple transitions from the same state, the code that was generated ignored any guards that appeared after a junction connector that followed any of the transitions. This problem has been corrected in version 8.0. To preserve the previous code generation behavior for pre-8.0 models, the CG::Statechart::IgnoreGuardsAfterJunctionConnector property was added to the backward compatibility settings for C, C++, and Java with a value of True.

## LocalTerminationSemantics

The LocalTerminationSemantics property specifies whether activity diagram mode is enabled for local termination of behavior.

Local termination refers to the termination of an activity rather than an instance when a Termination connector is reached in an activity diagram or statechart.

There are two termination types:

- Local termination - When the Termination connector is inside a composite state, it is considered a Final state, which terminates the activity represented by the composite state, but not the instance performing the activity.
- There are two types of local termination:
- Statechart mode - Local termination semantics can be applied only to an Or state that has a Final state as a substate.
- Activity diagram mode - Local termination semantics can be applied to any Or state, even one without a Final state.
- Global termination - When the Termination connector is inside the top state, it is considered a Termination state, which terminates the state machine, causing the instance to be destroyed.

*If this property is Cleared, statechart mode is enabled for local termination. If it is checked, activity diagram mode is enabled for local termination. (Default = Cleared)*

## PerformPostJunctionActionIfJunctionFollowsCondition

Prior to version 8.0, in cases where a junction connector followed a condition that exited a condition connector, the code that was generated for the else part of the condition also included the action that followed the junction connector. This problem has been corrected in version 8.0. To preserve the previous code generation behavior for pre-8.0 models, the

CG::Statechart::PerformPostJunctionActionIfJunctionFollowsCondition property was added to the backward compatibility settings for C, C++, and Java with a value of True.

## RootPrefixInStateAnnotations

Prior to version 8.0, certain annotations in statechart code included the string ROOT as a prefix before the state name. Beginning in version 8.0, the state name in the annotation matches the state name in the code. To preserve the previous code generation behavior for pre-8.0 models, the

CG::Statechart::RootPrefixInStateAnnotations property was added to the backward compatibility settings for C, C++, and Java with a value of True.

## StateInOperationReturnType

When code is generated for statecharts, a number of operations are generated for testing whether the system is in a given state. By default, these operations return an int in RiC and a bool in RiC++. The property StateInOperationReturnType allows you to specify a different return type for these operations. This can be used to ensure that the generated code meets standards such as MISRA.

*Default = Blank*

## TimeoutsOptimizationPolicy

The `TimeoutsOptimizationPolicy` property controls whether to reduce the number of timeout pointers in the code and object size by "merging" timeouts with the same parent.

- `None` - Select this choice when you do not need to control the number of timeout pointers in the code and object size.
- `PerParent` - Select this choice when you want to reduce the number of timeout pointers in the code and object size by "merging" timeouts with the same parent.

*Default = None*

## UsePre80StatechartCG

In version 8.0, the following changes were made in the code generated for statecharts:

- Previously, in cases where orthogonal states exited to a join connector that was followed by a termination connector, the code that was generated always used the exit action of the first state regardless of which of the orthogonal states the system was in. This problem was corrected in version 8.0.
- Previously, in cases where a single trigger triggered multiple transitions, the code that was generated ignored any guards that appeared after a junction connector that followed any of the transitions. This problem was corrected in version 8.0.
- Previously, in cases where a junction connector followed a condition that exited a condition connector, the code that was generated for the else part of the condition also included the action that followed the junction connector. This problem was corrected in version 8.0.
- Previously, in cases where two states had transitions entering a join sync bar, and one of the states had a second transition that used a guard, the system proceeded past the join sync bar without waiting for the completion of the transition from the other state. This problem was corrected in version 8.0.

To preserve the previous code generation behavior for pre-8.0 models, the `CG::Statechart::UsePre80StatechartCG` property was added to the backward compatibility settings for C, C++, and Java with a value of `True`.

## UseSingleExitActionIfTerminationFollowsJoin

Prior to version 8.0, in cases where orthogonal states exited to a join connector that was followed by a termination connector, the code that was generated always used the exit action of the first state regardless of which of the orthogonal states the system was in. This problem has been corrected in version 8.0. To preserve the previous code generation behavior for pre-8.0 models, the `CG::Statechart::UseSingleExitActionIfTerminationFollowsJoin` property was added to the backward compatibility settings for C, C++, and Java with a value of `True`.

## Type

The `Type` metaclass contains properties that control data types.

## Animate

The Animate property specifies whether animation code is generated for an element. You can specify your own animation function by using the `CPP_CG::Type::AnimSerializeOperation` property.

The semantics of the Animate property is always in favor of the owner settings:

- If a package Animate property check box is cleared, all the classes owned by the package are not animated, regardless of the class Animate settings.
- If a class Animate property check box is cleared, all the elements in the class (attributes, operations, relations, and so on) are not animated.
- If an operation Animate property check box is cleared, all the arguments are not animated.
- If the AnimateArguments property check box is cleared, all the arguments are not animated, regardless of the specific argument Animate property settings.

For most of the Animate properties, the possible values are checked and Cleared. However, the `CG::Type::Animate` property has three possible values:

- True - The code generator analyzes the data type and instruments it according to its type. If the type is unknown, its address is converted to `void*`.
- False - Disable the generation of animation calls.
- Force - Generate animation calls. The code generator generates either standard (`x2String` or `string2X`) calls, or those calls defined in the `AnimSerializeOperation` and `AnimUnserializeOperation` properties.

If you define a type that cannot be instrumented, you should declare the name for the instrumentation operation and create the operations manually, or define properties to allow generation of these operations.

*(Default = Force)*

## AttributesAutoArrange

The `AttributesAutoArrange` property controls the order of struct and union members in the code generated for your model. When set to False, the order of the members in the code is the same as the order displayed on the Attributes tab of the features dialog. When set to True, members will appear in alphabetic order in the code regardless of the order displayed on the Attributes tab. This property can be set from the project level downward.

*Default = Cleared*

## EnumerationAsTypedef

*The `EnumerationAsTypedef` property is a Boolean value that determines whether enumeration composite types are printed with typedef statements. (Default = Checked)*

## Generate

*The `Generate` property specifies whether to generate code for a particular type of element. (Default =*

*Checked)*

## GenerateDeclarationDependency

When you provide code for the declaration of a "Language" type, or specify the type of an element such as an attribute by providing your own code in the C++ Declaration field (rather than selecting a type from the list of existing types), Rational Rhapsody cannot know with 100% certainty what type your declaration is dependent upon.

In such cases, Rational Rhapsody parses the code you entered, and searches the model for the type referenced in the declaration. The product takes the first type it encounters with this name, and the code that is generated includes a dependency upon this type.

In cases where you have defined types with the same name in different packages, this type-searching behavior by Rational Rhapsody might result in `#include` statements that do not reflect what you had intended. To avoid this, you can set the value of the `GenerateDeclarationDependency` property to `False`, and Rational Rhapsody does not generate any dependency in the code to reflect your declaration. You can then explicitly create the correct dependency in the model so that the generated code includes all the necessary dependencies.

For example:

If you have defined an enum called "color" in the package `Basic_types`, and then have a package called `Vehicles` with a class named `Car`, and that class contains an attribute named `exterior_color` of type "color", Rational Rhapsody by default generates the necessary `#include` in the code for class `Car`:

```
#include "Basic_types.h"
```

If, however, you set the value of the `GenerateDeclarationDependency` property to `False`, the generated code does not include the necessary dependency and you have to provide it directly.

Keep in mind that the property is displayed under the metaclass `Type`, but in terms of code generation it affects the code that is generated for specific attributes that are of a given type. So you have to open the features window for the relevant attributes and change the value of the property there.

*Default = Checked*

## Implementation

The `Implementation` property specifies how Rational Rhapsody generates code for a given element (for example, as a simple array, collection, or list).

When this property is set to `Default` and the multiplicity is bounded (not `*`) and the type of the attribute is not a class, code is generated without using the container properties (as in previous versions of Rational Rhapsody).

Note that Rational Rhapsody generates a single accessor and mutator for an attribute, as opposed to relations, which can have several accessors and mutators.

*(Default = Default)*

## UseAsExternal

The UseAsExternal property specifies whether an object is referenced as an external object (one that was not generated in Rational Rhapsody).

This property references an external object in the model without generating code for it.

To prevent compilation errors with instrumentation when inheriting from an external base class (in C++), set the UseAsExternal property for the external base class to checked.

This prevents serialization code from being added to the base class. Similarly, the serialization operations are not called in the subclass.

Setting the FileName property for the base class (for example, to "BaseClass," without the ".h") generates an #include of the base class in the specification file of the subclass.

If the FileName property is not defined, you must add the appropriate code to the SpecificationProlog property for the subclass.

You can also use the UseAsExternal property to create template (parameterized) classes as follows:

- Create the template class outside of Rational Rhapsody.
- In the Rational Rhapsody model, define a template class (for example, DataStore) as a placeholder for the external code defining the class.
- Set the UseAsExternal property for DataStore in the Rational Rhapsody model to checked.

Setting UseAsExternal to checked for the class prevents code from being generated for the external class when you generate code. You can still use the class in an object model diagram and inherit from it.

If you want to inherit from an external template class, you can specify the external base class in the SpecificationProlog property for the subclass.

*(Default = Cleared)*

## UseClassOrderSetting

Prior to release 7.5.3, you could control the order of struct and union members in generated code only by modifying the value of the CG::Class::AttributesAutoArrange property, which controlled both the order of class attributes and the order of struct and union members. In 7.5.3, the CG::Type::AttributesAutoArrange property was added to allow users to control the order of struct and union members in generated code regardless of the setting used for class attributes. The UseClassOrderSetting property was therefore added to the compatibility profile for 7.5.3 in order to provide the previous code generation behavior for older models. When UseClassOrderSetting is set to True, the order of struct and union members is controlled by the value of the CG::Class::AttributesAutoArrange property rather than the value of the CG::Type::AttributesAutoArrange property.

# COM

---

The COM subject contains metaclasses that contain properties for allowing mixed, distributed applications and objects to find and interact with each other over a network. For more information on COM properties, see the MSDN Online Library (<http://msdn.microsoft.com/library/>). The COM subject is available in Rational Rhapsody Developer for C++ only.

## Argument

The Argument metaclass contains properties that control COM arguments. Many of these properties are standard MSDN COM properties.

### AppendToClause

The AppendToClause property is an optional verbatim property that adds free text to be generated into the end of the description clause, before the closing bracket.

*(Default = empty string)*

### defaultvalue

The defaultvalue property is a standard MSDN COM property. This property enables specification of a default value for a typed optional parameter.

*(Default = empty string)*

### first\_is

*The first\_is property is a standard MSDN COM property. This property specifies the index of the first array element to be transmitted. (Default = empty string)*

### ignore

The ignore property is a standard MSDN COM property. This property designates that a pointer contained in a structure or union (and the object indicated by the pointer) is not transmitted, and is restricted to pointer members of structures or unions.

*(Default = Cleared)*

### iid\_is

The iid\_is property is a standard MSDN COM property. This property specifies the IID of the COM



interface pointed to by an interface pointer.

*(Default = empty string)*

## **last\_is**

The last\_is property is a standard MSDN COM property. This property specifies the index of the last array element to be transmitted. When the specified index is zero or negative, no array elements are transmitted.

*(Default = empty string)*

## **lcid**

The lcid property is a standard MSDN COM property. This property indicates that the parameter is a locale ID (LCID).

*(Default = Cleared)*

## **length\_is**

The length\_is property is a standard MSDN COM property. This property specifies the number of array elements to be transmitted. Specify a non-negative value.

*(Default = empty string)*

## **max\_is**

The max\_is property is a standard MSDN COM property. This property designates the maximum value for a valid array index.

*(Default = empty string)*

## **optional**

The optional property is a standard MSDN COM property. This property specifies an optional parameter.

*(Default = Cleared)*

## **pointer**

The pointer property is a standard MSDN COM property. Explicit pointer attributes are applied to the pointer at the definition or use site.

*(Default = empty string)*

## readonly

The readonly property is a standard MSDN COM property. This property specifies whether the parameter is read only.

*(Default = Cleared)*

## retval

*The retval property is a standard MSDN COM property. This property designates the parameter that receives the return value of the member. (Default = Cleared)*

## size\_is

The size\_is property is a standard MSDN COM property. This property specifies the size of memory allocated for sized pointers, sized pointers to sized pointers, and single- or multidimensional arrays.

*(Default = empty string)*

## string

The string property is a standard MSDN COM property. This property specifies a string.

*(Default = Cleared)*

# Attribute

The Attribute metaclass contains properties that control COM attributes.

## AppendToClause

The AppendToClause property is an optional verbatim property that adds free text to be generated into the end of the description clause, before the closing bracket.

*(Default = empty string)*

## bindable

The bindable property is a standard MSDN COM property. This property indicates that the property supports data binding.

*(Default = Cleared)*

## **call\_as**

The call\_as property is a standard MSDN COM property. This property enables mapping a function, which cannot be called remotely, to a remote function.

*(Default = empty string)*

## **defaultbind**

The defaultbind property is a standard MSDN COM property. This property indicates the single, bindable property that best represents the object.

*(Default = Cleared)*

## **defaultcollelm**

The defaultcollelm property is a standard MSDN COM property. This property allows for optimization of code.

*(Default = Cleared)*

## **helpcontext**

The helpcontext property is a standard MSDN COM property. This property sets the context in the help file.

*(Default = empty string)*

## **helpstring**

The helpstring property is a standard MSDN COM property. This property specifies the text that describes the element to which it applies, such as a library, dispinterface, or coclass.

*(Default = empty string)*

## **hidden**

The hidden property is a standard MSDN COM property. This property indicates that the item exists, but should not be displayed in a user-oriented browser.

*(Default = Cleared)*

## id

The id property specifies the ID used for get/put methods written to the COM interface file. Relations between COM interfaces are treated like COM interface attributes, and the required get/put methods are written to the COM interface file.

*(Default = empty string)*

## immediatebind

The immediatebind property is a standard MSDN COM property. This property allows individual, bindable, properties on a form to specify this behavior. When this bit is set, all changes is notified.

*(Default = Cleared)*

## implementation

The implementation property is an enumerated type that specifies which accessor and mutator methods (get/put/putref) should be generated for a COM attribute in the COM interface. The possible values are as follows:

- propget - Generate an accessor only.
- propput - Generate a mutator only.
- propputref - Generate a by-reference mutator only.
- propgetpropput - Generate both an accessor and a mutator (the default value).
- propgetpropputRef - Generate both an accessor and a by-reference mutator.

*(Default = propgetpropput)*

## local

The local property is a standard MSDN COM property. This property specifies to the MIDL compiler that an interface or a function is not remote.

*(Default = Cleared)*

## nonbrowsable

The nonbrowsable property is a standard MSDN COM property. This property indicates that the property is displayed in an object browser (which does not show property values), but does not appear in a properties browser (which does show property values).

*(Default = Cleared)*

## **requestedit**

The requestedit property is a standard MSDN COM property. This property indicates that the property supports the OnRequestEdit notification.

*(Default = Cleared)*

## **restricted**

The restricted property is a standard MSDN COM property. This property prevents the item from being used by a macro programmer.

*(Default = Cleared)*

## **string**

The string property is a standard MSDN COM property. This property specifies a string.

*(Default = empty string)*

## **uidefault**

The uidefault property is a standard MSDN COM property. This property returns or sets the UIDefault attribute of a member object. The UIDefault attribute indicates that the type information member is the default member for display in the user interface.

*(Default = Cleared)*

# **Class**

The Class metaclass contains properties that control how classes are mapped to code when you use COM.

## **In**

The In property specifies how code is generated when the type is used with an argument that has the modifier In. In the IDL, the class is: [in] class>\* In C++, it is realized as: class>\*

*(Default = \$type\*)*

## **InOut**

The InOut property specifies how code is generated when the type is used with an argument that has the modifier InOut. In the IDL, the class is:

[in,out] class>\*\* In C++, it is realized as: class>\*\*

(Default = \$type\*\*)

## Out

The Out property specifies how code is generated when the type is used with an argument that has the modifier Out. In the IDL, the class is: [out] class>\*\* In C++, it is realized as: class>\*\*

(Default = \$type\*\*)

## ReturnType

The ReturnType property specifies how code is generated when the type is used as a return type. In the IDL, the class is HRESULT.

(Default = \$type\*)

## TriggerArgument

*The TriggerArgument property specifies how the type should be passed in when used as an argument for events/triggered operations. By default, classes that are used as an argument are passed by way of a pointer. There are 4 other properties that effect how types are passed into and returned by regular operations: "in", "out", "in/out" and "return." (Default = \$type\*)*

## coclass

The coclass metaclass contains properties that affect COM coclasses.

## AppendToClause

The AppendToClause property is an optional verbatim property that adds free text to be generated into the end of the description clause, before the closing bracket.

(Default = empty string)

## appobject

The appobject property is a standard MSDN COM property. This property identifies the application object.

*(Default = Cleared)*

## **control**

The control property is a standard MSDN COM property. This property indicates that the item represents a control from which a container site derives additional type libraries or coclasses.

*(Default = Cleared)*

## **DefaultInterface**

The DefaultInterface property specifies the default interface that a «COM Coclass» class should expose. This property is blank by default. To override the property, assign it the name of the «COM Interface» class that you want a coclass to expose. If you set the DefaultInterface property while a package or component is selected, the property is automatically applied to all new «COM Coclass» classes that you create in that package or component.

*(Default = empty string)*

## **DefaultSourceInterface**

The DefaultSourceInterface property specifies the default source interface for a coclass.

*(Default = empty string)*

## **helpcontext**

The helpcontext property is a standard MSDN COM property. This property sets the context in the help file.

*(Default = empty string)*

## **helpstring**

The helpstring property is a standard MSDN COM property. This property specifies the text that describes the element to which it applies, such as a library, dispinterface, or coclass.

*(Default = empty string)*

## **hidden**

The hidden property is a standard MSDN COM property. This property indicates that the item exists, but should not be displayed in a user-oriented browser.

*(Default = Cleared)*

## licensed

The licensed property is a standard MSDN COM property. This property indicates that the class is licensed.

*(Default = Cleared)*

## noncreatable

The noncreatable property is a standard MSDN COM property. This property indicates that the class does not support creation at the top level (for example, through `TypeInfo::CreateInstance` or `CoCreateInstance`). An object of such a class is usually obtained through a method call on another object.

*(Default = Cleared)*

## uuid

The uuid property is a universal, unique identifier for a COM coclass (and other metaclasses). If this property is unspecified, Rational Rhapsody generates the value automatically.

*(Default = empty string)*

# Configuration

The Configuration metaclass contains properties that affect the COM configuration.

## COMClientStandardHeaders

The COMClientStandardHeaders property adds standard CPP includes for COM clients.

*(Default = "comdef.h")*

## COMEnable

The COMEnable property specifies whether Rational Rhapsody should generate COM client code. The possible values are as follows:

- Client - Generate COM client code.
- No - Do not generate COM client code.

*(Default = No)*



## COMInitialize

The COMInitialize property adds COM initialization code to the main() section for the client.

*(Default = CoInitialize(NULL);)*

## COMUnInitialize

The COMUnInitialize property adds COM uninitialization code to the main() section for the client.

*(Default = CoUninitialize(NULL);)*

## CreateInitialInstance

The CreateInitialInstance property is a template that creates an initial instance of a COM coclass for a COM client. The «COM TLB» stereotype is applied to components that are to be built into a TLB (a library of COM interfaces), and a ProxyStub.dll file. Running a make on such components runs first the Microsoft MIDL compiler and then the C++ compiler. The first (MIDL) phase of the make yields a TLB file. It also yields the sources for the ProxyStub.dll file, if the GenerateProxyStubDll property is set to True. In this case, the second (C++ compiler) phase compiles the sources yielded by the first phase into the ProxyStub.dll file. If the property is False, the ProxyStub.dll file is not built. A component with a «COM TLB» stereotype can have within its scope only the following elements:

- Packages stereotyped as «COM Library»
- Classes stereotyped as «COM Interface»
- Classes stereotyped as «COM Coclass»

The default value is as follows: IUnknownPtr m\_pUnk\$class(\_\_uuidof(\$class), NULL,CLSCTX\_ALL))

## DestroyInitialInstance

The DestroyInitialInstance property is a template for destroying the initial instance of a COM coclass.

*(Default = m\_pUnk\$class = NULL;)*

## GenerateProxyStubDll

The GenerateProxyStubDll property specifies whether the ProxyStub.dll file is built. See the CreateInitialInstance property for more information.

*(Default = Cleared)*

## MIDLCommand

The MIDLCommand property specifies the MIDL compiler command. The default value is as follows:

```
/* midl /tlb "$MIDLOutTypeLib" /h "$MIDLOutHeader" /iid "$MIDLOutIDFileName" /Oicf */
```

## ProxyStubCommand

The ProxyStubCommand property specifies the makefile command to create a ProxyStub DLL. The default value is as follows:

```
/* $ProxyStubDllName: dlldata.obj $ComLibraryPackage_p.obj $ComLibraryPackage_i.obj link /dll  
/out:$ProxyStubDllName / def:$ProxyStubDefFileName /entry:DllMain dlldata.obj  
$ComLibraryPackage_p.obj $ComLibraryPackage_i.obj \ kernel32.lib rpcndr.lib rpcns4.lib rpert4.lib  
oleaut32.lib uuid.lib .c.obj: cl /c /Ox /DWIN32 /D_WIN32_WINNT=0x0400 /  
DREGISTER_PROXY_DLL $ */
```

## ProxyStubDefFileName

The ProxyStubDefFileName property specifies the name of the .def file used to generate the ProxyStub. The default is \$componenttps.def, where \$component is replaced with the name of the server.

*(Default = \$componenttps.def)*

## ProxyStubDllName

The ProxyStubDllName property sets the name of the generated ProxyStub DLL.

*(Default = \$componenttps.dll)*

# IDL

The IDL metaclass contains properties that affect the COM IDL.

## IDL\_StandardImport

The IDL\_StandardImport property imports the user-specified IDL file into all generated IDL files. The default value is as follows:

```
/* import "oidl.idl"; import "ocidl.idl"; */
```

## IDL\_StandardInclude

The IDL\_StandardInclude property includes a specified IDL file into all generated IDL files.

*(Default = empty string)*

## **IDLExtension**

The IDLExtension property specifies the default extension for the IDL file.

*(Default = .idl)*

## **IDLImportFormat**

The IDLImportFormat property sets the format of an imported IDL file.

*(Default = import "\$FILENAME";)*

## **IDLIncludeFormat**

The IDLIncludeFormat property sets the format of an included IDL file.

*(Default = #include "\$FILENAME")*

## **InterfaceDeclaration**

The InterfaceDeclaration property is a template that specifies how the COM interface should be declared in the IDL file.

*(Default = interface \$interface;)*

## **MIDLOutHeader**

The MIDLOutHeader property sets the name of the header file generated by the MIDL compiler.

*(Default = \$Package.h)*

## **MIDLOutIDFileName**

The MIDLOutIDFileName property sets the name of the ID file generated by the MIDL compiler.

*(Default = \$Package.i.c)*

## **MIDLOutTypeLib**

The MIDLOutTypeLib property sets the name of the type library file generated by the MIDL compiler.

*(Default = \$Package\$TypeLibExtension)*

## TypeLib\_StandardImport

The TypeLib\_StandardImport property specifies the standard import files for a type library file. The default value is as follows:

```
importlib("stdole32.tlb"); importlib("stdole2.tlb");
```

## TypeLibExtension

The TypeLibExtension property specifies the extension for a type library file.

*(Default = .tlb)*

## TypeLibImportFormat

The TypeLibImportFormat property generates a .tlb extension for a file. You can add import statements manually by setting the TypeLibImportFormat property. The default value is as follows:

```
importlib("$FILENAME");
```

# Interface

The Interface metaclass contains properties that affect the COM interface.

## AppendToClause

The AppendToClause property is an optional verbatim property that adds free text to be generated into the end of the description clause, before the closing bracket.

*(Default = empty string)*

## ExternalInclude

The ExternalInclude property adds all external includes in the COM interface and COM library.

*(Default = empty MultiLine)*

## helpcontext

The helpcontext property is a standard MSDN COM property. This property sets the context in the help file.

*(Default = empty string)*

## **helpstring**

The helpstring property is a standard MSDN COM property. This property specifies the text that describes the element to which it applies, such as a library, dispinterface, or coclass.

*(Default = empty string)*

## **hidden**

The hidden property is a standard MSDN COM property. This property indicates that the item exists, but should not be displayed in a user-oriented browser.

*(Default = Cleared)*

## **local**

The local property is a standard MSDN COM property. This property specifies to the MIDL compiler that an interface or a function is not remote.

*(Default = Cleared)*

## **nonextensible**

The nonextensible property is a standard MSDN COM property. This property indicates that the IDispatch implementation includes only the properties and methods listed in the interface description.

*(Default = Cleared)*

## **object**

The object property is a standard MSDN COM property. This property identifies a COM interface.

*(Default = Checked)*

## **oleautomation**

The oleautomation property is a standard MSDN COM property. This property indicates that an interface is compatible with Automation.

*(Default = Cleared)*

## pointer\_default

The `pointer_default` property is a standard MSDN COM property. This property specifies the default pointer attribute for all pointers except top-level pointers that appear in parameter lists. The possible values are as follows:

- `unique` - Specifies a unique pointer
- `ptr` - Specifies a full pointer
- `ref` - Specifies a reference pointer

*(Default = unique)*

## replaceable

The `replaceable` property is a standard MSDN COM property.

*(Default = Cleared)*

## Type

The `uuid` property specifies the type of the COM interface. The possible values are as follows:

- `Dual` - An interface that exposes properties and methods through `IDispatch` and directly through the VTBL.
- `Custom` - A user-defined interface
- `dispinterface` - An interface that defines a set of properties and methods on which you can call `IDispatch::Invoke`

*(Default = Dual)*

## uuid

The `uuid` property is a universal, unique identifier for a COM coclass (and other metaclasses). If this property is unspecified, Rational Rhapsody generates the value automatically.

*(Default = empty string)*

## Library

The `Library` metaclass contains properties that affect COM libraries.

## AppendToClause

The AppendToClause property is an optional verbatim property that adds free text to be generated into the end of the description clause, before the closing bracket.

*(Default = empty string)*

## **control**

The control property is a standard MSDN COM property. This property indicates that the item represents a control from which a container site derives additional type libraries or coclasses.

*(Default = Cleared)*

## **ExternalInclude**

The ExternalInclude property adds all external includes in the COM interface and COM library.

*(Default = empty MultiLine)*

## **helpcontext**

*The helpcontext property is a standard MSDN COM property. This property sets the context in the help file. (Default = empty string)*

## **helpfile**

The helpfile property is a standard MSDN COM property. This property sets the name of the help file.

*(Default = empty string)*

## **helpstring**

The helpstring property is a standard MSDN COM property. This property specifies the text that describes the element to which it applies, such as a library, dispinterface, or coclass.

*(Default = empty string)*

## **helpstringdll**

The helpstringdll property is a standard MSDN COM property. This property sets the name of the DLL to use to perform the document string lookup (localization).

*(Default = empty string)*

## hidden

The hidden property is a standard MSDN COM property. This property indicates that the item exists, but should not be displayed in a user-oriented browser.

*(Default = Cleared)*

## IncludePath

The IncludePath property specifies the path for the COM library.

*(Default = empty string)*

## lcid

The lcid property is a standard MSDN COM property. This property indicates that the parameter is a locale ID (LCID).

*(Default = empty string)*

## restricted

The restricted property is a standard MSDN COM property. This property prevents the item from being used by a macro programmer.

*(Default = Cleared)*

## uuid

The uuid property is a universal, unique identifier for a COM coclass (and other metaclasses). If this property is unspecified, Rational Rhapsody generates the value automatically.

*(Default = empty string)*

## version

The version property specifies the version number of the COM Type Library.

*(Default = 1.0)*

# Operation



The Operation metaclass contains properties that affect COM operations.

## **AppendToClause**

The AppendToClause property is an optional verbatim property that adds free text to be generated into the end of the description clause, before the closing bracket.

*(Default = empty string)*

## **call\_as**

The call\_as property is a standard MSDN COM property. This property enables mapping a function, which cannot be called remotely, to a remote function.

*(Default = empty string)*

## **callback**

The callback property is a standard MSDN COM property. This property declares a static callback function that exists on the client side of the distributed application. Callback functions provide a way for the server to execute code on the client.

*(Default = Cleared)*

## **helpcontext**

The helpcontext property is a standard MSDN COM property. This property sets the context in the help file.

*(Default = empty string)*

## **helpstring**

The helpstring property is a standard MSDN COM property. This property specifies the text that describes the element to which it applies, such as a library, dispinterface, or coclass.

*(Default = empty string)*

## **hidden**

The hidden property is a standard MSDN COM property. This property indicates that the item exists, but should not be displayed in a user-oriented browser.

*(Default = Cleared)*

## **id**

The id property specifies the ID used for get/put methods written to the COM interface file. Relations between COM interfaces are treated like COM interface attributes, and the required get/put methods are written to the COM interface file.

*(Default = empty string)*

## **local**

The local property is a standard MSDN COM property. This property specifies to the MIDL compiler that an interface or a function is not remote.

*(Default = Cleared)*

## **restricted**

The restricted property is a standard MSDN COM property. This property prevents the item from being used by a macro programmer.

*(Default = Cleared)*

## **source**

The source property is a standard MSDN COM property. This property indicates that a member of a coclass, property, or method is a source of events. For a member of a coclass, this attribute means that the member is called rather than implemented.

*(Default = Cleared)*

## **vararg**

The vararg property is a standard MSDN COM property. This property specifies that the function takes a variable number of parameters. To accomplish this, the last parameter must be a safe array of VARIANT type that contains all the remaining parameters.

*(Default = Cleared)*

# **Relation**

The Relation metaclass contains a property that affects COM relations.

## **id**

The id property specifies the ID used for get/put methods written to the COM interface file. Relations between COM interfaces are treated like COM interface attributes, and the required get/put methods are written to the COM interface file.

*(Default = empty string)*

# Communication\_Diagram

---

The Communication\_Diagram subject contains metaclasses that contain properties that are used to control the appearance of elements in communication diagrams.

## AssociationRole

The AssociationRole metaclass contains a property that controls the display of associations in communication diagrams.

### color

*The color property specifies the default color of the border of a graphical item, such as an object box. (Default = 128,0,0)*

### line\_style

The line\_style property specifies the default line style for a graphical item.

The possible values are:

- straight\_arrows - straight line.
- rectilinear\_arrows - rectilinear line with right-angled corners placed at appropriate locations, depending on the starting and ending points of the line
- spline\_arrows - curved line without corners
- rounded\_rectilinear\_arrows - similar to ordinary rectilinear lines but use rounded bends rather than straight ninety-degree angles

*Default = spline\_arrows*

*Default = rectilinear\_arrows*

### line\_width

*The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines). (Default = 1)*

### name\_color

*The name\_color property specifies the default color of names of graphical items. (Default = 0,0,0)*

# AutoPopulate

The AutoPopulate metaclass contains properties that can be used to control the appearance of diagrams that are drawn automatically by Rational Rhapsody.

## ArrowDirection

The ArrowDirection property is used when Rational Rhapsody automatically generates a diagram, and it determines whether the flow of connectors in the diagram runs from top to bottom or bottom to top.

There are two situations where Rational Rhapsody automatically generates diagrams:

- If you have selected the Populate Diagrams option for Reverse Engineering (for those diagrams where this feature is supported).
- If you double-click a diagram in the browser that was generated by using the Rational Rhapsody API.

*Default = Bottom-Top*

# Classifier

The Classifier metaclass contains properties that control the display of classifiers in communication diagrams.

## color

*The color property specifies the default color of the border of a graphical item, such as an object box. (Default = 128,0,0)*

## Fillcolor

*The Fillcolor property specifies the default fill color for the object. (Default = 0,0,0)*

## line\_width

*The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines). (Default = 1)*

## name\_color

*The name\_color property specifies the default color of names of graphical items. (Default = 0,0,0)*

## ClassifierActor

The ClassifierActor metaclass contains properties that control the display of classifier actors in communication diagrams.

### color

*The color property specifies the default color of the border of a graphical item, such as an object box. (Default = 128,0,0)*

### Fillcolor

*The Fillcolor property specifies the default fill color for the object. (Default = 0,0,0)*

### line\_width

*The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines). (Default = 1)*

### name\_color

*The name\_color property specifies the default color of names of graphical items. (Default = 0,0,0)*

## CommunicationDiagramGE

The CollaborationDiagramGE metaclass contains a property that controls the display of communication diagrams in Rational Rhapsody.

### Fillcolor

The Fillcolor property specifies the default fill color for the object.

*Default = 218,218,218*

## CollMessage

The CollMessage metaclass contains properties that control the display of link messages in communication diagrams.

## **color**

*The color property specifies the default color of the border of a graphical item, such as an object box. (Default = 0,0,0)*

## **line\_width**

*The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines). (Default = 1)*

## **name\_color**

*The name\_color property specifies the default color of names of graphical items. (Default = 0,0,0)*

# **Comment**

The Comment metaclass contains properties that control the appearance of comments in communication diagrams.

## **color**

*The color property specifies the default color of the border of a graphical item, such as an object box. (Default = 128,128,0)*

## **Fillcolor**

*The Fillcolor property specifies the default fill color for the object. (Default = 0,255,255)*

## **line\_width**

*The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines). (Default = 1)*

## **name\_color**

*The name\_color property specifies the default color of names of graphical items. (Default = 0,0,0)*

## Complete

The metaclass Complete contains properties that determine whether or not Rational Rhapsody automatically draws the relations that exist between an element added to a diagram and elements already on the diagram.

### Complete\_Relation

The Complete\_Relation property is used to specify that when an element is added to a diagram, Rational Rhapsody should automatically draw the relations that exist between the element and elements already on the diagram.

*Default = Cleared*

## Constraint

The Constraint metaclass contains properties that control the appearance of comments in communication diagrams.

### color

*The color property specifies the default color of the border of a graphical item, such as an object box. (Default = 128,128,0)*

### Fillcolor

*The Fillcolor property specifies the default fill color for the object. (Default = 0,255,255)*

### line\_width

*The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines). (Default = 1)*

### name\_color

*The name\_color property specifies the default color of names of graphical items. (Default = 0,0,0)*



# Depends

The Depends metaclass contains properties that control the appearance of dependency relation lines in communication diagrams.

## color

*The color property specifies the default color of the border of a graphical item, such as an object box. (Default = 0,0,255)*

## line\_style

The line\_style property specifies the default line style for a graphical item.

The possible values are:

- straight\_arrows - straight line.
- rectilinear\_arrows - rectilinear line with right-angled corners placed at appropriate locations, depending on the starting and ending points of the line
- spline\_arrows - curved line without corners
- rounded\_rectilinear\_arrows - similar to ordinary rectilinear lines but use rounded bends rather than straight ninety-degree angles

*Default = spline\_arrows*

*Default = straight\_arrows*

## line\_width

*The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines). (Default = 1)*

## name\_color

*The name\_color property specifies the default color of names of graphical items. (Default = 0,0,255)*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).

- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = None*

## DiagramFrame

The DiagramFrame metaclass contains properties that control the appearance of a diagram frame.

### DefaultSize

The DefaultSize property specifies default values for diagram frame dimensions.

*Default = 20, 20, 590, 500*

### ShowName

The ShowName property determines the text to display next to a graphic element in a diagram. For most elements, you can provide a name and a label. Your descriptive label can be useful in cases where the name itself might not be sufficient due to various constraints, such as the inability to use spaces if the name of the element is to appear in the code.

The possible values for this property vary for the different elements, as does the default value used. The possible values are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is displayed with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

Note that for some elements, such as destruction elements in sequence diagrams and timing diagrams, the property is a boolean property that hides/shows the name of the element.

*Default = Name*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## General

Contains properties relating to the general appearance of communication diagrams.

### SysMLDiagramKindTitle

When a diagram is created in SysML projects, the diagram frame contains a header for the diagram that includes information such as the name of the package that contains the diagram. The property SysMLDiagramKindTitle is used to specify a prefix for the diagram frame header that reflects the type of diagram. For example, when you create an internal block diagram, the header in the diagram frame starts with the prefix "ibd".

Note that the structure of the entire header is controlled by the property General::Graphics::DiagramHeader.

## Messages

Contains properties that affect the display of messages in communication diagrams.

### ShowNumbering

ShowNumbering is a boolean property that determines whether or not Rational Rhapsody displays the sequence numbers for messages in a communication diagram.

*Default = Checked*

## MultiObj

The MultiObj metaclass contains properties that control the display of multiple objects in communication diagrams.

### color

*The color property specifies the default color of the border of a graphical item, such as an object box. (Default = 128,0,0)*

### Fillcolor

*The Fillcolor property specifies the default fill color for the object. (Default = 0,0,0)*

### line\_width

*The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines). (Default = 1)*

### name\_color

*The name\_color property specifies the default color of names of graphical items. (Default = 0,0,0)*

## Note

The Note metaclass contains properties that control the appearance of notes in communication diagrams.

### color

*The color property specifies the default color of the border of a graphical item, such as an object box. (Default = 0,128,64)*

### Fillcolor

*The Fillcolor property specifies the default fill color for the object. (Default = 0,255,255)*

### line\_width

*The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines). (Default = 1)*

## **name\_color**

*The name\_color property specifies the default color of names of graphical items. (Default = 0,128,255)*

# **Requirement**

The Requirement metaclass contains properties that control the appearance of requirements in communication diagrams.

## **color**

*The color property specifies the default color of the border of a graphical item, such as an object box. (Default = 128,128,0)*

## **Fillcolor**

*The Fillcolor property specifies the default fill color for the object. (Default = 0,255,255)*

## **line\_width**

*The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines). (Default = 1)*

## **name\_color**

*The name\_color property specifies the default color of names of graphical items. (Default = 0,0,0)*

# **ReverseCollMessage**

The ReverseCollMessage metaclass contains properties that control the display of reverse link messages in communication diagrams.

## **color**

*The color property specifies the default color of the border of a graphical item, such as an object box.*

*(Default = 0,0,0)*

## **line\_width**

*The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines). (Default = 1)*

## **name\_color**

*The name\_color property specifies the default color of names of graphical items. (Default = 0,0,0)*

## **ShowStereotype**

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*(Default = None)*

# ComponentDiagram

---

The ComponentDiagram subject contains metaclasses that contain properties that are used to control the appearance of elements in component diagrams.

## AutoPopulate

The AutoPopulate metaclass contains properties that can be used to control the appearance of diagrams that are drawn automatically by Rational Rhapsody.

### ArrowDirection

The ArrowDirection property is used when Rational Rhapsody automatically generates a diagram, and it determines whether the flow of connectors in the diagram runs from top to bottom or bottom to top.

There are two situations where Rational Rhapsody automatically generates diagrams:

- If you have selected the Populate Diagrams option for Reverse Engineering (for those diagrams where this feature is supported).
- If you double-click a diagram in the browser that was generated by using the Rational Rhapsody API.

*Default = Bottom-Top*

## Comment

The Comment metaclass contains properties that control the appearance of comments in component diagrams.

### color

*The color property specifies the default color of the border of a graphical item, such as an object box. (Default = 128,128,0)*

### FillColor

*The Fillcolor property specifies the default fill color for the object. (Default = 0,255,255)*

### line\_width

*The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action*

*state lines). (Default = 1)*

## **name\_color**

*The name\_color property specifies the default color of names of graphical items. (Default = 0,0,0)*

## **Complete**

The metaclass Complete contains properties that determine whether or not Rational Rhapsody automatically draws the relations that exist between an element added to a diagram and elements already on the diagram.

## **Complete\_Relation**

The Complete\_Relation property is used to specify that when an element is added to a diagram, Rational Rhapsody should automatically draw the relations that exist between the element and elements already on the diagram.

*Default = Cleared*

## **Component**

The Component metaclass contains properties that control how components are displayed in component diagrams.

## **color**

The color property specifies the default color of the border of a graphical item, such as an object box.

*Default =*

## **line\_width**

The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines).

*Default =*

## **name\_color**



The `name_color` property specifies the default color of names of graphical items.

*Default =*

## ShowName

The `ShowName` property specifies how the name of an object should be displayed. The possible values are as follows:

- `Full_path` - Show the object name with the full path.. For example, "Default::A.B."
- `Relative` - Show the object name with a relative path. For example, "A.B."
- `Name_only` - Show only the object name without any path information. For example, "B."

*Default = Name\_only*

## ShowStereotype

The `ShowStereotype` property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- `Label` - Show only the stereotype label (text).
- `Bitmap` - Show only the stereotype bitmap.
- `None` - Do not show stereotypes in diagrams.

*Default = Label*

# ComponentDiagramGE

The `ComponentDiagramGE` metaclass contains a property that controls the fill color of component diagrams.

## FillColor

The `FillColor` property specifies the default fill color for the object.

*Default = 218,218,218*

# CompRealization

The `CompRealization` metaclass contains a property that controls how realization (instantiation) is displayed in component diagrams.

## color

The color property specifies the default color of the border of a graphical item, such as an object box.

*Default =*

## line\_style

The line\_style property specifies the default line style for a graphical item.

The possible values are:

- straight\_arrows - straight line.
- rectilinear\_arrows - rectilinear line with right-angled corners placed at appropriate locations, depending on the starting and ending points of the line
- spline\_arrows - curved line without corners
- rounded\_rectilinear\_arrows - similar to ordinary rectilinear lines but use rounded bends rather than straight ninety-degree angles

*Default = spline\_arrows*

*Default = straight\_arrows*

## line\_width

The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines).

*Default =*

## name\_color

The name\_color property specifies the default color of names of graphical items.

*Default =*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = None*

## Constraint

The Constraint metaclass contains properties that control the appearance of constraints in component diagrams.

### color

*The color property specifies the default color of the border of a graphical item, such as an object box. (Default = 128,128,0)*

### Fillcolor

*The Fillcolor property specifies the default fill color for the object. (Default = 0,255,255)*

### line\_width

*The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines). (Default = 1)*

### name\_color

*The name\_color property specifies the default color of names of graphical items. (Default = 0,0,0)*

## Depends

The Depends metaclass contains a property that controls the appearance of dependency relation lines in component diagrams.

### color

The color property specifies the default color of the border of a graphical item, such as an object box.

*Default =*

### line\_style

The `line_style` property specifies the default line style for a graphical item.

The possible values are:

- `straight_arrows` - straight line.
- `rectilinear_arrows` - rectilinear line with right-angled corners placed at appropriate locations, depending on the starting and ending points of the line
- `spline_arrows` - curved line without corners
- `rounded_rectilinear_arrows` - similar to ordinary rectilinear lines but use rounded bends rather than straight ninety-degree angles

*Default = `spline_arrows`*

*Default = `straight_arrows`*

## **line\_width**

The `line_width` property specifies the default line width, in pixels, for drawing lines (for example, action state lines).

*Default = 1*

## **name\_color**

The `name_color` property specifies the default color of names of graphical items.

*Default =*

## **ShowStereotype**

The `ShowStereotype` property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- `Label` - Show only the stereotype label (text).
- `Bitmap` - Show only the stereotype bitmap.
- `None` - Do not show stereotypes in diagrams.

*Default = `None`*

# **DiagramFrame**

The `DiagramFrame` metaclass contains properties that control the appearance of a diagram frame.

## DefaultSize

The DefaultSize property specifies default values for diagram frame dimensions.

*Default = 20, 20, 590, 500*

## ShowName

The ShowName property determines the text to display next to a graphic element in a diagram. For most elements, you can provide a name and a label. Your descriptive label can be useful in cases where the name itself might not be sufficient due to various constraints, such as the inability to use spaces if the name of the element is to appear in the code.

The possible values for this property vary for the different elements, as does the default value used. The possible values are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is displayed with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

Note that for some elements, such as destruction elements in sequence diagrams and timing diagrams, the property is a boolean property that hides/shows the name of the element.

*Default = Name*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

# FileComponent

The FileComponent metaclass contains properties that control how file components are displayed in component diagrams.

## color

The color property specifies the default color of the border of a graphical item, such as an object box.

*Default =*

## line\_width

The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines).

*Default =*

## name\_color

The name\_color property specifies the default color of names of graphical items.

*Default =*

## ShowName

The ShowName property specifies how the name of an object should be displayed. The possible values are as follows:

- Full\_path - Show the object name with the full path.. For example, "Default::A.B."
- Relative - Show the object name with a relative path. For example, "A.B."
- Name\_only - Show only the object name without any path information. For example, "B."

*Default = Name\_only*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.

- None - Do not show stereotypes in diagrams.

*Default = Label*

## Flow

The Flow metaclass contains properties that control how information flows are displayed in component diagrams.

### color

The color property specifies the default color of the border of a graphical item, such as an object box.

*Default =*

### flowKeyword

The flowKeyword property is a Boolean value that specifies whether the flow keyword for the information flow is displayed in the diagram.

*Default = Checked*

### infoItemsColor

The infoItemsColor property specifies the color used to draw information items in diagrams.

*Default =*

### line\_style

The line\_style property specifies the default line style for a graphical item.

The possible values are:

- straight\_arrows - straight line.
- rectilinear\_arrows - rectilinear line with right-angled corners placed at appropriate locations, depending on the starting and ending points of the line
- spline\_arrows - curved line without corners
- rounded\_rectilinear\_arrows - similar to ordinary rectilinear lines but use rounded bends rather than straight ninety-degree angles

*Default = spline\_arrows*

*Default = rectilinear\_arrows*

## **line\_width**

The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines).

*Default =*

## **FolderComponent**

The FolderComponent metaclass contains properties that control how folder components are displayed in component diagrams.

### **color**

The color property specifies the default color of the border of a graphical item, such as an object box.

*Default =*

### **line\_width**

The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines).

*Default =*

### **name\_color**

The name\_color property specifies the default color of names of graphical items.

*Default =*

## **ShowName**

The ShowName property specifies how the name of an object should be displayed. The possible values are as follows:

- Full\_path - Show the object name with the full path.. For example, "Default::A.B."
- Relative - Show the object name with a relative path. For example, "A.B."
- Name\_only - Show only the object name without any path information. For example, "B."

*Default = Name\_only*



## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## General

Contains properties relating to the general appearance of component diagrams.

### SysMLDiagramKindTitle

When a diagram is created in SysML projects, the diagram frame contains a header for the diagram that includes information such as the name of the package that contains the diagram. The property SysMLDiagramKindTitle is used to specify a prefix for the diagram frame header that reflects the type of diagram. For example, when you create an internal block diagram, the header in the diagram frame starts with the prefix "ibd".

Note that the structure of the entire header is controlled by the property General::Graphics::DiagramHeader.

## Interface

The Interface metaclass contains properties that control how interfaces are displayed in component diagrams.

### color

The color property specifies the default color of the border of a graphical item, such as an object box.

*Default =*

### FillColor

The Fillcolor property specifies the default fill color for the object.

*Default = 0,255,255*

## **line\_width**

The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines).

*Default =*

## **name\_color**

The name\_color property specifies the default color of names of graphical items.

*Default =*

## **ShowAttributes**

The ShowAttributes property specifies which attributes are shown in an object box in a component diagram. The possible values are:

- All - Show all attributes.
- None - Do not show any attributes.
- Public - Show only the public attributes.
- Explicit - Show only those attributes that you have explicitly selected.

*Default = None*

## **ShowInheritedAttributes**

The ShowInheritedAttributes property specifies how attributes inherited from a base class are shown in a component diagram.

*Default = Cleared*

## **ShowInheritedOperations**

The ShowInheritedOperations property specifies how operations inherited from a base class are shown in a component diagram.

*Default = Cleared*

## **ShowBitmapMarkers**

The ShowBitmapMarkers property specifies whether to show bitmap markers for stereotypes.

*Default = True*

## ShowName

The ShowName property specifies how the name of an object should be displayed. The possible values are:

- Full\_path - Show the object name with the full path.. For example, "Default::A.B."
- Relative - Show the object name with a relative path. For example, "A.B."
- Name\_only - Show only the object name without any path information. For example, "B."

*Default = Name\_only*

## ShowOperations

The ShowOperations property specifies which operations to show in an object box in a component or object model diagram. The possible values are:

- All - Show all operations.
- None - Do not show any operations.
- Public - Show only the public operations.
- Explicit - Show only those operations that you have explicitly selected.

*Default = None*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

# InterfaceComponent

The InterfaceComponent metaclass contains properties that control how interface components are displayed in component diagrams.

## **color**

*The color property specifies the default color of the border of a graphical item, such as an object box. (Default = 255,255,0)*

## **line\_width**

*The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines). (Default = 1)*

## **name\_color**

*The name\_color property specifies the default color of names of graphical items. (Default = 255,255,0)*

# Note

The Note metaclass contains properties that control the appearance of notes in component diagrams.

## **color**

*The color property specifies the default color of the border of a graphical item, such as an object box. (Default = 0,128,64)*

## **Fillcolor**

*The Fillcolor property specifies the default fill color for the object. (Default = 0,255,255)*

## **line\_width**

*The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines). (Default = 1)*

## **name\_color**

*The name\_color property specifies the default color of names of graphical items. (Default = 0,128,255)*

# Requirement

The Requirement metaclass contains properties that control the appearance of requirements component diagrams.

## **color**

*The color property specifies the default color of the border of a graphical item, such as an object box. (Default = 128,128,0)*

## **Fillcolor**

*The Fillcolor property specifies the default fill color for the object. (Default = 0,255,255)*

## **line\_width**

*The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines). (Default = 1)*

## **name\_color**

*The name\_color property specifies the default color of names of graphical items. (Default = 0,0,0)*

# ConfigurationManagement

---

The ConfigurationManagement subject contains metaclasses that contain properties that specify values and command strings needed by various configuration management tools to interface with Rational Rhapsody.

## ClearCase

The ClearCase metaclass contains properties that enable the Rational ClearCase implementation with Rational Rhapsody.

### AddMember

The AddMember property specifies the command used to add an item to the archive.

For example, when you use Rational ClearCase, this command would be as follows:

```
"$OMROOT/etc/Executer.exe" "cleartool checkout -reserved -nc $rhpdirectory ; cleartool mkelem -eltype text_file -nc $unit; cleartool checkin -nc $rhpdirectory"
```

In this case, the argument to the command to run the Executer (\$OMROOT/etc/Executer.exe) consists of a list of three executable commands: cleartool checkout -reserved -nc \$rhpdirectory cleartool mkelem -eltype text\_file -nc \$unit cleartool checkin -nc \$rhpdirectory

The first command, cleartool checkout -reserved -nc, is defined within Rational ClearCase to check out an item. In this case, the item is checked out from the \_rpy directory (as indicated by the variable \$rhpdirectory).

In Rational ClearCase, to check an item out of the archive means to create a view-private, modifiable copy of a version. The option -reserved checks the item out as locked (R/W for the owner). Rational ClearCase expects items to be checked out of the repository for the user before they are checked into the archive.

The option -nc (no additional comment) is defined in Rational ClearCase to create an event record with no user-supplied comment string.

The second command, cleartool mkelem -eltype text\_file -nc \$unit, creates an element of type text\_file and assigns this element to the variable \$unit, which represents the unit of collaboration.

The third command, cleartool checkin -nc \$rhpdirectory, checks the unit in the \_rpy directory into the archive.

Because a second argument is not provided, the Executer runs these commands from the current (\_rpy) directory.

The default is as follows:

```
"$OMROOT/etc/Executer.exe" "cleartool checkout -reserved -nc "$UnitDirPath" ; cleartool mkelem -eltype text_file -nc "$UnitPath" ; cleartool checkin -nc "$UnitDirPath""
```

## AddMember\_ControlledFile

For Rational ClearCase only and like AddMember, this property is used for adding items to archive. However, the string contained in this property does not include the argument -eltype text.

The removal of this argument allows Rational ClearCase to store binary files, as well, and this is necessary for storing Controlled Files.

When Controlled Files are stored in Rational ClearCase, the value of this property is used. When other files are stored, the value contained in AddMember is used.

The default is as follows:

```
"$OMROOT/etc/Executer.exe" "cleartool checkout -reserved -nc "$UnitDirPath" ; cleartool mkelem -nc "$UnitPath" ; cleartool checkin -nc "$UnitDirPath"
```

## AddMember\_WithoutCheckOutCheckInDirectory

This property allows an item to be added to the archive without having to check out the parent directory, and check it back in after the item has been added. The use of this property allows you to check out and check in the parent directory separately.

*Default = cleartool mkelem -eltype text\_file -nc "\$UnitPath"*

## AddToArchiveAfterCreateUnitActivation

When you create an element in a Rational Rhapsody project that is not a unit by default (an actor, or class, for example), you can opt to create a unit from that element. When you do so, and this property is enabled, the new unit is automatically added as a unit in Rational ClearCase.

The possible values are as follows:

- Disabled - No units are archived
- UserConfirmation - prompts the user for confirmation before archiving the unit.
- Automatic - automatically archives the unit without asking the user.

*Default = Disable*

## Archive

The Archive property specifies the archive file for a project. This property is implicitly set when you use the Connect to Archive option within Rational Rhapsody, which uses the command string specified by the ConnectToCMRepository property.

Do not set the Archive property manually.

## ArchiveRoot

The ArchiveRoot property is reserved for future use.

## ArchiveSelection

The ArchiveSelection property specifies whether the archive façade is a file or directory. Most configuration management tools expect a file, whereas others expect a directory.

The Browse button in the Connect to Archive window runs a different browse utility for file-based and directory-based archives. When the ArchiveSelection property is set to File, the Open window for files is displayed by default. In this case, you cannot select a directory as the archive. If the archive is a directory rather than a file, set the ArchiveSelection property to Directory. The Browse for Folder window is displayed instead, which allows you to select a directory.

The possible values are as follows:

- Directory - The configuration management archive is a directory.
- File - The configuration management archive is a file. This is the default for PVCS.
- None - Neither a file nor directory is expected.

With Rational ClearCase, the ArchiveSelection property is not set (blank). This should not be changed. Rational ClearCase ignores the path to the archive, so the ArchiveSelection property is irrelevant. Similarly, the Browse button in the Connect to Archive window is disabled for Rational ClearCase.

## AskOnCheckoutReadWrite

The AskOnCheckoutReadWrite property is a Boolean value that specifies whether Rational Rhapsody should prompt users if they want to perform a check out (because they already have a read/write copy of the file).

*Default = Yes*

## AskOnLoadFromArchive

The AskOnLoadFromArchive property specifies whether to display a message when you check a file out of an archive that is not part of the current model.

If this property is set to Yes, Rational Rhapsody displays a window when you perform a checkout from the archive (or fetch operation for SCC) from either the ListArchive or Synchronize window.

When you check out a file from the archive and it is not the part of the current model, there are two possibilities:

- The corresponding parent unit is not loaded in the model.
- The parent unit, if loaded, is not the latest one.



Therefore, before Rational Rhapsody checks out an element from an archive, it checks whether the file is already part of the model. If not, it checks whether there is a stub unit in the model that refers to the same file. If neither criteria is met, Rational Rhapsody displays an informational message.

*Default = Yes*

## AuxProjPath

The AuxProjPath property is a string that identifies the SCC project path. Do not change this property. Removing the property value disconnects the link between the Rational Rhapsody project and the SCC project.

If you remove this value, you must also remove the ProjName property.

*Default = empty string*

## BaseAwareAutoMergeInvocation

The BaseAwareAutoMergeInvocation property specifies how to run the external textual DiffMerge tool which supports a base-aware detection of triviality of textual difference and a base-aware automatic merging between a base unit file and two other unit files. An automatic model merge operation can be performed only if all the differences in the current diff session are trivial differences.

If two units are being compared with a base unit, this three-way comparison makes it possible for the DiffMerge tool to determine automatically the need for some merges by using the concept of trivial versus non-trivial differences. For a difference in which only one unit differs from the base unit, it is identified as a non-conflicting difference or trivial difference. Similarly, if both of the units are different from base contributor but the differences are same, then it is also a trivial difference.

This applies to differences between model elements or between the attributes of model elements. However, if one unit contains a difference that does not appear in either the base unit or the other unit being compared, this is a non-trivial difference that must be resolved by the developer.

During the automatic merge, all of the differences are automatically accepted.

*Default = cleardiff -out \$output -base \$sourceBase -abo -qui \$source1 \$source2*

## BaseAwareDiffInvocation

*The BaseAwareAutoMergeInvocation property specifies how to run an external textual DiffMerge tool supporting a base-aware comparison and merging in Base Aware Diff mode between a base unit file and two other unit files. Possible tools supporting a base-aware comparison include TkDiff and the Rational ClearCase textual DiffMerge tools ClearDiff and ClearDiffMrg. \$source1: First unit selected in the window. \$source2: Second unit selected in the window. \$sourceBase: The text file containing compared values from the base. Default = cleardiffmrg -base \$sourceBase \$source1 \$source2*

## BaseAwareDiffMergeInvocation

*The BaseAwareDiffMergeInvocation property specifies how to run the external textual DiffMerge tool supporting a base-aware comparison and merging in Base Aware Merge mode between a base unit file and two other unit files. Default = \$BaseAwareDiffInvocation -out \$output*

## **BaseAwareTextDiffMergeEnabled**

*Determines whether a base-aware (three-unit) textual DiffMerge tool is available to be started. Default = Checked*

## **CallCheckOutOnSynchronize**

The CallCheckOutOnSynchronize property is a Boolean value that determines whether a checkout operation when you specify synchronization in Rational Rhapsody. Some configuration management tools require a full checkout of the selected elements, followed by an Add to Model operation. However, Rational ClearCase does not require a checkout. Put a Check in this property check box to ensure a checkout operation.

*Default = Cleared*

## **CheckIn**

The CheckIn property specifies the command used to check an item into the archive through the use of the main Configuration Items window.

This command is specific to the configuration management tool in use.

This command references the LogPart property, which in turn references the internal variable \$log, and the variable \$unit.

*Default = cleartool checkin \$LogPart \$UnitPath*

By default, Rational ClearCase does not allow you to check in an unmodified version of an item. Rational Rhapsody includes a script, SensativeCheckin.bat, that checks whether a checked-out version of an item is different from the previous version, and then performs the checkin. If the item has not been changed, the script automatically performs an “uncheckout” operation for that item.

To use this batch file, set the ConfigurationManagement::ClearCase::CheckIn property to \$OMROOT/etc/SensitiveCheckin.bat \$unit \$log.

## **CheckInRevisionPart**

The CheckInRevision property defines a new revision number used to check in a unit by using the CheckIn property. The value for \$CheckInRevisionPart is taken from the Label field in the window for checking in a unit that displays after selecting FileConfiguration Items, selecting a unit, and then selecting Check In the selected items.

## CheckOut

The CheckOut property specifies the command used to check an item out of the archive through the use of the main Configuration Items window.

If the item is locked, the variable \$mode is replaced by the contents of the ReadWrite property; otherwise, it is replaced by the contents of the ReadOnly property. LabelPart is also evaluated and replaced by the result.

The default value is as follows:

```
cleartool checkout -nc $mode $LabelPart
```

This command references the LabelPart property and the \$mode.

## CheckOutCheckInDirectoryOnceDuringAddToArchive

The CheckOutCheckInDirectoryOnceDuringAddToArchive property specifies that a directory is checked out only once when more than one file is added to the directory. The directory is checked in when all the files have been added.

*Default = Cleared*

## CheckOutFromArchive

The CheckOutFromArchive property specifies the command used to check an item out of the archive directly through the use of the List Archive window. For all the currently supported tools (Rational ClearCase and Serena PVCS Dimensions), this command is the same as that stored in the CheckOut property. Therefore, its value is \$CheckOut.

*Default = \$CheckOut*

## CheckOutRevisionPart

The CheckOutRevisionPart property defines a revision number used to check out a unit by using the CheckOut property. The value for \$CheckOutRevisionPart is taken from the Revision/Label field in the window for checking out a unit that displays after selecting FileConfiguration Items, selecting a unit, and then selecting Check Out the selected items.

The default is ( \$label ? -r \$label : -h ).

## CMHeaderItsLockedBy

The CMHeaderItsLockedBy property specifies the regular expression used to extract the name of the person who last locked the unit.

For all tools, an empty string in CMHeader\* means that the unit does not contain this type of information, therefore it is missing from the Configuration Items window when you select the List Archive option.

All properties of the format CMHeader\* are regular expressions that extract header information embedded into a configuration item when it is first checked into the archive. Embedded information can include such things as the ID of the item, its revision number, and the name of the person who last locked it (the “locker”).

Header information is contained in predefined, tool-specific keywords, which the various configuration management tools know how to expand.

The list of predefined keywords embedded in an item when it is first checked in is contained in a file stored in the HeaderFile property.

All regular expressions in CMHeader\* properties match only one set of parentheses.

## **CMHeaderItsVersion**

The CMHeaderItsVersion property specifies the regular expression that extracts the version information.

Consider the following PVCS expression: \ \$Revision: +([0-9\.] +)

This expression searches the header for a string that begins with \ \$Revision: and contains a version number that can consist of one or more digits 0 through 9, the backslash character (\), or a period.

All properties of the format CMHeader\* are regular expressions that extract header information embedded into a configuration item when it is first checked into the archive. Embedded information can include such things as the ID of the item, its revision number, and the name of the person who last locked it (the “locker”).

Header information is contained in predefined, tool-specific keywords, which the various configuration management tools know how to expand.

The list of predefined keywords embedded in an item when it is first checked in is contained in a file stored in the HeaderFile property.

All regular expressions in CMHeader\* properties match only one set of parentheses.

## **CommentsRequiredForCheckIn**

The CommentsRequiredForCheckIn property sets whether comments are required upon SCC Check In.

When this flag is set to Checked, comments are required for successful SCC Check In operation.

*Default = Cleared*

## **ConnectToCMRepository**

The ConnectToCMRepository property specifies the command used to connect Rational Rhapsody to a configuration management archive. For some tools, this is simply an echo.

For Rational ClearCase, the connect command is as follows: "\$OMROOT/etc/Executer.exe" "move \$rhpdirectory \$rhpdirectory.orig ; cleartool mkelem -eltype directory -nc -nco \$rhpdirectory ;" ".."

This is a command to run the Rational Rhapsody Executer with two arguments:

- The first argument consists of two executable commands: move \$rhpdirectory \$rhpdirectory.orig  
cleartool mkelem -eltype directory -nc -nco \$rhpdirectory
- The first command backs up the repository (the \_rpy directory for the user). The second command is defined within Rational ClearCase to create an element of type directory. The -nc option (no additional comment) creates an event record with no user-supplied comment string. The new directory points to the repository.
- The second argument, "..", tells the Executer to run the commands from the directory just above the current one.

A side-effect of the ConnectToCMRepository property is that it sets the Archive property to the location of the configuration management archive, even if a configuration management command is not actually executed.

## Delete

The Delete property specifies the script that deletes a particular item from the current Rational ClearCase directory element.

When you delete a Rational Rhapsody unit and the project settings indicate that the configuration management tool is Rational ClearCase, Rational Rhapsody runs this delete script to remove the deleted item from the Rational ClearCase view as well.

If the property does not exist, or if it is empty, the unit is removed from the Rational Rhapsody model, but not from the Rational ClearCase view.

By default, this feature is disabled. To enable it, set the DeleteActivation property.

The default value of the Delete property is as follows: "\$OMROOT/etc/Executer.exe" "cleartool checkout -reserved -nc "\$UnitDirPath" ; cleartool rmname -nc "\$UnitPath" ; cleartool checkin -nc "\$UnitDirPath""

It uses the following keywords:

- \$units - Specifies the full path names of the unit file names, separated by spaces
- \$dirs - Specifies the names of the unit directories, separated by spaces

## DeleteActivation

The property DeleteActivation controls how the deletion of Rhapsody units is handled in terms of configuration management. The possible values for the property are:

- Disable - when a unit is deleted from a model, no corresponding action is taken in the configuration management system

- **UserConfirmation** - when a unit is deleted from a model, the user is asked whether the file should also be deleted in the configuration management system
- **Automatic** - when a unit is deleted from a model, the deletion is also carried out in the configuration management system

*Default = Disable*

## DeleteDirectory

The DeleteDirectory property specifies the command to delete a directory in the Rational ClearCase configuration management system.

The default value is as follows:

```
"$OMROOT/etc/Executer.exe" "cleartool checkout -reserved -nc "$UnitDirPath\.." ; cleartool rmname -nc "$UnitDirPath" ; cleartool checkin -nc "$UnitDirPath\.."
```

## DiffInvocation

The DiffInvocation property specifies the command to run the external, textual DiffMerge tool. For Rational ClearCase, this property runs the Rational Clearcase diff tool.

The product searches for the property value as follows:

- First, it searches through the configuration management property values.
- If the property is not found or if it is set to empty, Rational Rhapsody searches under General::DiffMerge.

By default, site-specific properties override the factory properties.

*Default = cleardiffmrg \$source1 \$source2*

## DiffMergeInvocation

The DiffMergeInvocation property specifies the command to run the external, textual Diff/Merge tool.

For Serena PVCS and Rational Clearcase, this property runs the corresponding diff tool for the configuration management tool. For the other configuration management tools, this property is set to an empty string.

The product searches for the property value as follows:

- First, it searches through the configuration management property values.
- If the property is not found or if it is set to empty, Rational Rhapsody searches under General::DiffMerge.

By default, site-specific properties override the factory properties.

*Default = \$DiffInvocation -out \$output*

## EnableSCCCancel

The EnableSCCCancel property is used to provide a cancel option during configuration management operations.

When set to True, the SCC provider displays its cancel window during configuration management operations.

The DeleteActivation property specifies whether deleting units from the Rational Rhapsody model triggers the delete command in the archive.

The possible values are as follows:

- Cleared - The delete operation is disabled.
- UserConfirmation - The delete operation is performed only on user confirmation.
- Automatic - If you delete a unit from Rational Rhapsody, the unit is automatically removed from the archive without any notification.

*Default = Cleared*

## Fetch

The Fetch property specifies the command used by the Rational Rhapsody Diff/Merge to fetch files from the configuration management tool.

*Default = \$OMROOT/etc/copy.bat \$FetchLabelPart \$targetDir\%unit*

## FetchFromArchive

The FetchFromArchive property specifies the command used by the Rational Rhapsody Diff/Merge to fetch files from the configuration management archive.

*Default = \$Fetch*

## FetchLabelPart

The FetchLabelPart property is a string used by the Fetch property to identify units by label. For more information, see the Fetch property listed previously in this metaclass.

The default is ( \$label ? \$UnitPath@%\$label : \$UnitPath ).

## FooterFile

The FooterFile property specifies the file footer.

## HeaderFile

The HeaderFile property specifies the file header. This property is reserved for future use.

## History

The History property specifies the batch script that views the version tree of a given item.

The default value is as follows: `cleartool lsvtree -graph $UnitPath`

## InValidCharactersInRevisionDescription

The InValidCharactersInRevisionDescription property provides a list of iInvalid characters in the Revision Description (or comments) during a Check In operation. The list of invalid characters - provided by the user - validates the revision description and notifies the user if there are any invalid characters. By default this property value is set to ">" and users can append more characters to it. This property is used only in Rational ClearCase Batch Mode CheckIn operation.

## LabelPart

The LabelPart property specifies how to embed a revision label.

The syntax for embedding labels in Rational ClearCase is: `($label ? -version $UnitPath@ @$label : $UnitPath)`

This expression uses the `(Exp1 ? Exp2 : Exp3)` construct. If you entered a label in the Revision/Label field in the Check In or Check Out window, \$label is True and LabelPart evaluates to `-version $unit@ @$label`, where \$unit and \$label are replaced by their respective values.

Otherwise, \$label is False and LabelPart evaluates to \$unit.

## ListArchive

The ListArchive property specifies the command to list the contents of the archive.

For example, the command to list the archive in Rational ClearCase is:

```
cleartool ls -vob_only -long -recurse
```

Expansion of ListArchive is done by “simple” substitution. The expanded command is executed as a shell command and the output is assigned to a temporary string inside Rational Rhapsody. This string is then matched against the relevant regular expression found in ListArchive\* to extract specific information from the output.

The following example shows sample output from the ListArchive command:



version animPingPong.cfg@@\main\2 Rule: element \* \main\LATEST version  
animPingPongFlat.cfg@@\main\2 Rule: element \* \main\LATEST version  
animPingPongMultiThread.cfg@@\main\2 Rule: element \* \main\LATEST version  
Default.sbs@@\main\2 Rule: element \* \main\LATEST version DefaultConfig.cfg@@\main\2 Rule:  
element \* \main\LATEST version Model1.ond@@\main\4 Rule: element \* \main\LATEST version  
MSC1.msc@@\main\3 Rule: element \* \main\LATEST

An empty string in any of the ListArchive\* properties means that the configuration item does not contain this information; therefore, it is missing from the archive listing.

## ListArchiveItsLockedBy

The ListArchiveItsLockedBy property specifies the regular expression that extracts from the output of the List Archive command the name of the person who last locked a configuration item.

Consider the following PVCS expression: Locked by: +([0-9a-zA-Z]+)

This expression tells the interpreter to look in the header information for “Locked by:” string beginning. Once the string is matched, only the regular expression between the brackets is taken as the “locker.”

For example, the Rational ClearCase CheckIn command is: cleartool checkin \$LogPart \$unit

This command references the LogPart property, which in turn references the internal variable \$log, and the variable \$unit.

## ListArchiveItsVersion

The ListArchiveItsVersion property specifies the version of the configuration management archive.

The default value is as follows:

@@ ?[?([0-9a-zA-Z\_: \\*-\.\|+)]?

## ListArchiveItsWorkingFile

The ListArchiveItsWorkingFile specifies the regular expression that extracts from the output of the List Archive command the working file of an item.

The default is as follows: ([\0-9\.\a-zA-Z\_-]+)@@

As an example, this command "[0-9a-zA-Z\_-]+)@@" tells the interpreter that the working file is indicated by the part of the output string preceding the two @ symbols. Once the string is matched, only the regular expression between the brackets is taken as the working file.

## ListArchiveRevisionPart

The ListArchiveRevisionPart property is a string that represents the revision part from the output of the

List Archive command.

## LockItem

The LockItem property is a string that specifies the command tool used to lock an item in the archive.

## LogPart

The LogPart property specifies how to embed a log, if provided, in a configuration management command. The log is the comment entered in the Revision/Description field in the Check In window.

For example, a log is embedded in Rational ClearCase as follows: ( \$log ? -c \$log : -nc )

This expression uses the (Exp1 : Exp2 : Exp3) construct. If you entered a comment in the Check In window, \$log is True and LogPart evaluates to -c, followed by the comment string. Otherwise, \$log is False and LogPart evaluates to -nc.

## MakeCMShadowDir

The MakeCMShadowDir property specifies the Rational ClearCase command to create a directory as a VOB element.

The default value is as follows:

```
"$OMROOT/etc/Executer.exe" "cleartool checkout -reserved -nc "$parentdir" ; move "$fulldir"  
"$fulldir.orig" ; cleartool mkelem -eltype directory -nc -nco "$fulldir" ; cleartool checkin -nc "$parentdir"  
; copy "$fulldir.orig" "$fulldir" ; " .."
```

## MakeCMShadowDirActivation

The MakeCMShadowDirActivation property controls whether new directories created by a save in Rational Rhapsody is elements in Rational ClearCase.

The possible values are as follows:

- Disable - Disable this functionality.
- UserConfirmation - Prompt the user for confirmation before creating the elements.
- Automatic - Automatically create elements whenever new directories are created by a save in Rational Rhapsody.

If you set this property to Automatic, every new package that is saved creates a configuration management directory, including branches. If you do not want this to occur, set this property to UserConfirmation.

*Default = Disable*

## MergeOutput

The MergeOutput property specifies the file that is to hold the results of a merge operation.

*Default = \$temp\out.txt*

## ModePart

The ModePart property specifies the locking mode of a configuration item. This is defined as: \$mode

If the item is locked, \$mode is replaced by the value of the ReadWrite property; otherwise, it is replaced by the value of the ReadOnly property.

## Move

The Move property specifies the Rational ClearCase command for a unit move.

The default value is as follows: "\$OMROOT/etc/Executer.exe" "cleartool checkout -reserved -nc "\$olddir" "\$newdir"; cleartool mv -nc "\$oldName" "\$newName" ; cleartool checkin -nc "\$olddir" "\$newdir" "

## MoveActivation

The MoveActivation property is a Boolean value that specifies whether moving units in the Rational Rhapsody model (in a way that changes the unit file location on the hard drive) triggers a move command in the archive.

The possible values are as follows:

- Disable - Disable this functionality.
- UserConfirmation - Prompt the user for confirmation before moving the elements in the archive.
- Automatic - Automatically move elements whenever units are moved in the Rational Rhapsody model.

*Default = Disable*

## MoveDirectory

The MoveDirectory property specifies the command to move a directory in the Rational ClearCase configuration management system.

The default value is as follows: "\$OMROOT/etc/Executer.exe" "cleartool checkout -reserved -nc "\$olddir" "\$newdir"; cleartool mv -nc "\$oldName" "\$newName" ; cleartool checkin -nc "\$olddir" "\$newdir" "

## MultiRecordDelimiter

The MultiRecordDelimiter property specifies the regular expression representing the symbol used to separate multiple records in an archive listing.

For example, in Rational ClearCase, the multiple-record delimiter is defined as:

(version )(file element)

Multiple records are separated by the string version, located at the beginning of each line. See the ListArchive property for sample output.

## OperationErrorPattern

The OperationErrorPattern property notifies you that the specified error occurred during batch mode. Rhapsody searches all configuration management operation output for the string specified in this property. Rhapsody first checks for errors, then for warnings.

Note the following:

- The string pattern is not case-sensitive.
- It can be a regular expression.

For example, on Rational ClearCase, you could specify the following string: cleartool: Error:

*Default = empty string*

## OperationWarningPattern

The OperationWarningPattern property notifies you that the specified warning occurred during batch mode. Rhapsody searches all configuration management operation output for the string specified in this property. Rhapsody first checks for errors, then for warnings.

Note the following:

- The string pattern is not case-sensitive.
- It can be a regular expression.

For example, on Rational ClearCase, you could specify the following string: cleartool: Warning

*Default = empty string*

## PostConnectToCMRepository

The PostConnectToCMRepository property is used for internal purposes only. Do not change the value of this property.

## ProjName

The ProjName property is a string that identifies the SCC project name. Do not change this property. Removing the property value disconnects the link between the Rational Rhapsody project and the SCC project.

If you remove this value, you must also remove the AuxProjPath property.

*Default = empty string*

## ReadOnly

The ReadOnly property specifies how to embed a ReadOnly flag in the configuration management command.

*Default = -unreserved*

## ReadWrite

The ReadWrite property specifies how to embed a ReadWrite flag in the configuration management command.

For example, the Rational ClearCase CheckIn command is: cleartool checkin \$LogPart \$unit

*Default = -reserved*

## RedirectOutputToRhapsody

The UseSCCTool property is a Boolean value that specifies whether the resulting output of an SCC CM command should be redirected (displayed) in the Rational Rhapsody CM output window tab.

*Default = Checked*

## Rename

The Rename property specifies the script that renames a particular item in the current Rational ClearCase directory element.

When you rename a Rational Rhapsody unit (either explicitly by changing the file name field in the Edit Unit window, or implicitly by changing the unit name) and the project settings indicate that the configuration management tool is Rational ClearCase, Rational Rhapsody runs this rename script to rename the item in the Rational ClearCase view as well.

If the property does not exist, or if it is empty, the unit is renamed in the Rational Rhapsody model, but not in the Rational ClearCase view.

By default, this feature is disabled. To enable it, set the RenameActivation property.

The default value of the Rename property is as follows: "\$SOMROOT/etc/Executer.exe" "cleartool checkout -reserved -nc \$dir ; cleartool mv -nc \$oldName \$newName ; cleartool checkin -nc \$dir"

It uses the following keywords:

- \$oldName - Specifies the full path name of the existing unit file name
- \$newName - Specifies the full path name of the new unit file name
- \$dir - Specifies the name of the unit directory

## RenameActivation

The property RenameActivation controls how the renaming of Rhapsody units is handled in terms of configuration management. The possible values for the property are:

- Disable - when a unit is renamed, no corresponding change is made in the configuration management system
- UserConfirmation - when a unit is renamed, the user is asked whether the name change should also be made in the configuration management system
- Automatic - when a unit is renamed, the name change is also made in the configuration management system

*Default = Disable*

## RenameDirectory

The RenameDirectory property specifies the command to rename a directory in the Rational ClearCase configuration management system.

The default value is as follows: "\$OMROOT/etc/Executer.exe" "cleartool checkout -reserved -nc \$dir ; cleartool mv -nc \$oldName \$newName ; cleartool checkin -nc \$dir"

The RenameActivation property controls whether the rename operation (specified by the Rename property) is enabled.

## ReplaceNewLinesInRevisionDescriptionWithSpaces

When SensitiveCheckin.bat is used for check in, you can enable this property to properly format revision descriptions or comments that contain multiple lines of text.

*Default = Cleared*

## Repository

The Repository property is used for internal purposes only. Do not change the value of this property.

The default is .\\$SubDirs\\$FileName.

For example, the Rational ClearCase CheckIn command is:

```
cleartool checkin $LogPart $unit
```

This command references the LogPart property, which in turn references the internal variable \$log and the

variable \$unit.

## **SaveOnCheckOut**

The SaveOnCheckOut property is a Boolean value that specifies whether a Rational Rhapsody save should be triggered whenever a checkout occurs.

*Default = Checked*

## **ShowNewItemInSynchronize**

The ShowNewItemInSynchronize property is a Boolean value that is directly related to what you see in the Synchronize window.

If this property is set to No, new items that are added (by another member of the team) to the archive after the Rational Rhapsody project is open are not displayed.

*Default = Yes*

## **StoreInSeparateDirectoryActivation**

The StoreInSeparateDirectoryActivation property affects how an existing package is converted. The following conditions apply with this property:

- Enable this property by selecting UserConfirmation or Automatic from the drop-down list. When an existing flat package is converted to a package as a directory, the directory is created on the configuration management side and the package with its children moved to this directory.
- Disable this property by selecting Disable. When an existing flat package is converted to a package as a directory, the directory is removed on the configuration management side and the children of this package are removed as well.

*Default = Disable*

## **SupportTreeRepository**

The SupportTreeRepository property is provided for compatibility with previous versions of Rational Rhapsody.

For PVCS Dimensions, this property is for internal purposes only. Do not change the value of this property.

If you checked in Rational Rhapsody projects to an SCC archive prior to Version 4.0, you must perform the following steps:

- Create a directory in the configuration management tool with the same name of the directory that holds the .rpy file.
- Disconnect from the existing archive.

- Change the value of this property to an empty string.
- Reconnect to the archive.

The default value for PVCS is No; the default for SCC is an empty string.

## UnLockItem

The UnLockItem property is a string that specifies the command used to release a lock placed on an item in the archive.

For example, the Rational ClearCase CheckIn command is:

```
cleartool checkin $LogPart $unit
```

This command references the LogPart property, which in turn references the internal variable \$log, and the variable \$unit.

This command references the CheckInRevisionPart, ModePart, and LogPart properties; and the \$archive, \$archivedirectory, and \$unit internal variables

## General

The General properties inform Rational Rhapsody which configuration management tool you are using, and the length of the timeout for commands for that particular tool.

## CMConflictResolution

The CMConflictResolution property is related to CM synchronization and Rational Rhapsody model changes.

When CM synchronization is enabled and you try to make changes in a Rational Rhapsody model (delete, rename, or move), and Rational Rhapsody finds that those changes cannot be made in the CM system, it displays an informative window (“Change in Directory Structure”).

The possible values are as follows:

- AskUser - Always ask the user.
- ModelOnly - Changes is done in the model, but the file and directory layout remains the same.
- ModelAndFileSystem - Changes is done in both the model and the file/directory layout.

*Default = AskUser*

## CMOperationEndSeparator

The CMOperationStartSeparator property is a MultiLine value that specifies the separator for the end of a



configuration management operation. The text specified in this property is printed to the configuration management window after the configuration management operation has executed.

This property supports the following keywords:

- \$Time - The current time
- \$Date - The date
- \$User - The current user name
- \$Operation - The name of the operation.

Note that the configuration management output window is not cleared between consecutive configuration management operations. To clear the output window, right-click on the window. After you close a project, this window is cleared automatically.

*Default = empty MultiLine*

## **CMOperationStartSeparator**

The CMOperationStartSeparator property is a MultiLine value that specifies the separator for the start of a configuration management operation. The text specified in this property is printed to the configuration management window as the header before the configuration management operation is executed.

This property supports the following keywords:

- \$Time - The current time
- \$Date - The date
- \$User - The current user name
- \$Operation - The name of the operation.

The default value is as follows:

```
==== $Operation; ==== Time: $Time; Date: $Date; User: $User ; ====
```

For example: ===== Checkout; Time: 2:38 Eastern Standard Time PM ; Date: Wed, 21, Nov 2001 ; User: npadmawar ; =====

Note that the configuration management output window is not cleared between consecutive configuration management operations. To clear the output window, right-click on the window. After you close a project, this window is cleared automatically.

## **CMTool**

The CMTool property specifies which configuration management tool you are using. Valid properties for each configuration management tool are predefined in metaclasses of the same name. When evaluating property strings that reference other properties, Rational Rhapsody looks only within the same metaclass.

For example, the CheckOutFromArchive property for Rational ClearCase references another property called CheckOut.

The possible values are:

- None
- ClearCase
- Rational Team Concert

*Default = None*

Note: If you are using Rational Synergy as your configuration management tool, you need to set the UseSCCTool property to "Yes" and leave this property set to "None."

## **DefaultLockReserveOnCheckOut**

The DefaultLockReserveOnCheckOut property provides a default lock or reserved value during a Checkout operation in Batchmode.

The default is Cleared, meaning that whenever a checkout operation is performed it is locked or reserved.

## **EncloseCommentsInQuotes**

Comments provided during configuration management operations are enclosed in double quotation marks by default. These double quotation marks around comments cause problems in configuration management tools such as Rational Synergy and Rational ClearCase. To avoid double quotation marks in comments, the EncloseCommentsInQuotes property can be set to 'No' which prevents quotes from being enclosed around comments during configuration management operations. By default this property value is "Yes" which results in double quotation marks being enclosed around comments.

## **FilterUnresolvedUnitsInCMSynchDialog**

This property allows you to filter the display of unresolved units in a CM synchronization window.

*Default = Cleared. This means that unresolved units are displayed in a CM synchronization window.*

## **GUI**

The GUI property specifies the way units are displayed in the Configuration Items window.

The possible values are as follows:

- Flat - Units are displayed as a flat list.
- Tree - Units are displayed in a tree format:

It is possible to make multiple selections in the tree.

Selecting a higher-level unit in the tree does not automatically select subordinate units, unless you select the With Descendants option in the Check In/Out window.

If you select a unit that has subordinate units without selecting the With Descendants option, you might get stubs (unresolved units) for the subordinate units.

*Default = Flat*

## ReportLoadingError

The ReportLoadingError property redirects loading errors.

After a checkout or fetch, Rational Rhapsody attempts to load the checked out element into the project. However, this operation might fail (for example, if the file is checked out to a wrong location, or is corrupt).

The possible values are as follows:

- OutputWindow - Display the error in the output window.
- MessageBox - Display the error in a message box.
- Both - Display the error in the output window and a message box.
- None - Do not display loading errors.

*Default = OutputWindow*

## RunCMToolCommand

The RunCMToolCommand property specifies the command to execute the configuration management tool. This command is tied to a user-defined button.

*Default = empty File*

## ToolCommandTimeOut

The ToolCommandTimeOut property specifies the time, in milliseconds, that Rational Rhapsody should wait for the configuration management tool to return its output before timing out.

This value should be higher for cross-network archives with a loaded network and for large units.

*Default = 30000*

## UseHybridModeWhenPossible

The Rational ClearCase SCC interface does not provide the required functionality to successfully perform certain configuration management operations, such as "Diff with Rational Rhapsody " and "Store in Separate Directory," which are provided in the Batch mode for Rational ClearCase. In order to successfully perform these operations in Rational ClearCase SCC mode, executing in the hybrid mode should be allowed. In this case, Rational Rhapsody checks if the SCC provider is Rational ClearCase, and if so it executes the corresponding batch commands.

*Default = Checked*

## **UserDefCommand\_1**

The UserDefCommand\_1 property specifies the first parameter used in the command that is tied to a user-defined button for the configuration management tool.

*Default = empty string*

## **UserDefCommand\_1\_Title**

In the Configuration Items window (File > Configuration Items) there are four user defined command buttons on the lower right side. The user can specify a title for each of these four command buttons (in this case UserDefCommand\_1\_Title). When rolled over with the pointer, the user-defined title is displayed.

Right-clicking on a unit in the browser and selecting Configuration Management from the submenu that opens and displays configuration management operations and user-defined titles. When a user-defined title is selected, the command value is displayed on the status bar.

## **UserDefCommand\_2**

The UserDefCommand\_2 property specifies the second parameter used in the command that is tied to a user-defined button for the configuration management tool.

*Default = empty string*

## **UserDefCommand\_2\_Title**

In the Configuration Items window (File > Configuration Items) there are four user defined command buttons on the lower right side. The user can specify a title for each of these four command buttons (in this case UserDefCommand\_2\_Title). When rolled over with the pointer, the user-defined title is displayed.

Right-clicking on a unit in the browser and selecting Configuration Management from the submenu that opens and displays configuration management operations and user-defined titles. When a user-defined title is selected, the command value is displayed on the status bar.

## **UserDefCommand\_3**

*The UserDefCommand\_3 property specifies the third parameter used in the command that is tied to a user-defined button for the configuration management tool. Default = empty string*

## **UserDefCommand\_3\_Title**

In the Configuration Items window (File > Configuration Items) there are four user defined command buttons on the lower right side. The user can specify a title for each of these four command buttons (in this

case UserDefCommand\_3\_Title). When rolled over with the pointer, the user-defined title is displayed.

Right-clicking on a unit in the browser and selecting Configuration Management from the submenu that opens and displays configuration management operations and user-defined titles. When a user-defined title is selected, the command value is displayed on the status bar.

## UserDefCommand\_4

*The UserDefCommand\_4 property specifies the fourth parameter used in the command that is tied to a user-defined button for the configuration management tool. Default = empty string*

## UserDefCommand\_4\_Title

In the Configuration Items window (File > Configuration Items) there are four user defined command buttons on the lower right side. The user can specify a title for each of these four command buttons (in this case UserDefCommand\_4\_Title). When rolled over with the pointer, the user-defined title is displayed.

Right-clicking on a unit in the browser and selecting Configuration Management from the submenu that opens and displays configuration management operations and user-defined titles. When a user-defined title is selected, the command value is displayed on the status bar.

## UseSCCtool

The UseSCCtool property specifies whether the standard SCC interface between Rational Rhapsody and your configuration management tool is used. Note that when you use the SCC interface, all of the batch mode command properties are unused.

Use the UseSCCtool property (and the ConfigurationManagement::General::CMTool property, if necessary) to set the tool you are using. For example:

- If you are using Rational Synergy as your configuration management tool, set UseSCCtool to "Yes" and CMTool to "None"
- If you are using Rational ClearCase in Batch mode, set UseSCCtool to "No" and CMTool to "ClearCase"; in SCC mode, set UseSCCtool to "Yes" and CMTool to "None"

*Default = No*

## UseUnitTimeStamps

The UseUnitTimeStamps property lets you set whether or not to store unit time stamps within the repository files. This time stamp is used by the Synchronize functionality in Rational Rhapsody to detect changes done to the repository file from outside of Rational Rhapsody.

Use this property along with the General::Model::AutoSynchronize property.

*Default = Cleared*

# PVCS

The PVCS metaclass contains properties that enable the Serena PVCS Dimensions implementation with Rational Rhapsody..

## AddMember

The AddMember property specifies the command used to add an item to the archive.

The PVCS default value is as follows: ( \$SupportTreeRepository ? vcs -C"\$archive" -n -T' ' -i \$unit : vcs -C"\$archive" -n -T' ' -i "\$UnitArchiveDir"("\$UnitPath") )

## Archive

The Archive property specifies the archive file for a project. This property is implicitly set when you use the Connect to Archive option within Rational Rhapsody, which uses the command string specified by the ConnectToCMRepository property.

Do not set the Archive property manually.

## ArchiveRoot

The ArchiveRoot property is reserved for future use.

The default is ( \$SupportTreeRepository ? : \$ArchivePath ).

## ArchiveSelection

The ArchiveSelection property specifies whether the archive façade is a file or directory. Most configuration management tools expect a file, whereas others expect a directory.

The Browse button in the Connect to Archive window runs a different browse utility for file-based and directory-based archives. When the ArchiveSelection property is set to File, the Open window for files is displayed by default. In this case, you cannot select a directory as the archive.

If the archive is a directory rather than a file, set the ArchiveSelection property to Directory. The Browse for Folder window is displayed instead, which allows you to select a directory.

The possible values are as follows:

- Directory - The configuration management archive is a directory.
- File - The configuration management archive is a file. This is the default for PVCS.
- None - Neither a file nor directory is expected.

With Rational ClearCase, the ArchiveSelection property is not set (blank). This should not be changed. Rational ClearCase ignores the path to the archive, so the ArchiveSelection property is irrelevant. Similarly, the Browse button in the Connect to Archive window is disabled for Rational ClearCase.

## AskOnCheckoutReadWrite

The AskOnCheckoutReadWrite property is a Boolean value that specifies whether Rational Rhapsody should prompt users if they want to perform a check out (because they already have a read/write copy of the file).

The default value for PVCS is Yes.

## AskOnLoadFromArchive

The AskOnLoadFromArchive property specifies whether to display a message when you check a file out of an archive that is not part of the current model.

If this property is set to Yes, Rational Rhapsody displays a window when you perform a checkout from the archive (or fetch operation for SCC) from either the ListArchive or Synchronize window.

When you check out a file from the archive and it is not the part of the current model, there are two possibilities:

- The corresponding parent unit is not loaded in the model.
- The parent unit, if loaded, is not the latest one.

Therefore, before Rational Rhapsody checks out an element from an archive, it checks whether the file is already part of the model. If not, it checks whether there is a stub unit in the model that refers to the same file. If neither criteria is met, Rational Rhapsody displays an informational message.

*Default = Yes*

## AuxProjPath

The AuxProjPath property is a string that identifies the SCC project path. Do not change this property. Removing the property value disconnects the link between the Rational Rhapsody project and the SCC project.

If you remove this value, you must also remove the ProjName property.

*Default = empty string*

## BaseAwareAutoMergeInvocation

The BaseAwareAutoMergeInvocation property specifies how to run the external textual DiffMerge tool which supports a base-aware detection of triviality of textual difference and a base-aware automatic merging between a base unit file and two other unit files.

An automatic model merge operation can be performed only if all the differences in the current diff session are trivial differences. If two units are being compared with a base unit, this three-way comparison makes it possible for the DiffMerge tool to determine automatically the need for some merges by using the concept of trivial versus non-trivial differences. For a difference in which only one unit differs from the base unit, it is identified as a non-conflicting difference or trivial difference. Similarly, if both of the units are different from base contributor but the differences are same, then it is also a trivial difference. This applies to differences between model elements or between the attributes of model elements. However, if one unit contains a difference that does not appear in either the base unit or the other unit being compared, this is a non-trivial difference that must be resolved by the developer. During the automatic merge, all the differences is automatically accepted. There is no default value.

## BaseAwareDiffInvocation

The BaseAwareDiffInvocation property specifies how to run an external textual DiffMerge tool supporting base-aware comparison and merging in Base Aware Diff mode, between a base unit file and two other unit files.

*Default = Blank*

## BaseAwareDiffMergeInvocation

The BaseAwareDiffMergeInvocation property specifies how to run the external textual DiffMerge tool supporting a base-aware comparison and merging in Base Aware Merge mode between a base unit file and two other unit files.

## CallCheckOutOnSynchronize

The CallCheckOutOnSynchronize property is a Boolean value that determines whether a checkout operation when you specify synchronization in Rational Rhapsody. Some configuration management tools require a full checkout of the selected elements, followed by an Add to Model operation. However, Rational ClearCase does not require a checkout. Put a Check in this property check box to ensure a checkout operation.

*Default = Cleared*

## CheckIn

The CheckIn property specifies the command used to check an item into the archive through the use of the main Configuration Items window.

This command is specific to the configuration management tool in use.

The PVCS default is as follows:

```
( $SupportTreeRepository ? put -C"$archive" -y $CheckInRevisionPart $mode $LogPart $unit : put -C"$archive" -y $CheckInRevisionPart $mode $LogPart "$UnitArchiveDir("$UnitPath") ).
```



## CheckInRevisionPart

The CheckInRevision property defines a new revision number used to check in a unit by using the CheckIn property. The value for \$CheckInRevisionPart is taken from the Label field in the window for checking in a unit that displays after selecting FileConfiguration Items, selecting a unit, and then selecting Check In the selected items.

The PVCS default value is ( \$label ? -v\$label ).

## CheckOut

The CheckOut property specifies the command used to check an item out of the archive through the use of the main Configuration Items window.

The PVCS default value is as follows:

```
( $SupportTreeRepository ? get -C"$archive" -y -r$label $mode $unit : get -C"$archive" -y -r$label $mode "$UnitArchiveDir"("$UnitPath") ).
```

## CheckOutFromArchive

The CheckOutFromArchive property specifies the command used to check an item out of the archive directly through the use of the List Archive window. For all the currently supported tools (Rational ClearCase and Serena PVCS Dimensions), this command is the same as that stored in the CheckOut property. Therefore, its value is \$CheckOut.

*Default = \$CheckOut*

## CheckOutRevisionPart

The CheckOutRevisionPart property defines a revision number used to check out a unit by using the CheckOut property. The value for \$CheckOutRevisionPart is taken from the Revision/Label field in the window for checking out a unit that displays after selecting FileConfiguration Items, selecting a unit, and then selecting Check Out the selected items.

*Default = ( \$label ? -r \$label : -h )*

## CMHeaderItsLockedBy

The CMHeaderItsLockedBy property specifies the regular expression used to extract the name of the person who last locked the unit.

For all tools, an empty string in CMHeader\* means that the unit does not contain this type of information, therefore it is missing from the Configuration Items window when you select the List Archive option.

All properties of the format CMHeader\* are regular expressions that extract header information embedded

into a configuration item when it is first checked into the archive. Embedded information can include such things as the ID of the item, its revision number, and the name of the person who last locked it (the “locker”).

Header information is contained in predefined, tool-specific keywords, which the various configuration management tools know how to expand. For example, PVCS knows how to expand the keyword \$Header: /cvsroot/telelogic/Rhapsody/help.core/com.ibm.rhapsody.property.definition.doc/ConfigurationManagement.xml,v 1.23 2010/09/22 18:59:43 pder Exp \$ into a string containing the name of the working file, the version, and the locker.

The list of predefined keywords embedded in an item when it is first checked in is contained in a file stored in the HeaderFile property.

All regular expressions in CMHeader\* properties match only one set of parentheses.

The PVCS default is empty string (blank).

## **CMHeaderItsVersion**

The CMHeaderItsVersion property specifies the regular expression that extracts the version information.

Consider the following PVCS expression: \ \$Revision: +([0-9\.] +)

This expression searches the header for a string that begins with \ \$Revision: and contains a version number that can consist of one or more digits 0 through 9, the backslash character (\), or a period.

All properties of the format CMHeader\* are regular expressions that extract header information embedded into a configuration item when it is first checked into the archive. Embedded information can include such things as the ID of the item, its revision number, and the name of the person who last locked it (the “locker”).

Header information is contained in predefined, tool-specific keywords, which the various configuration management tools know how to expand. For example, PVCS knows how to expand the keyword \$Header: /cvsroot/telelogic/Rhapsody/help.core/com.ibm.rhapsody.property.definition.doc/ConfigurationManagement.xml,v 1.23 2010/09/22 18:59:43 pder Exp \$ into a string containing the name of the working file, the version, and the locker.

The list of predefined keywords embedded in an item when it is first checked in is contained in a file stored in the HeaderFile property.

All regular expressions in CMHeader\* properties match only one set of parentheses.

The PVCS default is \ \$Revision: +([0-9\.] +).

This command references the CheckInRevisionPart, ModePart, and LogPart properties; and the \$archive, \$archivedirectory, and \$unit internal variables

## **CommentsRequiredForCheckIn**

The CommentsRequiredForCheckIn property sets whether comments are required upon SCC Check In.

When this flag is set to “True”, comments are required for successful SCC Check In operation. By default this setting is set to "False".

## ConnectToCMRepository

The ConnectToCMRepository property specifies the command used to connect Rational Rhapsody to a configuration management archive. For some tools, this is simply an echo.

This is a command to run the Rational Rhapsody Executer with two arguments:

- The first argument consists of two executable commands: `move $rhpdirectory $rhpdirectory.orig`  
`cleartool mkelem -eltype directory -nc -nco $rhpdirectory`
- The first command backs up the repository (the `_rpy` directory for the user). The second command is defined within Rational ClearCase to create an element of type directory. The `-nc` option (no additional comment) creates an event record with no user-supplied comment string. The new directory points to the repository.
- The second argument, `".."`, tells the Executer to run the commands from the directory just above the current one.

A side-effect of the ConnectToCMRepository property is that it sets the Archive property to the location of the configuration management archive, even if a configuration management command is not actually executed.

The PVCS default value is as follows:

```
( $SupportTreeRepository ? echo "Connected to $archive" : findstr "VCSDIR" "$archive" )
```

## Delete

The Delete property specifies the script that deletes a particular item from the current Rational ClearCase directory element.

When you delete a Rational Rhapsody unit and the project settings indicate that the configuration management tool is Rational ClearCase, Rational Rhapsody runs this delete script to remove the deleted item from the Rational ClearCase view as well.

If the property does not exist, or if it is empty, the unit is removed from the Rational Rhapsody model, but not from the Rational ClearCase view.

By default, this feature is disabled. To enable it, set the DeleteActivation property.

The default value of the Delete property is as follows: `\"$OMROOT/etc/Executer.exe\" \"cleartool checkout -reserved -nc $dirs ; cleartool rmname -nc $units ; cleartool checkin -nc $dirs\"`

It uses the following keywords:

- `$units` - Specifies the full path names of the unit file names, separated by spaces
- `$dirs` - Specifies the names of the unit directories, separated by spaces

## DeleteActivation

The DeleteActivation property is a Boolean value that specifies whether deletion of units from the Rational Rhapsody model triggers a delete command in the archive.

The possible values are as follows:

- Disable - Disable the trigger.
- UserConfirmation - Prompt the user for confirmation before performing the deletion.
- Automatic - Automatically trigger the delete command in the archive when a unit is deleted in Rational Rhapsody.

*Default = Disable*

## DeleteDirectory

The DeleteDirectory property specifies the command to delete a directory in the Rational Clearcase configuration management system.

The PVCS default value is as follows:

```
"$OMROOT/etc/Executer.exe" "cleartool checkout -reserved -nc "$UnitDirPath\.." ; cleartool rmname -nc "$UnitDirPath" ; cleartool checkin -nc "$UnitDirPath\.."
```

## DiffInvocation

The DiffInvocation property specifies the command to run the external, textual DiffMerge tool.

For PVCS, this property runs the corresponding diff tool for the configuration management tool. For the other configuration management tools, this property is set to an empty string.

The product searches for the property value as follows:

- First, it searches through the configuration management property values.
- If the property is not found or if it is set to empty, Rational Rhapsody searches under General::DiffMerge.

By default, site-specific properties override the factory properties.

A new batch file, pvcsmerge.bat, has been added to the Rational Rhapsody installation, which runs the PVCS DiffMerge tool if the configuration management tool is set to PVCS.

SCC users who use PVCS Dimensions can set this property to the PVCS value because both PVCS Dimensions and PVCS Version Manager use the same DiffMerge tool on Windows systems.

The PVCS default is "\$OMROOT\etc\pvcsdiffmerge.bat" \$source1 \$source2.

## DiffMergeInvocation

The DiffMergeInvocation property specifies the command to run the external, textual Diff/Merge tool.

For Serena PVCS and Rational Clearcase, this property runs the corresponding diff tool for the configuration management tool. For the other configuration management tools, this property is set to an empty string.

The product searches for the property value as follows:

- First, it searches through the configuration management property values.
- If the property is not found or if it is set to empty, Rational Rhapsody searches under General::DiffMerge.

By default, site-specific properties override the factory properties.

A new batch file, pvcsmerge.bat, has been added to the Rational Rhapsody installation, which runs the PVCS Diff/Merge tool if the configuration management tool is set to PVCS.

SCC users who use PVCS Dimensions can set this property to the PVCS value because both PVCS Dimensions and PVCS Version Manager use the same Diff/Merge tool on Windows systems.

The PVCS default is \$DiffInvocation \$output.

## EnableSCCCancel

The EnableSCCCancel property is used to provide a cancel option during configuration management operations.

When set to True, the SCC provider displays its cancel window during configuration management operations.

The DeleteActivation property specifies whether deleting units from the Rational Rhapsody model triggers the delete command in the archive.

The possible values are as follows:

- Disable - The delete operation is disabled.
- UserConfirmation - The delete operation is performed only on user confirmation.
- Automatic - If you delete a unit from Rational Rhapsody, the unit is automatically removed from the archive without any notification.

*Default = Disable*

## Fetch

The Fetch property specifies the command used by the Rational Rhapsody Diff/Merge to fetch files from the configuration management tool.

The PVCS default is `get -P -C"$archive" -y -r$label $mode $unit >$targetDir\$unit`.

## FetchFromArchive

The `FetchFromArchive` property specifies the command used by the Rational Rhapsody Diff/Merge to fetch files from the configuration management archive.

*Default = \$Fetch*

## FooterFile

The `FooterFile` property specifies the file footer.

The PVCS default is `$OMROOT/cm/PVCSFooter.txt`.

## HeaderFile

The `HeaderFile` property specifies the file header. This property is reserved for future use.

The PVCS default is `$OMROOT/cm/PVCSHeader.txt`

## HeaderInfoItsRepositoryPath

The `HeaderInfoItsRepositoryPath` property is used for internal purposes only. Do not change this value.

The default value for PVCS is as follows: `VCSDIR([="0-9a-zA-Z:_.\|"]+)`

## InValidCharactersInRevisionDescription

The `InValidCharactersInRevisionDescription` property provides a list of invalid characters in the Revision Description (or comments) during a Check In operation. The list of invalid characters - provided by the user - validates the revision description and notifies the user if there are any invalid characters.

*Default = Blank*

## ListArchive

The `ListArchive` property specifies the command to list the contents of the archive.

The following example shows sample output from the `ListArchive` command:

```
version animPingPong.cfg@@\main\2 Rule: element * \main\LATEST version
animPingPongFlat.cfg@@\main\2 Rule: element * \main\LATEST version
animPingPongMultiThread.cfg@@\main\2 Rule: element * \main\LATEST version
```

Default.sbs@@\main\2 Rule: element \* \main\LATEST version DefaultConfig.cfg@@\main\2 Rule: element \* \main\LATEST version Model1.ond@@\main\4 Rule: element \* \main\LATEST version MSC1.msc@@\main\3 Rule: element \* \main\LATEST

An empty string in any of the ListArchive\* properties means that the configuration item does not contain this information; therefore, it is missing from the archive listing.

The PVCS default is `vlog -C"$archive" $ListArchiveRevisionPart *.??v`.

## ListArchiveItsLockedBy

The ListArchiveItsLockedBy property specifies the regular expression that extracts from the output of the List Archive command the name of the person who last locked a configuration item.

Consider the following PVCS expression: Locked by: +([0-9a-zA-Z]+)

This expression tells the interpreter to look in the header information for a “Locked by:” string beginning. Once the string is matched, only the regular expression between the brackets is taken as the “locker.”

## ListArchiveItsVersion

The ListArchiveItsVersion property specifies the version of the configuration management archive.

The PVCS default is `Rev ([0-9\.] +)`.

## ListArchiveItsWorkingFile

The ListArchiveItsWorkingFile specifies the regular expression that extracts from the output of the List Archive command the working file of an item.

The PVCS default is `v\([([0-9a-zA-Z:_.\|-]+)\)`.

## ListArchiveRevisionPart

The ListArchiveRevisionPart property is a string that represents the revision part from the output of the List Archive command.

The PVCS default is `( $label ? -br $label : -br )`.

## LockItem

The LockItem property is a string that specifies the command tool used to lock an item in the archive.

The PVCS default is as follows:

`( $SupportTreeRepository ? vcs -C"$archive" -y -l$label $unit : vcs -C"$archive" -y -l$label`

"\$UnitArchiveDir"("\$UnitPath") ).

## LogPart

The LogPart property specifies how to embed a log, if provided, in a configuration management command. The log is the comment entered in the Revision/Description field in the Check In window.

The PVCS default is ( \$log ? -m\$log : -m' ' ).

## MakeCMSShadowDir

The MakeCMSShadowDir property specifies the Rational ClearCase command to create a directory as a VOB element.

The SCC default value is as follows:

```
"$OMROOT/etc/Executer.exe" "cleartool checkout -reserved -nc "$parentdir" ; move "$fulldir"
"$fulldir.orig" ; cleartool mkelem -eltype directory -nc -nco "$fulldir" ; cleartool checkin -nc "$parentdir"
; copy "$fulldir.orig" "$fulldir" ; " .."
```

## MergeOutput

The MergeOutput property specifies the file that is to hold the results of a merge operation.

The PCVS default value is \$temp\out.txt.

## MultiRecordDelimiter

The MultiRecordDelimiter property specifies the regular expression representing the symbol used to separate multiple records in an archive listing.

The PCVS default value is =+.

## OperationErrorPattern

The OperationErrorPattern property notifies you that the specified error occurred during batch mode. Rhapsody searches all configuration management operation output for the string specified in this property. Rhapsody first checks for errors, then for warnings.

Note the following:

- The string pattern is not case-sensitive.
- It can be a regular expression.

*Default = empty string*



## OperationWarningPattern

The OperationWarningPattern property notifies you that the specified warning occurred during batch mode. Rhapsody searches all configuration management operation output for the string specified in this property. Rhapsody first checks for errors, then for warnings.

Note the following:

- The string pattern is not case-sensitive.
- It can be a regular expression.

*Default = empty string*

## PostConnectToCMRepository

The PostConnectToCMRepository property is used for internal purposes only. The default is ( \$ArchiveRoot ? "\$OMROOT/etc/Executer.exe" "md "\$ArchiveRoot\projectname\_rpy" " : ).

Do NOT change the value of this property.

## ReadOnly

The ReadOnly property specifies how to embed a ReadOnly flag in the configuration management command.

The default value for PVCS is an empty string (blank).

## ReadWrite

The ReadWrite property specifies how to embed a ReadWrite flag in the configuration management command.

*Default = "-l"*

This command references the LogPart property, which in turn references the internal variable \$log, and the variable \$unit.

## Repository

The Repository property is used for internal purposes only. Do not change the value of this property.

The PVCS default is ( \$ArchiveRoot ? "\$ArchiveRoot\$SubDirs" : ).

## SaveOnCheckOut

The SaveOnCheckOut property is a Boolean value that specifies whether a Rational Rhapsody save should be triggered whenever a checkout occurs.

*Default = Checked*

## ShowNewItemsInSynchronize

The ShowNewItemsInSynchronize property is a Boolean value that is directly related to what you see in the Synchronize window.

If this property is set to No, new items that are added (by another member of the team) to the archive after the Rational Rhapsody project is open are not displayed.

*Default = Yes*

## SupportTreeRepository

The SupportTreeRepository property is provided for compatibility with previous versions of Rational Rhapsody.

For PVCS, this property is for internal purposes only. Do not change the value of this property.

If you checked in Rational Rhapsody projects to an SCC archive prior to Version 4.0, you must perform the following steps:

- Create a directory in the configuration management tool with the same name of the directory that holds the .rpy file.
- Disconnect from the existing archive.
- Change the value of this property to an empty string.
- Reconnect to the archive.

The default value for PVCS is No.

## UnLockItem

The UnLockItem property is a string that specifies the command used to release a lock placed on an item in the archive.

The SCC default value is as follows:

```
( $SupportTreeRepository ? vcs -C"$sarchive" -y -u$label $unit : vcs -C"$sarchive" -y -u$label "$UnitArchiveDir"("$UnitPath") )
```

# RationalTeamConcert

Contains properties related to the direct CM integration with Rational Team Concert (RTC).

## DeleteActivation

The property DeleteActivation controls how the deletion of Rhapsody units is handled in terms of configuration management. The possible values for the property are:

- Disable - when a unit is deleted from a model, no corresponding action is taken in the configuration management system
- UserConfirmation - when a unit is deleted from a model, the user is asked whether the file should also be deleted in the configuration management system
- Automatic - when a unit is deleted from a model, the deletion is also carried out in the configuration management system

*Default = UserConfirmation*

## RenameActivation

The property RenameActivation controls how the renaming of Rhapsody units is handled in terms of configuration management. The possible values for the property are:

- Disable - when a unit is renamed, no corresponding change is made in the configuration management system
- UserConfirmation - when a unit is renamed, the user is asked whether the name change should also be made in the configuration management system
- Automatic - when a unit is renamed, the name change is also made in the configuration management system

*Default = UserConfirmation*

## ShowCMStatus

If you are using the direct CM-integration with Rational Team Concert, you have the option of displaying status icons in the Rhapsody browser to reflect the CM status of the element's file in RTC, for example, Outgoing, Incoming, or Unresolved.

To have the statuses displayed in the browser, set the value of the ShowCMStatus property to True.

*Default = False*

## UnitAutomaticChangeSetComment

If the property UnitAutomaticOperationOnSave is set to CheckIn or CheckInAndDeliver, the value of the property UnitAutomaticChangeSetComment is used as a default comment for change sets that have been automatically checked-in to RTC after the files were modified in Rhapsody. This string can be used to easily identify such change sets on the Pending Changes tab.

*Default = ?<IsConditionalProperty>Rhapsody Files Check-In: \$<ProjectName>*

## **UnitAutomaticOperationOnSave**

The property UnitAutomaticOperationOnSave is used to specify that when modifications to a unit are saved in Rhapsody, the unit should automatically be checked-in to RTC, or automatically checked-in and delivered.

*Default = CheckIn*

## **UnitLockIfReadWrite**

When the property UnitLockIfReadWrite is set to True, files are locked on RTC if the corresponding unit is set to read/write in the Unit Information window in Rhapsody. Similarly, files are unlocked if the corresponding unit is set to read-only in the Unit Information window.

*Default = True*

# **SCC**

The SCC metaclass contains properties that enable you to use the SCC interface with Rational Rhapsody.

## **AddNewUnitsToArchiveDuringCheckin**

This property controls whether new subunits that might have been added to a checked out unit are automatically added to your configuration management archive during checkin if the parent unit is checked in with descendants. When this property is set to Checked, new subunits that were added to a checked out unit are automatically added to your configuration management archive during checkin if the parent unit is checked in with descendants.

*Default = Cleared*

## **AskOnCheckoutReadWrite**

The AskOnCheckoutReadWrite property is a Boolean value that specifies whether Rational Rhapsody should prompt users if they want to perform a check out (because they already have a read/write copy of the file).

The default value for SCC is No.

## **AskOnLoadFromArchive**

The AskOnLoadFromArchive property specifies whether to display a message when you check a file out

of an archive that is not part of the current model.

If this property is set to Yes, Rational Rhapsody displays a window when you perform a checkout from the archive (or fetch operation for SCC) from either the ListArchive or Synchronize window.

When you check out a file from the archive and it is not the part of the current model, there are two possibilities:

- The corresponding parent unit is not loaded in the model.
- The parent unit, if loaded, is not the latest one.

Therefore, before Rational Rhapsody checks out an element from an archive, it checks whether the file is already part of the model. If not, it checks whether there is a stub unit in the model that refers to the same file. If neither criteria is met, Rational Rhapsody displays an informational message.

*Default = Yes*

## AuxProjPath

The AuxProjPath property is a string that identifies the SCC project path. Do not change this property. Removing the property value disconnects the link between the Rational Rhapsody project and the SCC project.

If you remove this value, you must also remove the ProjName property.

*Default = empty string*

## BaseAwareAutoMergeInvocation

The BaseAwareAutoMergeInvocation property specifies how to run the external textual DiffMerge tool which supports a base-aware detection of triviality of textual difference and a base-aware automatic merging between a base unit file and two other unit files.

An automatic model merge operation can be performed only if all the differences in the current diff session are trivial differences. If two units are being compared with a base unit, this three-way comparison makes it possible for the DiffMerge tool to determine automatically the need for some merges by using the concept of trivial versus non-trivial differences. For a difference in which only one unit differs from the base unit, it is identified as a non-conflicting difference or trivial difference. Similarly, if both of the units are different from base contributor but the differences are same, then it is also a trivial difference. This applies to differences between model elements or between the attributes of model elements. However, if one unit contains a difference that does not appear in either the base unit or the other unit being compared, this is a non-trivial difference that must be resolved by the developer. During the automatic merge, all the differences is automatically accepted. There is no default value.

## BaseAwareDiffInvocation

The BaseAwareDiffInvocation property specifies how to run an external textual DiffMerge tool supporting base-aware comparison and merging in Base Aware Diff mode, between a base unit file and two other unit files.

Default = Blank

## BaseAwareDiffMergeInvocation

The BaseAwareDiffMergeInvocation property specifies how to run an external textual DiffMerge tool supporting base-aware comparison and merging in Base Aware Merge mode between a base unit file and two other unit files.

## CallCheckOutOnSynchronize

The CallCheckOutOnSynchronize property is a Boolean value that determines whether a checkout operation when you specify synchronization in Rational Rhapsody. Some configuration management tools require a full checkout of the selected elements, followed by an Add to Model operation. However, Rational ClearCase does not require a checkout. Put a Check in this property check box to ensure a checkout operation.

*Default = Cleared*

## CheckIn

The CheckIn property specifies the command used to check an item into the archive through the use of the main Configuration Items window.

This command is specific to the configuration management tool in use.

The PVCS default is as follows:

```
( $SupportTreeRepository ? put -C"$archive" -y $CheckInRevisionPart $mode $LogPart $unit : put  
-C"$archive" -y $CheckInRevisionPart $mode $LogPart "$UnitArchiveDir("$UnitPath") ).
```

## CheckInRevisionPart

The CheckInRevision property defines a new revision number used to check in a unit by using the CheckIn property. The value for \$CheckInRevisionPart is taken from the Label field in the window for checking in a unit that displays after selecting FileConfiguration Items, selecting a unit, and then selecting Check In the selected items.

The SCC default value is ( \$label ? -v\$label ).

## CheckOut

The CheckOut property specifies the command used to check an item out of the archive through the use of the main Configuration Items window.

The SCC default value is as follows:

```
( $SupportTreeRepository ? get -C"$archive" -y -r$label $mode $unit : get -C"$archive" -y -r$label $mode "$UnitArchiveDir"("$UnitPath") ).
```

## CheckOutFromArchive

The CheckOutFromArchive property specifies the command used to check an item out of the archive directly through the use of the List Archive window. For all the currently supported tools (Rational ClearCase and Serena PVCS Dimensions), this command is the same as that stored in the CheckOut property. Therefore, its value is \$CheckOut.

*Default = \$CheckOut*

## CheckOutRevisionPart

The CheckOutRevisionPart property defines a revision number used to check out a unit by using the CheckOut property. The value for \$CheckOutRevisionPart is taken from the Revision/Label field in the window for checking out a unit that displays after selecting FileConfiguration Items, selecting a unit, and then selecting Check Out the selected items.

*Default = ( \$label ? -r \$label : -h )*

## CommentsRequiredForCheckIn

The CommentsRequiredForCheckIn property sets whether comments are required upon SCC Check In. When this flag is set to Checked, comments are required for successful SCC Check In operation. The default is Cleared.

## ConnectToCMRepository

The ConnectToCMRepository property specifies the command used to connect Rational Rhapsody to a configuration management archive. For some tools, this is simply an echo.

This is a command to run the Rational Rhapsody Executer with two arguments:

- The first argument consists of two executable commands: `move $rhpdirectory $rhpdirectory.orig cleartool mkelem -eltype directory -nc -nco $rhpdirectory`
- The first command backs up the repository (the `_rpy` directory for the user). The second command is defined within Rational ClearCase to create an element of type directory. The `-nc` option (no additional comment) creates an event record with no user-supplied comment string. The new directory points to the repository.
- The second argument, `".."`, tells the Executer to run the commands from the directory just above the current one.

A side-effect of the ConnectToCMRepository property is that it sets the Archive property to the location of the configuration management archive, even if a configuration management command is not actually executed.

The SCC default value is as follows:

```
( $SupportTreeRepository ? echo "Connected to $archive" : findstr "VCSDIR" "$archive" )
```

## Delete

The Delete property specifies the script that deletes a particular item from the current Rational ClearCase directory element.

When you delete a Rational Rhapsody unit and the project settings indicate that the configuration management tool is Rational ClearCase, Rational Rhapsody runs this delete script to remove the deleted item from the ClearCase view as well.

If the property does not exist, or if it is empty, the unit is removed from the Rational Rhapsody model, but not from the Rational ClearCase view.

By default, this feature is disabled. To enable it, set the DeleteActivation property.

The default value of the Delete property is as follows: `\"$OMROOT/etc/Executer.exe\" \"cleartool checkout -reserved -nc $dirs ; cleartool rmname -nc $units ; cleartool checkin -nc $dirs\"`

It uses the following keywords:

- \$units - Specifies the full path names of the unit file names, separated by spaces
- \$dirs - Specifies the names of the unit directories, separated by spaces

## DeleteActivation

The property DeleteActivation controls how the deletion of Rhapsody units is handled in terms of configuration management. The possible values for the property are:

- Disable - when a unit is deleted from a model, no corresponding action is taken in the configuration management system
- UserConfirmation - when a unit is deleted from a model, the user is asked whether the file should also be deleted in the configuration management system
- Automatic - when a unit is deleted from a model, the deletion is also carried out in the configuration management system

*Default = Disable*

## DeleteDirectory

The DeleteDirectory property specifies the command to delete a directory in the Rational Clearcase configuration management system.

The SCC default value is as follows:

```
"$OMROOT/etc/Executer.exe" "cleartool checkout -reserved -nc "$UnitDirPath\" ; cleartool rmname -nc
```



"\$UnitDirPath" ; cleartool checkin -nc "\$UnitDirPath\.."

## DiffInvocation

The DiffInvocation property specifies the command to run the external, textual DiffMerge tool.

For SCC, this property is set to an empty string.

The product searches for the property value as follows:

- First, it searches through the configuration management property values.
- If the property is not found or if it is set to empty, Rational Rhapsody searches under General::DiffMerge.

By default, site-specific properties override the factory properties.

A new batch file, pvcsmerge.bat, has been added to the Rational Rhapsody installation, which runs the PVCS DiffMerge tool if the configuration management tool is set to PVCS.

SCC users who use Serena PVCS Dimensions can set this property to the PVCS value because both PVCS Dimensions and Serena PVCS Version Manager use the same DiffMerge tool on Windows systems.

*Default = Blank*

## DiffMergeInvocation

The DiffMergeInvocation property specifies the command to run an external, textual Diff/Merge tool.

SCC users who use Serena PVCS Dimensions can set this property to the PVCS value because both Serena PVCS Dimensions and Serena PVCS Version Manager use the same Diff/Merge tool on Windows systems.

## EnableSCCCancel

The EnableSCCCancel property is used to provide a cancel option during configuration management operations.

When set to Checked, the SCC provider displays its cancel window during configuration management operations.

The DeleteActivation property specifies whether deleting units from the Rational Rhapsody model triggers the delete command in the archive.

The possible values are as follows:

- Cleared - The delete operation is disabled.
- Checked - If you delete a unit from Rational Rhapsody, the unit is automatically removed from the archive without any notification.

*Default = Cleared*

## **IgnoreAddToArchiveForExistingUnits**

This property controls whether the Add to Archive operation in SCC mode is ignored if units already exist in your configuration management archive. When this property is set to Checked, the Add to Archive command is ignored if units already exist in your configuration management archive. A message displays on the Configuration Management tab of the Output window to this effect.

If you set the property to Cleared, a pop-up error message window might open instead, which you then have to close before you can continue.

*Default = Checked*

## **IgnoreUndoCheckoutForNotCheckedoutUnits**

This property controls whether the Undo Checkout operation in SCC mode is ignored if a unit is not already checked out. When this property is set to Checked, the Undo Checkout operation is ignored if a unit is not already checked out. A message displays on the Configuration Management tab of the Output window to this effect.

If you set the property to Cleared, a pop-up error message window might open instead, which you then have to close before you can continue.

*Default = Checked*

## **MergeOutput**

The MergeOutput property specifies the file that is to hold the results of a merge operation.

The SCC default value is empty string.

## **MoveActivation**

The MoveActivation property is a Boolean value that specifies whether moving units in the Rational Rhapsody model (in a way that changes the unit file location on the hard drive) triggers a move command in the archive.

The possible values are as follows:

- Disable - Disable this functionality.
- UserConfirmation - Prompt the user for confirmation before moving the elements in the archive.
- Automatic - Automatically move elements whenever units are moved in the Rational Rhapsody model.

*Default = Disable*

## ProjName

The ProjName property is a string that identifies the SCC project name. Do not change this property. Removing the property value disconnects the link between the Rational Rhapsody project and the SCC project.

If you remove this value, you must also remove the AuxProjPath property.

*Default = empty string*

## RedirectOutputToRhapsody

The UseSCCTool property is a Boolean value that specifies whether the resulting output of an SCC configuration management command should be redirected (displayed) in the Rational Rhapsody CM output window tab.

*Default = Checked*

## RefreshCMStatusAtProjectOpenup

RefreshCMStatusAtProjectOpenup is a property that determines whether or not the configuration management status of project units is updated from the configuration management repository when a project is opened. This property is set at the project level. The value of this property is only relevant when the ConfigurationManagement::SCC::ShowCMStatus property is set to true.

The possible values are Yes, No, and Ask User.

*Default = Ask User*

## RenameActivation

The property RenameActivation controls how the renaming of Rhapsody units is handled in terms of configuration management. The possible values for the property are:

- Disable - when a unit is renamed, no corresponding change is made in the configuration management system
- UserConfirmation - when a unit is renamed, the user is asked whether the name change should also be made in the configuration management system
- Automatic - when a unit is renamed, the name change is also made in the configuration management system

*Default = Disable*

## SaveOnCheckOut

The SaveOnCheckOut property is a Boolean value that specifies whether a Rational Rhapsody save should

be triggered whenever a checkout occurs.

*Default = Checked*

## ShowCMStatus

The ShowCMStatus property property that determines whether or not Rational Rhapsody displays the configuration management status of project units. This property is set at the project level.

*Default = Cleared*

## StoreInSeparateDirectoryActivation

The StoreInSeparateDirectoryActivation property affects how an existing package is converted. The following conditions apply with this property only if the SCC tool is Rational ClearCase:

- Enable this property by selecting UserConfirmation or Automatic from the drop-down list. When an existing flat package is converted to a package as a directory, the directory is created on the configuration management side and the package with its children moved to this directory.
- Disable this property by selecting Disable. When an existing flat package is converted to a package as a directory, the directory is removed on the configuration management side and the children of this package are removed as well.

*Default = Disable*

## SupportTreeRepository

The SupportTreeRepository property is provided for compatibility with previous versions of Rational Rhapsody.

For PVCS, this property is for internal purposes only. Do not change the value of this property.

If you checked in Rational Rhapsody projects to an SCC archive prior to Version 4.0, you must perform the following steps:

- Create a directory in the configuration management tool with the same name of the directory that holds the .rpy file.
- Disconnect from the existing archive.
- Change the value of this property to an empty string.
- Reconnect to the archive.

The default for SCC is an empty string.

## UnrollLoops

When you perform a configuration management operation, the UnrollLoops property specifies whether Rational Rhapsody initiates a single SCC call or multiple SCC calls for the operation.

If this property is enabled, a single SCC call is issued. However, note that including descendants when checking out cannot be handled in a single SCC call, so it is an exception.

*Default = Cleared*

## SourceIntegrity

The SourceIntegrity metaclass contains properties that enable you to use the SourceIntegrity implementation with Rational Rhapsody.

### AddMember

The AddMember property specifies the command used to add an item to the archive.

The default is ( \$SupportTreeRepository ? "\$OMROOT/etc/Executer.exe" "copy \$unit \"\$archivedirectory\\.\"; pj add -y -P \"\$archive\" \"\$archivedirectory\$unit\" : "\$OMROOT/etc/Executer.exe" "copy \$unitPath \"\$archivedirectory\$unitdirectory\\.\"; pj add -y -P \"\$archive\" \"\$archivedirectory\$unitdirectory\$unit\" )

### Archive

The Archive property specifies the archive file for a project. This property is implicitly set when you use the Connect to Archive option within Rational Rhapsody, which uses the command string specified by the ConnectToCMRepository property.

Do not set the Archive property manually.

*Default = \$currentdirectory*

### ArchiveSelection

The ArchiveSelection property specifies whether the archive façade is a file or directory. Most configuration management tools expect a file, whereas others expect a directory.

The Browse button in the Connect to Archive window runs a different browse utility for file-based and directory-based archives. When the ArchiveSelection property is set to File, the Open window for files is displayed by default. In this case, you cannot select a directory as the archive.

If the archive is a directory rather than a file, set the ArchiveSelection property to Directory. The Browse for Folder window is displayed instead, which allows you to select a directory.

The possible values are as follows:

- Directory - The configuration management archive is a directory.
- File - The configuration management archive is a file. This is the default for SourceIntegrity.

- None - Neither a file nor directory is expected.

## AskOnCheckoutReadWrite

The AskOnCheckoutReadWrite property is a Boolean value that specifies whether Rational Rhapsody should prompt users if they want to perform a check out (because they already have a read/write copy of the file).

*Default = Yes*

## AskOnLoadFromArchive

The AskOnLoadFromArchive property specifies whether to display a message when you check a file out of an archive that is not part of the current model.

If this property is set to Yes, Rational Rhapsody displays a window when you perform a checkout from the archive (or fetch operation for SCC) from either the ListArchive or Synchronize window.

When you check out a file from the archive and it is not the part of the current model, there are two possibilities:

- The corresponding parent unit is not loaded in the model.
- The parent unit, if loaded, is not the latest one.

Therefore, before Rational Rhapsody checks out an element from an archive, it checks whether the file is already part of the model. If not, it checks whether there is a stub unit in the model that refers to the same file. If neither criteria is met, Rational Rhapsody displays an informational message.

*Default = Yes*

## BaseAwareAutoMergeInvocation

The BaseAwareAutoMergeInvocation property specifies how to run the external textual DiffMerge tool which supports a base-aware detection of triviality of textual difference and a base-aware automatic merging between a base unit file and two other unit files.

An automatic model merge operation can be performed only if all the differences in the current diff session are trivial differences. If two units are being compared with a base unit, this three-way comparison makes it possible for the DiffMerge tool to determine automatically the need for some merges by using the concept of trivial versus non-trivial differences. For a difference in which only one unit differs from the base unit, it is identified as a non-conflicting difference or trivial difference. Similarly, if both of the units are different from base contributor but the differences are same, then it is also a trivial difference. This applies to differences between model elements or between the attributes of model elements. However, if one unit contains a difference that does not appear in either the base unit or the other unit being compared, this is a non-trivial difference that must be resolved by the developer. During the automatic merge, all the differences is automatically accepted. There is no default value.

## BaseAwareDiffInvocation

The BaseAwareDiffInvocation property specifies how to run an external textual DiffMerge tool supporting base-aware comparison and merging in Base Aware Diff mode, between a base unit file and two other unit files.

*Default = Blank*

## BaseAwareDiffMergeInvocation

The BaseAwareDiffMergeInvocation property specifies how to run an external textual DiffMerge tool supporting base-aware comparison and merging in Base Aware Merge mode between a base unit file and two other unit files.

## CMHeaderItsLockedBy

The CMHeaderItsLockedBy property specifies the regular expression used to extract the name of the person who last locked the unit.

For all tools, an empty string in CMHeader\* means that the unit does not contain this type of information, therefore it is missing from the Configuration Items window when you select the List Archive option.

All properties of the format CMHeader\* are regular expressions that extract header information embedded into a configuration item when it is first checked into the archive. Embedded information can include such things as the ID of the item, its revision number, and the name of the person who last locked it (the “locker”).

Header information is contained in predefined, tool-specific keywords, which the various configuration management tools know how to expand. For example, PVCS knows how to expand the keyword \$Header: /cvsroot/telelogic/Rhapsody/help.core/com.ibm.rhapsody.property.definition.doc/ConfigurationManagement.xml,v 1.23 2010/09/22 18:59:43 pder Exp \$ into a string containing the name of the working file, the version, and the locker.

The list of predefined keywords embedded in an item when it is first checked in is contained in a file stored in the HeaderFile property.

All regular expressions in CMHeader\* properties match only one set of parentheses.

## CMHeaderItsVersion

The CMHeaderItsVersion property specifies the regular expression that extracts the version information.

Consider the following PVCS expression: \ \$Revision: +([0-9\.]++)

This expression searches the header for a string that begins with \ \$Revision: and contains a version number that can consist of one or more digits 0 through 9, the backslash character (\), or a period.

All properties of the format CMHeader\* are regular expressions that extract header information embedded into a configuration item when it is first checked into the archive. Embedded information can include such things as the ID of the item, its revision number, and the name of the person who last locked it (the “locker”).

Header information is contained in predefined, tool-specific keywords, which the various configuration management tools know how to expand. For example, PVCS knows how to expand the keyword \$Header: /cvsroot/telelogic/Rhapsody/help.core/com.ibm.rhapsody.property.definition.doc/ConfigurationManagement.xml,v 1.23 2010/09/22 18:59:43 pder Exp \$ into a string containing the name of the working file, the version, and the locker.

The list of predefined keywords embedded in an item when it is first checked in is contained in a file stored in the HeaderFile property.

All regular expressions in CMHeader\* properties match only one set of parentheses.

This command references the CheckInRevisionPart, ModePart, and LogPart properties; and the \$archive, \$archivedirectory, and \$unit internal variables

## CheckIn

The CheckIn property specifies the command used to check an item into the archive through the use of the main Configuration Items window.

This command is specific to the configuration management tool in use.

The default is as follows:

```
( $SupportTreeRepository ? pj ci -y -t " $CheckInRevisionPart $ModePart $LogPart -P "$archive" -w .  
"$archivedirectory/$unit" : pj ci -y -t " $CheckInRevisionPart $ModePart $LogPart -P "$archive" -w .  
"$archivedirectory$UnitDirectory/$unit" ).
```

## CheckInRevisionPart

The CheckInRevision property defines a new revision number used to check in a unit by using the CheckIn property. The value for \$CheckInRevisionPart is taken from the Label field in the window for checking in a unit that displays after selecting FileConfiguration Items, selecting a unit, and then selecting Check In the selected items.

The default is ( \$label ? -N \$label ).

## CheckOut

The CheckOut property specifies the command used to check an item out of the archive through the use of the main Configuration Items window.

The default value is as follows:

```
( $SupportTreeRepository ? pj co -y $CheckOutRevisionPart $ModePart -P "$archive" -w .  
"$archivedirectory/$unit" : pj co -y $CheckOutRevisionPart $ModePart -P "$archive" -w .  
"$archivedirectory$UnitDirectory/$unit" ).
```



## CheckOutFromArchive

The CheckOutFromArchive property specifies the command used to check an item out of the archive directly through the use of the List Archive window. For all the currently supported tools (Rational ClearCase and Serena PVCS Dimensions), this command is the same as that stored in the CheckOut property. Therefore, its value is \$CheckOut.

*Default = \$CheckOut*

## CheckOutRevisionPart

The CheckOutRevisionPart property defines a revision number used to check out a unit by using the CheckOut property. The value for \$CheckOutRevisionPart is taken from the Revision/Label field in the window for checking out a unit that displays after selecting FileConfiguration Items, selecting a unit, and then selecting Check Out the selected items.

*Default = ( \$label ? -r \$label : -h )*

## ConnectToCMRepository

The ConnectToCMRepository property specifies the command used to connect Rational Rhapsody to a configuration management archive. For some tools, this is simply an echo.

This is a command to run the Rational Rhapsody Executer with two arguments:

- The first argument consists of two executable commands:
- move \$rhpdirectory \$rhpdirectory.orig
- cleartool mkelem -eltype directory -nc -nco \$rhpdirectory

The first command backs up the repository (the \_rpy directory for the user). The second command is defined within Rational ClearCase to create an element of type directory. The -nc option (no additional comment) creates an event record with no user-supplied comment string. The new directory points to the repository.

The second argument, "..", tells the Executer to run the commands from the directory just above the current one.

A side-effect of the ConnectToCMRepository property is that it sets the Archive property to the location of the configuration management archive, even if a configuration management command is not actually executed.

## DiffInvocation

The DiffInvocation property specifies the command to run the external, textual DiffMerge tool.

For SourceIntegrity, this property is set to an empty string.

The product searches for the property value as follows:

- First, it searches through the configuration management property values.
- If the property is not found or if it is set to empty, Rational Rhapsody searches under General::DiffMerge.

By default, site-specific properties override the factory properties.

*Default = Blank*

## **DiffMergeInvocation**

The DiffMergeInvocation property specifies the command to run an external, textual Diff/Merge tool.

SCC users who use PVCS Dimensions can set this property to the PVCS value because both PVCS Dimensions and PVCS Version Manager use the same Diff/Merge tool on Windows systems.

## **Fetch**

The Fetch property specifies the command used by the Rational Rhapsody Diff/Merge to fetch files from the configuration management tool.

The default is `pj co -y -p $CheckoutRevisionPart $ModePart -P "$archive" -w . "$archivedirectory$UnitDirectory/$unit" >$targetDir\$unit`.

## **FetchFromArchive**

The FetchFromArchive property specifies the command used by the Rational Rhapsody Diff/Merge to fetch files from the configuration management archive.

*Default = \$Fetch*

## **FooterFile**

The FooterFile property specifies the file footer.

The default is \$OMROOT/cm/SIFooter.txt.

## **HeaderFile**

The HeaderFile property specifies the file header. This property is reserved for future use.

The default is \$OMROOT/cm/SIHeader.txt.

## **HeaderInfoRepositoryPath**

The HeaderInfoItsRepositoryPath property is used for internal purposes only. Do not change this value.

The default value is as follows: Repository: ([0-9a-zA-Z:\_.\]+)

## **InValidCharactersInRevisionDescription**

The InValidCharactersInRevisionDescription property provides a list of iInvalid characters in the Revision Description (or comments) during a Check In operation. The list of invalid characters - provided by the user - validates the revision description and notifies the user if there are any invalid characters. By default this property value is blank.

## **ListArchive**

The ListArchive property specifies the command to list the contents of the archive.

The following example shows sample output from the ListArchive command:

```
version animPingPong.cfg@@\main\2 Rule: element * \main\LATEST version  
animPingPongFlat.cfg@@\main\2 Rule: element * \main\LATEST version  
animPingPongMultiThread.cfg@@\main\2 Rule: element * \main\LATEST version  
Default.sbs@@\main\2 Rule: element * \main\LATEST version DefaultConfig.cfg@@\main\2 Rule:  
element * \main\LATEST version Model1.umd@@\main\4 Rule: element * \main\LATEST version  
MSC1.msc@@\main\3 Rule: element * \main\LATEST
```

An empty string in any of the ListArchive\* properties means that the configuration item does not contain this information; therefore, it is missing from the archive listing.

## **ListArchiveItsLockedBy**

The ListArchiveItsLockedBy property specifies the regular expression that extracts from the output of the List Archive command the name of the person who last locked a configuration item.

Consider the following expression: Locked by: +([0-9a-zA-Z]+)

This expression tells the interpreter to look in the header information for a “Locked by:” string beginning. Once the string is matched, only the regular expression between the brackets is taken as the “locker.”

## **ListArchiveItsVersion**

The ListArchiveItsVersion property specifies the version of the configuration management archive.

The default is Revision: ([0-9\.] +).

## **ListArchiveItsWorkingFile**

The ListArchiveItsWorkingFile specifies the regular expression that extracts from the output of the List

Archive command the working file of an item.

The default is Archive File: ([0-9a-zA-Z: \_\.\|-]+).

## ListArchiveRevisionPart

The ListArchiveRevisionPart property is a string that represents the revision part from the output of the List Archive command.

The default is ( \$label ? -r \$label : -h ).

## LockItem

The LockItem property is a string that specifies the command tool used to lock an item in the archive.

The default is pj lock -y \$CheckoutRevisionPart -P "\$archive" "\$archivedirectory/\$unit".

## LogPart

The LogPart property specifies how to embed a log, if provided, in a configuration management command. The log is the comment entered in the Revision/Description field in the Check In window.

The default is ( \$log ? -m\$log : -m " ).

## MergeOutput

The MergeOutput property specifies the file that is to hold the results of a merge operation.

The default value is an empty string (blank).

## ModePart

The ModePart property specifies the locking mode of a configuration item. This is defined as: \$mode

If the item is locked, \$mode is replaced by the value of the ReadWrite property; otherwise, it is replaced by the value of the ReadOnly property.

## Move

The Move property specifies the Rational ClearCase command for a unit move.

The default value is as follows: "\$OMROOT/etc/Executer.exe" "cleartool checkout -reserved -nc "\$solddir" "\$newdir"; cleartool mv -nc "\$oldName" "\$newName" ; cleartool checkin -nc "\$solddir" "\$newdir" "

## MultiRecordDelimiter

The MultiRecordDelimiter property specifies the regular expression representing the symbol used to separate multiple records in an archive listing.

*Default = The*

## OperationErrorPattern

The OperationErrorPattern property notifies you that the specified error occurred during batch mode. Rhapsody searches all configuration management operation output for the string specified in this property. Rhapsody first checks for errors, then for warnings.

Note the following:

- The string pattern is not case-sensitive.
- It can be a regular expression.

*Default = empty string*

## OperationWarningPattern

The OperationWarningPattern property notifies you that the specified warning occurred during batch mode. Rhapsody searches all configuration management operation output for the string specified in this property. Rhapsody first checks for errors, then for warnings.

Note the following:

- The string pattern is not case-sensitive.
- It can be a regular expression.

*Default = empty string*

## ReadOnly

The ReadOnly property specifies how to embed a ReadOnly flag in the configuration management command.

The default value is an empty string (blank).

## ReadWrite

The ReadWrite property specifies how to embed a ReadWrite flag in the configuration management command.

*Default = "-l"*

This command references the LogPart property, which in turn references the internal variable \$log, and the variable \$unit.

## Repository

The Repository property is used for internal purposes only. Do not change the value of this property.

## SaveOnCheckOut

The SaveOnCheckOut property is a Boolean value that specifies whether a Rational Rhapsody save should be triggered whenever a checkout occurs.

*Default = Checked*

## SupportTreeRepository

The SupportTreeRepository property is provided for compatibility with previous versions of Rational Rhapsody.

If you checked in Rational Rhapsody projects to an SCC archive prior to Version 4.0, you must perform the following steps:

- Create a directory in the configuration management tool with the same name of the directory that holds the .rpy file.
- Disconnect from the existing archive.
- Change the value of this property to an empty string.
- Reconnect to the archive.

*Default = No*

## UnLockItem

The UnLockItem property is a string that specifies the command used to release a lock placed on an item in the archive.

The default value is as follows:

```
(pj unlock -y $CheckOutRevisionPart -P "$archive" "$archivedirectory/$unit"
```

## Synergy

The Synergy metaclass contains properties that control the interaction of Rational Rhapsody with Rational Synergy.

## AssignedTasksItsTaskId

This property specifies the regular expression that extracts the Task ID from the output of the ListAssignedTasks command.

When you create a task, Rational Synergy names it, by default, as Task task\_number. However, when you configure your DCM server, you can set it to insert a prefix before task\_number. In this case, you might want to update this property to use the regular expression you want. For example, if the prefix used is ukan#, then the value of this property should be changed to Task ukan#([0-9\..]+); otherwise the default is Task ([0-9\..]+).

The default is Task ([0-9\..]+).

## AssignedTasksItsTitle

This property specifies the regular expression that extracts the Task Title from the output of the ListAssignedTasks command.

The default is Task (.\*)

## CheckinCurrentTask

This property specifies the command used to check in the current (default) Rational Synergy task.

*Default = "\$OMROOT/etc/Executer.exe" "ccm task -checkin default -comment \"default task checked in from Rational Rhapsody\" -y"*

## CreateTask

This property specifies the command used to create a Rational Synergy task.

*Default = "\$OMROOT/etc/Executer.exe" "ccm task -create -gui"*

## GetCurrentTask

This property specifies the command used to get current (default) Rational Synergy task of the user.

*Default = ccm task -default*

## GetCurrentTaskItsTaskId

This property specifies the regular expression that extracts the Task ID from the output of the GetCurrentTask command.

The default is (([^\#]\*#)?[0-9\..]+).

## ListAssignedTasks

This property specifies the command used to list the Rational Synergy tasks assigned to the current user.

*Default = "\$OMROOT/etc/Executer.exe" "ccm task -query -task\_scope all\_my\_assigned"*

## LoadTaskOnOpenProject

If this value is set to Checked, Rational Rhapsody loads the task list when opening a project. If the property is set to Cleared, use the refresh button to load the tasks after loading the project.

This is useful when loading the tasks slows down “open project” because of the CM server is in a remote location and loading the task takes a while. The default value of the property is Checked.

## MultiRecordDelimiter

This property specifies the regular expression representing the symbol used to separate multiple records in an archive listing.

*Default = Empty string*

## SetCurrentTask

This property specifies the command used to set current (default) Rational Synergy task.

*Default = "\$OMROOT/etc/Executer.exe" "ccm task -default \$TaskId"*

## ViewTask

This property specifies the command used to view a Rational Synergy task.

*Default = "\$OMROOT/etc/Executer.exe" "ccm task -gui \$TaskId"*



# CORBA

---

The CORBA subject contains metaclasses that contain properties that enable you to use CORBA attributes with Rational Rhapsody Developer for C++.

The CORBA subject is available only in Rational Rhapsody Developer for C++.

## Attribute

Contains properties relating to the use of attributes with CORBA.

### UnionCase

The UnionCase property is used to specify the discriminator value that should be used for an attribute when defining a CORBA union. The values should reflect the type you specified for the property CORBA::Type::Discriminator.

Note that you can also set the value of the property UnionCase to the string "default" for the data type that you want to use as the default data type for the union.

For example, if you are using "short" as the discriminator type, you can set the value of the UnionCase property to 1 for an attribute named length\_short, 2 for an attribute named length\_long, and "default" for an attribute named length\_double.

Using these values, the following code will be generated:

```
union length switch (short) {  
  
case 1 : short length_short; /// attribute length_short  
  
case 2 : long length_long; /// attribute length_long  
  
default : double length_double; /// attribute length_double  
  
};
```

For more information, see "Defining CORBA Unions" in the Rational Rhapsody help.

*Default = Blank*

## C++Mapping\_CORBABasic

The C++Mapping\_CORBABasic metaclass contains properties that affect how CORBA stereotypes are

mapped to C++ code.

## **in**

The "in" property specifies how the CORBA type "in" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType)*

## **inout**

The "inout" property specifies how the CORBA type "inout" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType&)*

## **out**

The "out" property specifies how the CORBA type "out" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType&)*

## **ReturnValue**

The ReturnValue property specifies how a return value that is a CORBA type is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType)*

## **TriggerArgument**

The TriggerArgument property specifies how the type should be passed in when used as an argument for events/triggered operations. By default, classes that are used as an argument are passed by way of a pointer. There are 4 other properties that effect how types are passed into and returned by regular operations: "in", "out", "in/out" and "return."

*(Default = \$MappedType)*

# **C++Mapping\_CORBAEnum**

The C++Mapping\_CORBAEnum metaclass contains properties that affect how CORBA stereotypes are mapped to C++ code.

## **in**

The "in" property specifies how the CORBA type "in" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType)*

## **inout**

The "inout" property specifies how the CORBA type "inout" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType&)*

## **out**

The "out" property specifies how the CORBA type "out" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType&)*

## **ReturnValue**

The ReturnValue property specifies how a return value that is a CORBA type is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType)*

## **TriggerArgument**

The TriggerArgument property specifies how the type should be passed in when used as an argument for events/triggered operations. By default, classes that are used as an argument are passed by way of a pointer. There are 4 other properties that effect how types are passed into and returned by regular operations: "in", "out", "in/out" and "return."

*(Default = \$MappedType)*

# **C++Mapping\_CORBAFixedArray**

The C++Mapping\_CORBAFixedArray metaclass contains properties that affect how CORBA stereotypes are mapped to C++ code.

## **in**

The "in" property specifies how the CORBA type "in" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = const \$MappedType)*

## **inout**

The "inout" property specifies how the CORBA type "inout" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType)*

## **out**

The "out" property specifies how the CORBA type "out" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType)*

## **ReturnValue**

The ReturnValue property specifies how a return value that is a CORBA type is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType slice\*)*

## **TriggerArgument**

The TriggerArgument property specifies how the type should be passed in when used as an argument for events/triggered operations. By default, classes that are used as an argument are passed by way of a pointer. There are 4 other properties that effect how types are passed into and returned by regular operations: "in", "out", "in/out" and "return."

*(Default = \$MappedType slice\*)*

# **C++Mapping\_CORBAFixedSequence**

The C++Mapping\_CORBAFixedSequence metaclass contains properties that affect how CORBA stereotypes are mapped to C++ code.

## **in**

The "in" property specifies how the CORBA type "in" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = const \$MappedType&)*

## **inout**

The "inout" property specifies how the CORBA type "inout" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType&)*

## **out**

The "out" property specifies how the CORBA type "out" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType\*&)*

## **ReturnValue**

The ReturnValue property specifies how a return value that is a CORBA type is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType\*)*

## **TriggerArgument**

The TriggerArgument property specifies how the type should be passed in when used as an argument for events/triggered operations. By default, classes that are used as an argument are passed by way of a pointer. There are 4 other properties that effect how types are passed into and returned by regular operations: "in", "out", "in/out" and "return."

*(Default = \$MappedType\*)*

# **C++Mapping\_CORBAFixedStruct**

The C++Mapping\_CORBAFixedStruct metaclass contains properties that affect how CORBA stereotypes are mapped to C++ code.

## **in**

The "in" property specifies how the CORBA type "in" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = const \$MappedType&)*

## **inout**

The "inout" property specifies how the CORBA type "inout" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType&)*

## **out**

The "out" property specifies how the CORBA type "out" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType&)*

## **ReturnValue**

The ReturnValue property specifies how a return value that is a CORBA type is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType)*

## **TriggerArgument**

The TriggerArgument property specifies how the type should be passed in when used as an argument for events/triggered operations. By default, classes that are used as an argument are passed by way of a pointer. There are 4 other properties that effect how types are passed into and returned by regular operations: "in", "out", "in/out" and "return."

*(Default = \$MappedType)*

# **C++Mapping\_CORBAFixedUnion**

The C++Mapping\_CORBAFixedUnion metaclass contains properties that affect how CORBA stereotypes are mapped to C++ code.

## **in**

The "in" property specifies how the CORBA type "in" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = const \$MappedType&)*

## **inout**

The "inout" property specifies how the CORBA type "inout" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType&)*

## **out**

The "out" property specifies how the CORBA type "out" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType&)*

## **ReturnValue**

The ReturnValue property specifies how a return value that is a CORBA type is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType)*

## **TriggerArgument**

The TriggerArgument property specifies how the type should be passed in when used as an argument for events/triggered operations. By default, classes that are used as an argument are passed by way of a pointer. There are 4 other properties that effect how types are passed into and returned by regular operations: "in", "out", "in/out" and "return."

*(Default = \$MappedType)*

# **C++Mapping\_CORBAInterfaceReference**

The C++Mapping\_CORBAInterfaceReference metaclass contains properties that affect how CORBA stereotypes are mapped to C++ code.

## **in**

The "in" property specifies how the CORBA type "in" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType)*

## **inout**

The "inout" property specifies how the CORBA type "inout" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType&)*

## **out**

The "out" property specifies how the CORBA type "out" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType&)*

## **ReturnValue**

The ReturnValue property specifies how a return value that is a CORBA type is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType)*

## **TriggerArgument**

The TriggerArgument property specifies how the type should be passed in when used as an argument for events/triggered operations. By default, classes that are used as an argument are passed by way of a pointer. There are 4 other properties that effect how types are passed into and returned by regular operations: "in", "out", "in/out" and "return."

*(Default = \$MappedType)*

# **C++Mapping\_CORBAInterfaceVariable**



The C++Mapping\_CORBAInterfaceVariable metaclass contains properties that affect how CORBA stereotypes are mapped to C++ code.

## **in**

The "in" property specifies how the CORBA type "in" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType)*

## **inout**

The "inout" property specifies how the CORBA type "inout" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType&)*

## **out**

The "out" property specifies how the CORBA type "out" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType&)*

## **ReturnValue**

The ReturnValue property specifies how a return value that is a CORBA type is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType)*

## **TriggerArgument**

The TriggerArgument property specifies how the type should be passed in when used as an argument for events/triggered operations. By default, classes that are used as an argument are passed by way of a pointer. There are 4 other properties that effect how types are passed into and returned by regular operations: "in", "out", "in/out" and "return."

*(Default = \$MappedType)*

# **C++Mapping\_CORBASequence**

The C++Mapping\_CORBASequence metaclass contains properties that affect how CORBA stereotypes are mapped to C++ code.

## **in**

The "in" property specifies how the CORBA type "in" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = const \$MappedType&)*

## **inout**

The "inout" property specifies how the CORBA type "inout" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType&)*

## **out**

The "out" property specifies how the CORBA type "out" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType\*&)*

## **ReturnValue**

The ReturnValue property specifies how a return value that is a CORBA type is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType\*)*

## **TriggerArgument**

The TriggerArgument property specifies how the type should be passed in when used as an argument for events/triggered operations. By default, classes that are used as an argument are passed by way of a pointer. There are 4 other properties that effect how types are passed into and returned by regular operations: "in", "out", "in/out" and "return."

*(Default = \$MappedType\*)*

# **C++Mapping\_CORBAVariableArray**

The C++Mapping\_CORBAVariableArray metaclass contains properties that affect how CORBA stereotypes are mapped to C++ code.

## **in**

The "in" property specifies how the CORBA type "in" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = const \$MappedType)*

## **inout**

The "inout" property specifies how the CORBA type "inout" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType)*

## **out**

The "out" property specifies how the CORBA type "out" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType slice\*&)*

## **ReturnValue**

The ReturnValue property specifies how a return value that is a CORBA type is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType slice\*)*

## **TriggerArgument**

The TriggerArgument property specifies how the type should be passed in when used as an argument for events/triggered operations. By default, classes that are used as an argument are passed by way of a pointer. There are 4 other properties that effect how types are passed into and returned by regular operations: "in", "out", "in/out" and "return."

*(Default = \$MappedType slice\*)*

# **C++Mapping\_CORBAVariableStruct**

The C++Mapping\_CORBAVariableStruct metaclass contains properties that affect how CORBA stereotypes are mapped to C++ code.

## **in**

The "in" property specifies how the CORBA type "in" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = const \$MappedType&)*

## **inout**

The "inout" property specifies how the CORBA type "inout" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType&)*

## **out**

The "out" property specifies how the CORBA type "out" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType\*&)*

## **ReturnValue**

The ReturnValue property specifies how a return value that is a CORBA type is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType\*)*

## **TriggerArgument**

The TriggerArgument property specifies how the type should be passed in when used as an argument for events/triggered operations. By default, classes that are used as an argument are passed by way of a pointer. There are 4 other properties that effect how types are passed into and returned by regular operations: "in", "out", "in/out" and "return."

*(Default = \$MappedType\*)*

# **C++Mapping\_CORBAVariableUnion**

The C++Mapping\_CORBAVariableUnion metaclass contains properties that affect how CORBA stereotypes are mapped to C++ code.

## in

The "in" property specifies how the CORBA type "in" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = const \$MappedType&)*

## inout

The "inout" property specifies how the CORBA type "inout" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType&)*

## out

The "out" property specifies how the CORBA type "out" is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType\*&)*

## ReturnValue

The ReturnValue property specifies how a return value that is a CORBA type is mapped to C++ code. The default mapping is defined by the IDL to C++ Language Mapping specification from the Object Management Group (OMG).

*(Default = \$MappedType\*)*

## TriggerArgument

The TriggerArgument property specifies how the type should be passed in when used as an argument for events/triggered operations. By default, classes that are used as an argument are passed by way of a pointer. There are 4 other properties that effect how types are passed into and returned by regular operations: "in", "out", "in/out" and "return."

*(Default = \$MappedType\*)*

## Class

The Class metaclass contains properties that affect CORBA classes.

## C++Implementation

The C++Implementation property allows the user to select the CORBA reference interface or the CORBA variable interface (`_ptr` and `_var` classes), for mapping during code generation.

The type of interface selected determines the metaclass that is used.

The possible values are Reference (default) and Variable.

## DefaultImplementationMethod

The DefaultImplementationMethod property specifies which method is used to implement CORBA interfaces.

The possible values are as follows:

- Inheritance - Use inheritance as the implementation method.
- TIE - Use TIE as the implementation method.

*(Default = Inheritance )*

## GenerateInterfacesAfterExceptions

Prior to release 8.2, when CORBA .idl files were generated, there were situations where exceptions appeared in the generated file before interfaces that referred to the exceptions, resulting in compilation errors. This situation was corrected in 8.2.

To preserve the previous code generation behavior for pre-8.2 models, the property `CORBA::Class::GenerateInterfacesAfterExceptions` was added to the backward compatibility settings for C++ with a value of False.

## IDLSequence

The IDLSequence property determines the name of the typedef used in the implementation of to-many relations.

*(Default = \$interfaceSeq )*

## InheritanceRealizes

The InheritanceRealizes property overrides the default implementation method for CORBA interfaces, as specified by the DefaultImplementationMethod property, for a particular class.

To implement a CORBA interface by using inheritance, if the default implementation method is set to TIE

, set the `InheritanceRealizes` property for the realizing class to the names of the CORBA interfaces that it should realize.

*(Default = empty string)*

## InstanceNameInConstructor

The `InstanceNameInConstructor` property specifies whether to add a string parameter representing the instance name to all class constructors.

*(Default = Cleared)*

## TIERealizes

The `TIERealizes` property overrides the default implementation method for CORBA interfaces, as specified by the `DefaultImplementationMethod` property, for a particular class.

To implement a CORBA interface by using TIE, if the default implementation method is set to `Inheritance`, set the `TIERealizes` property for the realizing class to the names of the CORBA interfaces that it should realize.

*(Default = empty string)*

# Configuration

The Configuration metaclass contains properties that control the CORBA configuration.

## CORBAEnable

The `CORBAEnable` property specifies whether to generate code for a CORBA client, CORBA server, or neither.

The possible values are as follows:

- `No` - Generate code for neither a client nor a server.
- `CORBAClient` - Generate code for a CORBA client.
- `CORBAServer` - Generate code for a CORBA server.

*(Default = No)*

## ExposeCorbaInterfaces

The `ExposeCorbaInterfaces` property generates server IDL code for the specified CORBA interfaces.

*(Default = empty string)*

## **IDLExtension**

The IDLExtension property specifies the extension for IDL files.

*(Default = .idl)*

## **IncludeIDL**

The IncludeIDL property is a string that lists the IDL files to include at the component level. Separate multiple files with commas.

*(Default = empty string)*

## **ORB**

The ORB property specifies the ORB with which you are working. The value is either TAO or UserDefinedORB - Use this setting to add a new ORB.

*(Default = TAO)*

## **StartFrameworkInMainThread**

The StartFrameworkInMainThread property is a Boolean value that specifies whether the framework should control the main thread. When you set this property to True at the configuration level, OXF::start(FALSE) is called and the OXF takes over the main thread.

You would use this property in the case where you want Product; to take over the main thread instead of CORBA.

*(Default = Cleared)*

## **UseCorbaInterfaces**

The UseCorbaInterfaces property generates client IDL code for the specified CORBA interfaces.

*(Default = empty string)*

## **UsePre82InterfaceOrder**

Prior to release 8.2, there were cases of generated .idl files where CORBA interfaces and exceptions were used before they were declared. This issue was corrected in release 8.2. In order to preserve the previous code generation behavior for pre-8.2 models, the property CORBA::Configuration::UsePre82InterfaceOrder was added to the backward compatibility settings for



C++, with a value of True.

## Operation

The Operation metaclass contains properties that affect CORBA operations.

### **C++DefaultThrow**

*(Default = CORBA::SystemException)*

### **IsOneWay**

The IsOneWay property specifies whether to create a OneWay CORBA operation.

*(Default = Cleared)*

### **ThrowExceptions**

The ThrowExceptions property specifies the exceptions that an operation can throw. Separate multiple exceptions with commas.

*(Default = empty string)*

## Package

The Package metaclass contains a property that controls interface packages.

### **DeclareInterfacesInModule**

The DeclareInterfacesInModule property specifies whether forward declaration of CORBA interfaces is enabled.

*(Default = Cleared)*

## TAO

The TAO metaclass contains properties that affect CORBA TAO.

## AddCORBAEnvParam

The AddCORBAEnvParam property specifies whether to add a CORBA environment parameter (of type CORBA\_env) as the last argument to CORBA operations.

*(Default = Cleared)*

## ClientMainLineTemplate

The ClientMainLineTemplate property adds code in the main function of a CORBA client.

*(Default = empty MultiLine)*

## CORBAIncludePath

The CORBAIncludePath property specifies the location of CORBA include files.

This property is visible both in the Properties window and on the Middleware tab of the Configuration window in the browser.

The default value is as follows: \$(ACE\_ROOT)\TAO\ \$(ACE\_ROOT)\ \$(ACE\_ROOT)\TAO\orbsvcs

## CORBALibs

The CORBALibs property specifies the locations of the CORBA libraries.

This property is visible both in the Properties window and on the Middleware tab of the Configuration window in the browser.

The default value is as follows: \$(ACE\_ROOT)\TAO\tao\PortableServer\TAO\_PortableServerd.lib  
\$(ACE\_ROOT)\TAO\tao\Valuetype\TAO\_Valuetyped.lib \$(ACE\_ROOT)\TAO\tao\TAOd.lib  
\$(ACE\_ROOT)\ace\aced.lib

## CPP\_CompileSwitches

The CPP\_CompileSwitches property is a string that specifies additional compiler switches.

The default value is as follows: /GR /D "ACE\_AS\_STATIC\_LIBS" /D "TAO\_AS\_STATIC\_LIBS"

## CPP\_LinkSwitches

The CPP\_LinkSwitches property is a string that specifies additional link switches, needed when your component is linked with the libraries for this ORB.

*(Default = empty string)*

## **CPP\_StandardInclude**

The CPP\_StandardInclude property is a string that specifies additional header files to be included in the generated sources, needed when your component is compiled with the include files for this ORB.

*(Default = tao/CORBA.h;tao/PortableServer/POA.h)*

## **DefTIEString**

The DefTIEString property specifies a template for the string generated into every IDL file that contains a CORBA interface, if the DefaultImplementationMethod property is set to TIE .

The default value is blank.

## **DestroyInitialInstance**

The DestroyInitialInstance property specifies a template that destroys the object created during the initial instance.

*(Default = orb-destroy(); )*

## **EnvParamDefaultVal**

The EnvParamDefaultVal property is a string that specifies an environment parameter as the last argument to operations that implement CORBA interfaces.

The default value is CORBA::default\_environment.

## **EnvParamName**

The EnvParamName property is a string that specifies the name of the environment parameter added by the EnvParamDefaultVal property.

*(Default = IT\_env)*

## **EnvParamType**

The EnvParamType property specifies the type of the environment parameter added by the EnvParamDefaultVal property.

*(Default = CORBA::Environment&)*

## IDLCompileCommand

The IDLCompileCommand property specifies the compile command for a given IDL compiler.

This property is visible both in the Properties window and on the Middleware tab of the Configuration window in the browser.

The default is as follows:

```
@echo IDL Compiling $OMFileSpecPath $(ACE_ROOT)\bin\tao_idl -o $OMFileSpecDir
```

## IDLCompileSwitches

The IDLCompileSwitches property specifies the switches for the IDL compiler.

There are two ways to glue a CORBA implementation class to the ORB - BOA and TIE. The Rational Rhapsody default is BOA. The TAO -B flag compiles the IDL so BOA objects are created.

This property is visible both in the Properties window and on the Middleware tab of the Configuration window in the browser.

*(Default = -B)*

## ImplementationExtension

The ImplementationExtension property specifies the extension for implementation files.

*(Default = .cpp)*

## InitialInstance

The InitialInstance property specifies any additional initial instance routines that are required by the ORB. This code template is generated for each instance of a specific class (implementing one or more CORBA interfaces). Note: the template is inserted when the class was selected in the Explicit Initial Instances in the Configuration window, but not for user-created instances.

```
For example: /***** Default TAO InitialInstance *****/ try {  
  
// If you open this commented code, the createRefFile should be defined in $instance  
  
// PortableServer::ServantBase_var servant = $instance;  
  
// $instance-createRefFile(orb);  
  
}  
  
catch(const CORBA::Exception e) {
```

```

omcerr "Got CORBA exception in instantiation" omendl;

omcerr e omendl;

return 1;

}

```

Note that \$ClassName expands to the name of the class being initialized and \$instance expands to its instance name.

*(Default = above example)*

## InitializeORB

The InitializeORB property specifies the ORB initialization routines. In most cases, this is the first executable command in the main function of the CORBA server.

You can place any ORB initialization code in this property. For example: `/* Default Initializing ORB and getting POA Manager */ // any ORB initialization PortableServer::POAManager_var poa_manager = NULL; CORBA::ORB_var orb = NULL; PortableServer::POA_var rootPOA = NULL; CORBA::PolicyList policies = NULL; try { // Initialize the ORB. orb = CORBA::ORB_init(argc, argv); // get a reference to the root POA CORBA::Object_var obj = orb-resolve_initial_references(\"RootPOA\"); rootPOA = PortableServer::POA::_narrow(obj); policies.length(1); policies[(CORBA::ULong)0] = rootPOA-create_lifespan_policy( PortableServer::PERSISTENT); // get the POA Manager poa_manager = rootPOA-the_POAManager(); } catch(const CORBA::Exception e) { cerr \"Got CORBA exception in initialization\" endl; cerr e endl; return 1; }`

"

*(Default = /\*\*\*\*\* Default TAO Initilizing ORB and getting POA Manager \*\*\*\*\*/*

```

CORBA::ORB_var orb = NULL;

CORBA::Object_var poaObj = NULL;

PortableServer::POA_var rootPoa = NULL;

PortableServer::POAManager_var manager = NULL;

try {

// Initialize the ORB.

orb = CORBA::ORB_init(argc, argv);

poaObj = orb - resolve_initial_references("RootPOA");

rootPoa = PortableServer::POA::_narrow(poaObj.in());

```

```

manager = rootPoa - the_POAManager();

}

catch(const CORBA::Exception e) {

omcerr "Got CORBA exception in initialization" omendl;

omcerr e omendl;

return 1;

}

)

```

The default value for VisiBrokerRT is “.cc ”; the default value for all other ORBs is “.cpp” .

## NeededObjForClient

The NeededObjForClient property is an enumerated type that specifies the file needed to create an object.

*(Default = Stub)*

## NeededObjForClientServer

The NeededObjForClientServer property is an enumerated type that specifies the file needed to create a client server. The possible values are as follows:

- Stub
- Skeleton
- Both

*(Default = Both)*

## NeededObjForServer

The NeededObjForServer property is an enumerated type that specifies the file needed to create a server. The possible values are as follows:

- Stub
- Skeleton
- Both

*(Default = Both)*

## ServerMainLineTemplate

The ServerMainLineTemplate property is a MultiLine type that defines how Tao interacts with Rational Rhapsody. This property is used in the “main” part of the generated application.

The default value for TAO is as follows:

```
/****** Default TAO Server Mainline *****/

try {

// Activate POA Manager

manager-activate();

// Wait for incoming requests

orb-run();

}

catch(const CORBA::Exception e) {

omcerr "Got CORBA exception in orb run" omendl;

omcerr e omendl;

return 1;

}
```

## Skeleton

The Skeleton property specifies the inheritance format used by a given ORB vendor.

*(Default = POA\_\$(interface))*

The CORBA interface name replaces the \$(interface) variable in the generated code.

## SkeletonImplementationName

The SkeletonImplementationName property is a string that defines the naming behavior for skeleton implementation files.

*(Default = \$(interfaceS))*

## **SkeletonSpecificationName**

The SkeletonSpecificationName property is a string that defines the naming behavior for skeleton specification files.

*(Default = \$interfaceS)*

## **SpecificationExtension**

The SpecificationExtension property is a string that specifies the extension for specification files.

*(Default = .h)*

## **StubImplementationName**

The StubImplementationName property is a string that defines the naming behavior for stub implementation files.

*(Default = \$interfaceC )*

## **StubSpecificationName**

The StubSpecificationName property is a string that defines the naming behavior for stub specification files.

*(Default = \$interfaceC)*

# **Type**

The Type metaclass contains properties that enable you to change the OMG default mappings.

## **C++Implementation**

The C++Implementation property allows the user to select the CORBA fixed construct or the CORBA variable construct, for mapping during code generation.

The type of construct selected determines the metaclass that is used.

The possible values are Fixed and Variable (default).

## **CORBAStereotype**



The CORBAStereotype property specifies the CORBA stereotype that is applied to a CORBA type.

The possible values are as follows:

CORBABasic

CORBAEnum

CORBAFixedArray

CORBAFixedSequence

CORBAFixedStruct

CORBAFixedUnion

CORBAInterfaceReference

CORBAInterfaceVariable

CORBASequence

CORBAVariableArray

CORBAVariableStruct

CORBAVariableUnion

*(Default = CORBABasic)*

## **CPP\_in**

The CPP\_in property is a string that overrides the OMG default IDL to C++ language mapping for a CORBA in parameter.

*(Default = empty string)*

## **CPP\_inout**

The CPP\_inout property is a string that overrides the OMG default IDL to C++ language mapping for a CORBA inout parameter.

*(Default = empty string)*

## **CPP\_out**

The CPP\_out property is a string that overrides the OMG default IDL to C++ language mapping for a

CORBA out parameter.

*(Default = empty string)*

## CPP\_return\_value

The CPP\_return\_value property is a string that overrides the OMG default IDL to C++ language mapping for a CORBA return value type.

*(Default = empty string)*

## Discriminator

The Discriminator property is used to specify the type of the discriminator you will be using when defining a CORBA union.

This property is used in conjunction with the property CORBA::Attribute::UnionCase.

For more information, see "Defining CORBA Unions" in the Rational Rhapsody help.

*Default = Blank*

## IDLSequence

The IDLSequence property determines the name of the typedef used in the implementation of to-many relations.

*(Default = \$typeSeq)*

## StringMaximumSize

When you define a CORBA bounded string or wstring, the StringMaximumSize property is used to specify the maximum string length that you want to allow. The value can be any positive integer. If you leave the value of the property blank, an unbounded string definition is generated.

For example, if you set the value of this property to 22 for a bounded string, the generated code will resemble the following code:

```
typedef string<22> lastName;
```

For more information, see "Defining CORBA bounded strings" in the Rational Rhapsody help.

*Default = Blank*

## UserDefinedORB

The UserDefinedORB metaclass contains properties that affect CORBA TAO.

### AddCORBAEnvParam

The AddCORBAEnvParam property specifies whether to add a CORBA environment parameter (of type CORBA\_env ) as the last argument to CORBA operations.

*(Default = Checked)*

### ClientMainLineTemplate

The ClientMainLineTemplate property adds code in the main function of a CORBA client.

*(Default = empty MultiLine)*

### CORBAIncludePath

The CORBAIncludePath property specifies the location of CORBA include files.

This property is visible both in the Properties window and on the Middleware tab of the Configuration window in the browser.

The default value is as follows: \$(IT\_CONFIG\_PATH)\..\include

### CORBALibs

The CORBALibs property specifies the locations of the CORBA libraries.

This property is visible both in the Properties window and on the Middleware tab of the Configuration window in the browser.

The default value is as follows: \$(IT\_CONFIG\_PATH)\..\lib\ITMi.lib

### CPP\_CompileSwitches

The CPP\_CompileSwitches property is a string that specifies additional compiler switches.

### CPP\_LinkSwitches

The `CPP_LinkSwitches` property is a string that specifies additional link switches, needed when your component is linked with the libraries for this ORB.

*(Default = empty string)*

## CPP\_StandardInclude

The `CPP_StandardInclude` property is a string that specifies additional header files to be included in the generated sources, needed when your component is compiled with the include files for this ORB.

*(Default = CORBA.h)*

## DefTIEString

The `DefTIEString` property specifies a template for the string generated into every IDL file that contains a CORBA interface, if the `DefaultImplementationMethod` property is set to `TIE`.

The default value is as follows: `DEF_TIE_$interface($class)`

## DestroyInitialInstance

The `DestroyInitialInstance` property specifies a template that destroys the object created during the initial instance.

The default value is `orb-destroy();`

## EnvParamDefaultVal

The `EnvParamDefaultVal` property is a string that specifies an environment parameter as the last argument to operations that implement CORBA interfaces.

The default value is `CORBA::default_environment`.

## EnvParamName

The `EnvParamName` property is a string that specifies the name of the environment parameter added by the `EnvParamDefaultVal` property.

*(Default = IT\_env)*

## EnvParamType

The `EnvParamType` property specifies the type of the environment parameter added by the `EnvParamDefaultVal` property.

*(Default = CORBA::Environment &)*

## **IDLCompileCommand**

The IDLCompileCommand property specifies the compile command for a given IDL compiler.

This property is visible both in the Properties window and on the Middleware tab of the Configuration window in the browser.

*(Default = idl)*

## **IDLCompileSwitches**

The IDLCompileSwitches property specifies the switches for the IDL compiler.

There are two ways to glue a CORBA implementation class to the ORB - BOA and TIE. The Rational Rhapsody default is BOA. The TAO -B flag compiles the IDL so BOA objects are created.

This property is visible both in the Properties window and on the Middleware tab of the Configuration window in the browser.

*(Default = -B )*

## **ImplementationExtension**

The ImplementationExtension property specifies the extension for implementation files.

*(Default = .cpp)*

## **InitialInstance**

The InitialInstance property specifies any additional initial instance routines that are required by the ORB. This code template is generated for each instance of a specific class (implementing one or more CORBA interfaces).

For example: */\*\*\*\*\* Default TAO InitialInstance \*\*\*\*\*/*

```
try {  
  
// If you open this commented code, the createRefFile should be defined in $instance  
  
// PortableServer::ServantBase_var servant = $instance;  
  
// $instance-createRefFile(orb);  
  
}
```

```

catch(const CORBA::Exception e) {

omcerr "Got CORBA exception in instantiation" omendl;

omcerr e omendl;

return 1;

}

```

Note that `$ClassName` expands to the name of the class being initialized and `$instance` expands to its instance name.

*(Default = above example)*

## InitializeORB

The InitializeORB property specifies the ORB initialization routines. In most cases, this is the first executable command in the main function of the CORBA server.

You can place any ORB initialization code in this property. For example: `/* Default Initializing ORB and getting POA Manager */` // any ORB initialization `PortableServer::POAManager_var poa_manager = NULL; CORBA::ORB_var orb = NULL; PortableServer::POA_var rootPOA = NULL; CORBA::PolicyList policies = NULL; try { // Initialize the ORB. orb = CORBA::ORB_init(argc, argv); // get a reference to the root POA CORBA::Object_var obj = orb-resolve_initial_references(\"RootPOA\"); rootPOA = PortableServer::POA::_narrow(obj); policies.length(1); policies[(CORBA::ULong)0] = rootPOA-create_lifespan_policy( PortableServer::PERSISTENT); // get the POA Manager poa_manager = rootPOA-the_POAManager(); } catch(const CORBA::Exception e) { cerr \"Got CORBA exception in initialization\" endl; cerr e endl; return 1; }`

"

*(Default = `/****** Default TAO Initilizing ORB and getting POA Manager *****/`)*

```

CORBA::ORB_var orb = NULL;

CORBA::Object_var poaObj = NULL;

PortableServer::POA_var rootPoa = NULL;

PortableServer::POAManager_var manager = NULL;

try {

// Initialize the ORB.

orb = CORBA::ORB_init(argc, argv);

poaObj = orb - resolve_initial_references("RootPOA");

```

```

rootPoa = PortableServer::POA::_narrow(poaObj.in());

manager = rootPoa - the_POAManager();

}

catch(const CORBA::Exception e) {

omcerr "Got CORBA exception in initialization" omendl;

omcerr e omendl;

return 1;

}

)

```

The default value for VisiBrokerRT is “.cc”; the default value for all other ORBs is “.cpp”.

## NeededObjForClient

The NeededObjForClient property is an enumerated type that specifies the file needed to create an object.

*(Default = Stub)*

## NeededObjForClientServer

The NeededObjForClientServer property is an enumerated type that specifies the file needed to create a client server. The possible values are as follows:

- Stub
- Skeleton
- Both

*(Default = Skeleton)*

## NeededObjForServer

The NeededObjForServer property is an enumerated type that specifies the file needed to create a server. The possible values are as follows:

- Stub
- Skeleton
- Both

*(Default = Skeleton)*

## ServerMainLineTemplate

The ServerMainLineTemplate property is a MultiLine type that defines how Tao interacts with Rational Rhapsody. This property is used in the “main” part of the generated application.

The default value for TAO is as follows:

```
/****** Default TAO Server Mainline *****/

try {

// Activate POA Manager

manager-activate();

// Wait for incoming requests

orb-run();

}

catch(const CORBA::Exception e) {

omcerr "Got CORBA exception in orb run" omendl;

omcerr e omendl;

return 1;

}
```

## Skeleton

The Skeleton property specifies the inheritance format used by a given ORB vendor.

*(Default = \$interfaceBOAImpl)*

The CORBA interface name replaces the "\$interface" variable in the generated code.

## SkeletonImplementationName

The SkeletonImplementationName property is a string that defines the naming behavior for skeleton implementation files.

*(Default = \$interfaceS)*



## **SkeletonSpecificationName**

The SkeletonSpecificationName property is a string that defines the naming behavior for skeleton specification files.

*(Default = \$interface)*

## **SpecificationExtension**

The SpecificationExtension property is a string that specifies the extension for specification files.

*(Default = .hh)*

## **StubImplementationName**

The StubImplementationName property is a string that defines the naming behavior for stub implementation files.

*(Default = \$interfaceC)*

## **StubSpecificationName**

The StubSpecificationName property is a string that defines the naming behavior for stub specification files.

*(Default = \$interface)*

# CPP\_CG

---

The CPP\_CG subject contains metaclasses that contain properties that specify the operating system environments, in addition to general metaclasses for code generation.

## Activity

Contains properties that affect code generation for activities.

### MeaningfulGuards

Beginning in version 7.6.1, the text specified for guards in token-based activity diagrams is included in the generated code. Before this change, guards could contain any text, and this could result in compilation problems if the new behavior was used for pre-7.6.1 models.

To prevent any such problems, the MeaningfulGuards property was added to the backward compatibility settings for C++ projects with a value of False.

## Argument

The Argument metaclass contains properties that control how arguments are generated in code.

### ClassWide

The ClassWide property determines whether a class-wide modifier is generated for the argument.

*Default = False*

### DeclarationModifier

The DeclarationModifier property is used to allow Rational Rhapsody to reverse engineer non-standard keywords that appear in argument declarations. Keywords that appear between the argument type and the argument name are stored as the value of this property, and the property is then used during code generation to re-create the original code.

Since this is a code generation property, it can also be used to add non-standard keywords to code even when reverse engineering is not used.

This property is used in conjunction with the PreDeclarationModifier and PostDeclarationModifier properties.

*Default = Blank*

## DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- `$Name` - the element name
- `$FullName` - the full path of the element (P1::P2::C.a)
- `$Description` - the text provided on the Description tab of the Features window for the element
- `$Requirements` - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the `$Requirements` keyword in the DescriptionTemplate property to get such comments generated for specific model elements.
- `$Login` - the login name of the user who generated the file
- `$CodeGeneratedDate` - the date on which the code was generated
- `$CodeGeneratedTime` - the time at which the code was generated
- `$RhapsodyVersion` - the version of Rhapsody that was used to generate the file
- For operations: `$Signature` - the operation signature, `$Arguments` - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: `$Type` - the type of the attribute
- For arguments: `$Type` - the argument type, `$Direction` - the argument direction (In, Out, or InOut)
- For events: `$Arguments` - the text provided on the Description tab of the Features window for the event arguments
- For association ends: `$Target` - the other end of the association
- For requirements: `$ID` - the ID specified for the requirement, `$Specification` - the specification text for the requirement
- The values of tags can be included by using the syntax `$tag_name`. For example, if you use a tag called `author` for your classes, you can use the keyword `$author` in the value of DescriptionTemplate in order to have the value of the `author` tag included in the element description.
- The values of properties can be included by using the syntax `$property_name`. For example, you can use the keyword `$Animate` in the value of DescriptionTemplate in order to have the value of the `Animate` property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as \$Name)
- Tag keywords
- Property keywords

If the value of a keyword is a MultiLine value, then each new line (except the first one) starts with the value of the `<lang>_CG::Configuration::DescriptionBeginLine` property, and each line ends with the value of the `<lang>_CG::Configuration::DescriptionEndLine` property.

*Default = Blank*

## FullQualifiedTypeName

To specify that the full qualified name should be generated when elements are used in contexts such as return types, set the value of the property `FullQualifiedTypeName` to `True`.

The value of this property can be set separately for attributes, relations, arguments, and operation return types (use `CPP_CG::Operation::FullQualifiedTypeName` for return types).

The property can be set for specific contexts such as specific attributes or operations, or it can be set at the package or project level, for example, using the full qualified name for return types in a specific package.

*Default = False*

## IsRegister

The `IsRegister` property can be used to specify that the keyword "register" should be generated in the code for a given argument.

*Default = Cleared*

## IsVolatile

The `IsVolatile` property allows you to specify that a specific operation argument should be declared as volatile.

*Default = Cleared*

## PostDeclarationModifier

The `PostDeclarationModifier` property is used to allow Rational Rhapsody to reverse engineer non-standard keywords that appear in argument declarations. Keywords that appear after the argument name are stored as the value of this property, and the property is then used during code generation to re-create the original code.

Since this is a code generation property, it can also be used to add non-standard keywords to code even when reverse engineering is not used.

This property is used in conjunction with the `PreDeclarationModifier` and `DeclarationModifier` properties.

*Default = Blank*

## **PreDeclarationModifier**

The `PreDeclarationModifier` property is used to allow Rational Rhapsody to reverse engineer non-standard keywords that appear in argument declarations. Keywords that appear before the argument type are stored as the value of this property, and the property is then used during code generation to re-create the original code.

Since this is a code generation property, it can also be used to add non-standard keywords to code even when reverse engineering is not used.

This property is used in conjunction with the `DeclarationModifier` and `PostDeclarationModifier` properties.

*Default = Blank*

## **PrintName**

When an operation argument is not accessed in the operation body, some compilers issue a warning. The `PrintName` property allows you to avoid such warnings by having the generated code include only the argument type but not the argument name.

If the property is set to `True`, the argument name is included in the generated code. If the property is set to `False`, only the argument type is included in the generated code.

*Default = True*

## **Attribute**

The `Attribute` metaclass contains properties that control attributes of code generation, such as whether to generate accessor operations.

### **Accessor**

The `Accessor` property is ignored by Rational Rhapsody.

### **AccessorGenerate**

The `AccessorGenerate` property specifies whether to generate accessor operations for attributes. The

possible values are:

- Checked - A `get()` method is generated for the attribute.
- Cleared - A `get()` method is not generated for the attribute.

Setting this property to Cleared is one way to optimize your code for size.

*Default = Checked*

## AccessorVisibility

The `AccessorVisibility` property specifies the access level of the generated accessor for attributes. This property defines the access level of an accessor operation regardless of the visibility of the attribute. The possible values are as follows:

- `fromAttribute` - Use the attribute access level for the accessor.
- `public` - Set the accessor access level to public.
- `private` - Set the accessor access level to private.
- `protected` - Set the accessor access level to protected.

*Default = fromAttribute*

## AttributeInitializationFile

In situations where the compiler will tolerate attribute initialization in either the specification or implementation file, the `AttributeInitializationFile` property can be used to specify where the initialization code should be generated.

The possible values are:

- `Default` - the decision where to generate the initialization code depends on characteristics of the attribute
- `Specification` - the initialization code will be generated in the specification file
- `Implementation` - the initialization code will be generated in the implementation file

*Default = Default*

## BitField

Allows you to define a bit field for an attribute. To define a bit field, open the Features window for the relevant attribute and enter the number you want to use for the bit field as the value of the `BitField` property.

For example, if you enter 2 as the value of `BitField` for an attribute named `attribute_1` of type `int`, the resulting code is:

```
int attribute_1 : 2;
```

## ConstantVariableAsDefine

This property is a Boolean value that determines whether the variable, defined as constant in file or package, is generated by using a `#define` macro. Otherwise, it is generated by using the `const` qualifier.

*Default = Cleared*

## DeclarationModifier

The DeclarationModifier property is used to allow Rational Rhapsody to reverse engineer non-standard keywords that appear in attribute declarations. Keywords that appear between the attribute type and the attribute name are stored as the value of this property, and the property is then used during code generation to re-create the original code.

Since this is a code generation property, it can also be used to add non-standard keywords to code even when reverse engineering is not used.

This property is used in conjunction with the PreDeclarationModifier and PostDeclarationModifier properties.

*Default = Blank*

## DeclarationPosition

The DeclarationPosition property controls the declaration order of attributes. The possible values are as follows:

- Default - Similar to the AfterClassRecord setting, with the following difference:
- For static attributes defined in a class with the `CPP_CG::Attribute::Visibility` property set to Public, these attributes are generated after types whose `CPP_CG::Type::Visibility` property is set to Public.
- You should not use this setting for new models. See the Rational Rhapsody Developer for Ada documentation for more information.
- BeforeClassRecord - Generate the attribute immediately before the class record.
- AfterClassRecord - Generate the attribute immediately after the class record.
- StartOfDeclaration - Generate the attribute immediately after the start of the section (private or public part of the specification, or package body).
- EndOfDeclaration - Generate the attribute immediately before the end of the section (private or public part of the specification, or package body).

*Default = Default*

## DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- `$Name` - the element name
- `$FullName` - the full path of the element (P1::P2::C.a)
- `$Description` - the text provided on the Description tab of the Features window for the element
- `$Requirements` - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the `$Requirements` keyword in the `DescriptionTemplate` property to get such comments generated for specific model elements.
- `$Login` - the login name of the user who generated the file
- `$CodeGeneratedDate` - the date on which the code was generated
- `$CodeGeneratedTime` - the time at which the code was generated
- `$RhapsodyVersion` - the version of Rhapsody that was used to generate the file
- For operations: `$Signature` - the operation signature, `$Arguments` - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: `$Type` - the type of the attribute
- For arguments: `$Type` - the argument type, `$Direction` - the argument direction (In, Out, or InOut)
- For events: `$Arguments` - the text provided on the Description tab of the Features window for the event arguments
- For association ends: `$Target` - the other end of the association
- For requirements: `$ID` - the ID specified for the requirement, `$Specification` - the specification text for the requirement
- The values of tags can be included by using the syntax `$tag_name`. For example, if you use a tag called `author` for your classes, you can use the keyword `$author` in the value of `DescriptionTemplate` in order to have the value of the `author` tag included in the element description.
- The values of properties can be included by using the syntax `$property_name`. For example, you can use the keyword `$Animate` in the value of `DescriptionTemplate` in order to have the value of the `Animate` property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as `$Name`)
- Tag keywords
- Property keywords



If the value of a keyword is a MultiLine value, then each new line (except the first one) starts with the value of the <lang>\_CG::Configuration::DescriptionBeginLine property, and each line ends with the value of the <lang>\_CG::Configuration::DescriptionEndLine property.

*Default = Blank*

## **EnableInitializationStyleForStaticAttributes**

The style used for initializing C++ attributes is determined by the CPP\_CG::Attribute::InitializationStyle property, which can take the values ByInitializer (default value) and ByAssignment. The EnableInitializationStyleForStaticAttributes property allows you to specify whether or not Rational Rhapsody should apply this property to static attributes as well.

If the value of this property is set to True, the code for initializing static attributes is based on the value of the InitializationStyle property.

If the value of this property is set to False, then the value of the InitializationStyle property have no effect on the initialization of static attributes. Rather, the initialization of static attributes is always by assignment.

*Default = True*

## **FullQualifiedTypeName**

To specify that the the full qualified name should be generated when elements are used in contexts such as return types, set the value of the property FullQualifiedTypeName to True.

The value of this property can be set separately for attributes, relations, arguments, and operation return types (use CPP\_CG::Operation::FullQualifiedTypeName for return types).

The property can be set for specific contexts such as specific attributes or operations, or it can be set at the package or project level, for example, using the full qualified name for return types in a specific package.

*Default = False*

## **GenerateAttributeImplementationForInterface**

Prior to release 8.0, if you added an attribute to an interface, the C++ code that was generated for the interface contained a data member for the attribute as well as a setter and getter for the attribute. Beginning in release 8.0, the C++ code that is generated for the interface does not contain a data member for the attribute, but only abstract setters and getters for modifying the value of the attribute. To preserve the previous code generation behavior for pre-8.0 models, the CPP\_CG::Attribute::GenerateAttributeImplementationForInterface property was added to the backward compatibility settings for C++ with a value of True.

## **GenerateVariableHelpers**

By default, Rational Rhapsody generates getter and setter methods for class attributes, but not for global variables. If you want Rational Rhapsody to generate getter and setter methods for global variables, set the value of the `GenerateVariableHelpers` property to `True`.

*Default = Cleared*

## ImplementationEpilog

The `ImplementationEpilog` property enables you to add any code that you want to be added as verbatim text (to be ignored by Rational Rhapsody) to the end of the definition of a model element. For example, you could wrap a section of code with an `#ifdef-#endif` pair, add compiler-specific keywords, or add a `#pragma` statement.

For example, to specify that an operation is available only when the code is compiled with `_DEBUG`, set the following properties for the operation:

- Set `SpecificationProlog` to `#ifdef _DEBUG cr.`
- Set `SpecificationEpilog` to `#endif.`
- Set `ImplementationProlog` to `#ifdef _DEBUG cr.`
- Set `ImplementationEpilog` to `#endif.`

The following table lists whether leading and trailing linefeeds are generated.

Metaclass	Leading Linefeed	Added?	Class	Yes	Package	No
-----------	------------------	--------	-------	-----	---------	----

*Default = Empty MultiLine*

## ImplementationName

The `ImplementationName` property gives an operation one model name and generate it with another name. It is introduced as a workaround that generates const and non-const operations with the same name. For example:

- Create a class A.
- Add a non-const operation `f()`.
- Add a const operation `f_const()`.
- Set the `CPP_CG::Operation::ImplementationName` property for `f_const()` to “f.”
- Generate the code.

The resulting code is as follows: `class A { ... void f(); /* the non const f */ ... void f() const; /* actually f_const() */ ... };` The creation of two operations with the same signature, differing only in whether it is a const, is a common practice in C++, especially for STL users.

*Default = Empty string*

## ImplementationProlog

The ImplementationProlog property enables you to add any code that you want to be added as verbatim text (to be ignored by the software) to the beginning of the definition of a model element.

For example, you could wrap a section of code with an `#ifdef-#endif` pair, add compiler-specific keywords, or add a `#pragma` statement. For example, to specify that an operation is available only when the code is compiled with `_DEBUG`, set the following properties for the operation:

- Set SpecificationProlog to `#ifdef _DEBUG` cr.
- Set SpecificationEpilog to `#endif`.
- Set ImplementationProlog to `#ifdef _DEBUG` cr.
- Set ImplementationEpilog to `#endif`.

The following table lists whether leading and trailing linefeeds are generated.

Metaclass Trailing Linefeed Added? Class No Package Yes

*Default = Empty MultiLine*

## InitializationStyle

The InitializationStyle property specifies the initialization style used for attributes. When you specify an initial value for an attribute, Rational Rhapsody initializes the attribute based on the value of this property. In Rational Rhapsody Developer for C++, the possible values are as follows:

- ByInitializer - Initialize the attribute in the initializer (`a(y)`). This is the default value. If the initialization style is ByInitializer, the attribute initialization should be done after the user initializer, in the same order as the order of attributes in the code.
- ByAssignment - Initialize the attribute in the constructor body (`a = y`).

*Default = ByInitializer*

## Inline

The Inline property specifies how inline operations are generated. Which operations are affected by the Inline property depends on the metaclass:

- Attribute - Applies only to operations that handle attributes (such as accessors and mutators)
- Operation - Applies to all operations
- Relation - Applies only to operations that handle relations

Inlining in Rational Rhapsody Developer for C++ The possible values for the Inline property are as follows:

- none - The operation is not generated inline.
- in\_header - The operation is generated inline in the specification file.
- in\_source - The operation is generated inline in the implementation file.
- in\_declaration - A class operation is generated inline in the class declaration. A global function is generated inline in the package specification file.

Inlining an operation in the header might cause problems if the function body refers to other classes. For example, if the inlined code refers to another class (by way of a pointer such as `itsRelatedClass`), inlined code that is generated in a header might not compile.

The implementation file for the class would have an `#include` for `RelatedClass`, but the specification file would not.

The workaround is to create a Usage dependency of the class with the inlined function on the related class. This forces an `#include` of the related class to be generated in the header of the dependent class with the inlined function.

*Default = none*

## **IsAliased**

The `IsAliased` property is a Boolean value that specifies whether attributes are aliased.

*Default = False*

## **IsMutable**

The `IsMutable` property is a Boolean value that allows you to specify that an attribute is a mutable attribute.

*Default = Cleared*

## **IsVolatile**

The `IsVolatile` property allows you to specify that an attribute should be declared as volatile.

*Default = Cleared*

## **Kind**

The `Kind` property specifies the kind of operation that should be generated for an element. The kind of operations that can be generated is language-dependent (for example, `virtual` and `abstract` exist only in C++ and Java).

In Java, `Kind` can be defined only for attributes and operations, but not for relations. This property affects class operations, in addition to accessors and mutators for relations and attributes. The possible values are as follows:

- `common` - Class operations and accessor/mutator are non-virtual.
- `virtual` - Class operations and accessor/mutator are virtual. This type is valid for C++ and Java only.
- `abstract` - Class operations and accessor/mutator are pure virtual. This type is valid for C++ and Java only.

*Default = common*

## MarkPrologEpilogInAnnotations

The MarkPrologEpilogInAnnotations property specifies whether to generate ignore annotations for the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties so they are ignored during roundtrip.

When you insert code element declarations (variables, types, functions, and so on) in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, after a full roundtrip those elements are added to the model and are duplicated on the next code generation. When you use the MarkPrologEpilogInAnnotations property, you can have Rational Rhapsody automatically ignore the information specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties instead of adding the ignore annotations manually.

The possible values for the MarkPrologEpilogInAnnotations property are:

- None - Rational Rhapsody does not generate any annotations.
- Ignore - Rational Rhapsody generates the `///  
[ ignore annotation before the code specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, and generates the ///  
[ annotation after the code specified in those properties.`
- Auto - If the code in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties is one line (it does not contain any newline characters `(\n)`), no annotations are generated (the same behavior has the None setting). If there is more than one line, Rational Rhapsody generates the `///  
[ ignore annotation before the code specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, and generates the ///  
[ annotation after the code specified in those properties (the same behavior as the Ignore setting).`

During roundtrip, any ignore annotations in the comments of the element are not included in its description. Because the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties are generated between the element annotation and its declaration, you cannot rename those elements on roundtrip. If you change the name of an element, it is removed from the model and added with the new name. Some model information (for example, property settings) might be lost.

*Default = None*

## MutatorGenerate

The MutatorGenerate property specifies whether to generate mutators for attributes. The possible values are as follows:

- Smart - Mutators are not generated for attributes that have the Constant modifier.
- Always - Mutators are generated, regardless of the modifier.
- Never - Mutators are not generated.

*Default = Smart*

## MutatorVisibility

The MutatorVisibility property specifies the access level of the generated mutator for attributes. This property defines the access level of a mutator operation regardless of the visibility of the attribute. The possible values are as follows:

- fromAttribute - Use the access level for the attribute for the mutator.
- public - Set the mutator access level to public.
- private - Set the mutator access level to private.
- protected - Set the mutator access level to protected. This value is not available in Rational Rhapsody Developer for C.
- default - Set the mutator access level to default. This value is available only in Rational Rhapsody Developer for Java.

*Default = fromAttribute*

## PostDeclarationModifier

The PostDeclarationModifier property is used to allow Rational Rhapsody to reverse engineer non-standard keywords that appear in attribute declarations. Keywords that appear after the attribute name are stored as the value of this property, and the property is then used during code generation to re-create the original code.

Since this is a code generation property, it can also be used to add non-standard keywords to code even when reverse engineering is not used.

This property is used in conjunction with the PreDeclarationModifier and DeclarationModifier properties.

*Default = Blank*

## PreDeclarationModifier

The PreDeclarationModifier property is used to allow Rational Rhapsody to reverse engineer non-standard keywords that appear in attribute declarations. Keywords that appear before the attribute type are stored as the value of this property, and the property is then used during code generation to re-create the original code.

Since this is a code generation property, it can also be used to add non-standard keywords to code even when reverse engineering is not used.

This property is used in conjunction with the DeclarationModifier and PostDeclarationModifier properties.

*Default = Blank*

## RefactorRenameRegularExpression

When you use the Refactor:Rename feature, Rhapsody searches the user-defined code in the model for all occurrences of the name of the specified element, including places where the element name appears as part of the name of elements whose name is derived from the element that is being renamed.

To locate these elements, Rhapsody uses a set of regular expressions.

The regular expressions used for this search are defined by the property RefactorRenameRegularExpression.

Different values are used for this property, depending on what type of element you are renaming. For example, you will see that this property has a different value under Class than it does under Attribute. For some types of elements, Rhapsody uses a more generic expression, taken from the value of the property under ModelElement.

In general, the value of this property is a regular expression that consists of a number of "Or" components. For the renaming to work properly, each such component must contain three groupings, with the middle grouping using the variable \$keyword, which represents the name of the element that is being renamed. In some cases, the expression takes this information from the value of another property rather than using the \$keyword variable.

```
Default = ([^A-Za-z0-9_]|^)( $keyword)([^A-Za-z0-9_]|$) |(^$<CG::Attribute::Mutator>[( $])  
|([^\A-Za-z0-9]$<CG::Attribute::Mutator>[( $]) |(^$<CG::Attribute::Accessor>[( $])  
|([^\A-Za-z0-9]$<CG::Attribute::Accessor>[( $])
```

## ReferenceImplementationPattern

The ReferenceImplementationPattern property specifies how the Reference option for attribute/typedefs (composite types) is mapped to code. See the Rational Rhapsody help for information about composite types.

*Default = \**

## Renames

The Renames property enables one element to rename another element of the same type. You can also rename an element by using a renames dependency. In the case of a conflict, the dependency has precedence. Note the following:

- For attributes, this property works only for static attributes in a class or for attributes in a package.
- For operations, this property contains the name of the operation this operation in renaming. The signatures of the two operations must match.

*Default = Empty string*

## Simplify

If you are using the Rational Rhapsody customizable code generation mechanism, the Simplify property can be used to change the way specific types of elements are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - The element is ignored.
- Copy - The element is copied from the original to the simplified model. It is not modified in any way.
- Default - Uses the standard simplification for this item, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user.
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for the element has been applied.

*Default = "Default"*

## SpecificationEpilog

The SpecificationEpilog property allows you to add code to the end of the declaration of a model element.

For example, you can use this property to add an `#ifdef` to indicate that an operation is available only when the code is compiled with `_DEBUG`, by setting the following properties:

- For SpecificationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For SpecificationEpilog, enter the value `#endif`
- For ImplementationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For ImplementationEpilog, enter the value `#endif`

*Default = Blank*

## SpecificationProlog

The SpecificationProlog property allows you to add code to the beginning of the declaration of a model element.

For example, you can use this property to add an `#ifdef` to indicate that an operation is available only when the code is compiled with `_DEBUG`, by setting the following properties:

- For SpecificationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For SpecificationEpilog, enter the value `#endif`
- For ImplementationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For ImplementationEpilog, enter the value `#endif`

*Default = Blank*

## VariableInitializationFile

The VariableInitializationFile property specifies how global constant variables are initialized. You can initialize these variables in the specification file. You can use these variables as compile-time constants that can be used to define array sizes, for example. Rational Rhapsody automatically identifies constant variables with `const`. By modifying this property, you can choose the initialization file directly. The



possible values are as follows:

- Default - The variable is initialized in the specification file if the type declaration begins with `const`. Otherwise, the variable is initialized in the implementation file.
- Implementation - Initialize global constant variables in the implementation file.
- Specification - Initialize global constant variables in the specification file.

*Default = Default*

## Visibility

The Visibility property specifies the visibility of that kind of model element. Code generation maps the visibility specified for an element to the same visibility in the generated language. The Visibility setting has the following applicability:

- Classes - Applies only to nested classes, which are defined inside other classes.
- Types - Applies only to types that are defined inside classes. It does not apply to global types, which are defined in packages.

The following table lists the visibility for the CPP\_CG subject.

- `protected` - Attribute is visible only within the scope of its class and descendants.
- `private` - Attribute is visible only within its class.
- `public` - Attribute is visible everywhere.
- `fromAttribute` - Attribute visibility depends on the Access selection in the Browser window, which specifies the visibility of accessors and mutators for an attribute.

*Default = protected*

## CallOperation

The CallOperation metaclass contains properties relating to code generation for call operation elements in a model.

## DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text

generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- `$Name` - the element name
- `$FullName` - the full path of the element (`P1::P2::C.a`)
- `$Description` - the text provided on the Description tab of the Features window for the element
- `$Requirements` - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the `$Requirements` keyword in the `DescriptionTemplate` property to get such comments generated for specific model elements.
- `$Login` - the login name of the user who generated the file
- `$CodeGeneratedDate` - the date on which the code was generated
- `$CodeGeneratedTime` - the time at which the code was generated
- `$RhapsodyVersion` - the version of Rhapsody that was used to generate the file
- For operations: `$Signature` - the operation signature, `$Arguments` - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: `$Type` - the type of the attribute
- For arguments: `$Type` - the argument type, `$Direction` - the argument direction (In, Out, or InOut)
- For events: `$Arguments` - the text provided on the Description tab of the Features window for the event arguments
- For association ends: `$Target` - the other end of the association
- For requirements: `$ID` - the ID specified for the requirement, `$Specification` - the specification text for the requirement
- The values of tags can be included by using the syntax `$tag_name`. For example, if you use a tag called `author` for your classes, you can use the keyword `$author` in the value of `DescriptionTemplate` in order to have the value of the `author` tag included in the element description.
- The values of properties can be included by using the syntax `$property_name`. For example, you can use the keyword `$Animate` in the value of `DescriptionTemplate` in order to have the value of the `Animate` property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as `$Name`)
- Tag keywords
- Property keywords

If the value of a keyword is a `MultiLine` value, then each new line (except the first one) starts with the value of the `<lang>_CG::Configuration::DescriptionBeginLine` property, and each line ends with the value of the `<lang>_CG::Configuration::DescriptionEndLine` property.

*Default = State \$Name `[[Description: $Description]]`*

# Class

The Class metaclass contains properties that affect the generated classes.

## AccessTypeName

The AccessTypeName property specifies the name of the access type generated for the class record.

*Default = Empty string*

## ActiveMessageQueueSize

The ActiveMessageQueueSize property specifies the size of the message queue allocated for active classes. The possible values are as follows:

- A string - Specifies the message queue size for an active class.
- An empty string (blank) - The value is set in an operating system-specific manner.

*Default = Empty string*

## ActiveStackSize

The ActiveStackSize property specifies the size of the stack allocated for active objects. The possible values are as follows:

- Any integer - Specifies that a stack of that size is allocated for active objects.
- An empty string (blank) - If not specified, the stack size is set in an operating system-specific manner, based on the value of the ActiveStackSize property for the framework.

*Default = Empty string*

## ActiveThreadName

The ActiveThreadName property indicates the real OS task or thread name. This property only matters when the class is set to active. This facilitates debugging in complex environments in which many threads are constantly being created and deleted on-the-fly. This property is effective for all targets. All strings entered must be enclosed in quotes (" "). The possible values are as follows:

- A string - Names the active thread.
- An empty string (blank) - The value is set in an operating system-specific manner, based on the value of the ActiveThreadName property for the framework.

*Default = Empty string (OS selects thread name)*

## ActiveThreadPriority

The ActiveThreadPriority property specifies the priority of active class threads. The possible values are as follows:

- A string - Specifies thread priority of an active class.
- An empty string (blank) - The value is set in an operating system-specific manner.

*Default = Empty string*

## AdditionalBaseClasses

The AdditionalBaseClasses property adds inheritance from external classes to the model.

*Default = Empty string*

## AdditionalNumberOfInstances

The AdditionalNumberOfInstances property is a string that specifies the size of the local heap allocated for events when the current pool is full. Triggered operations use the event properties.

This property provides support for static architectures found in hard real-time and safety-critical systems without memory management capabilities during run time. All events are dynamically allocated during initialization.

Once allocated, an event queue for a thread remains static in size. The possible values are as follows:

- An empty string (blank) - No additional memory is to be allocated when the initial memory pool is exhausted.
- n (a positive integer) - Specifies the size of the array allocated for additional instances.

*Default = Empty string*

## Animate

The Animate property specifies whether animation code is generated for an element. You can specify your own animation function by using the CPP\_CG::Type::AnimSerializeOperation property.

The semantics of the Animate property is always in favor of the owner settings:

- If a package Animate property is set to Cleared, all the classes owned by the package are not animated, regardless of the class Animate settings.
- If a class Animate property is set to Cleared, all the elements in the class (attributes, operations, relations, and so on) are not animated.
- If an operation Animate property is set to Cleared, all the arguments are not animated.
- If the AnimateArguments property is set to Cleared, all the arguments are not animated, regardless of the specific argument Animate property settings.

*Default = Checked*

## **AnimSerializeOperation**

The animation feature is capable of animating attributes that are of primitive types such as integers or that are one-dimensional arrays of such types. If you want the animation to also include attributes that are based on types or classes you have defined, you must write serialization and unserialization functions for handling these types and classes.

The AnimSerializeOperation property is used to tell Rational Rhapsody the name of the function that should be used for serialization.

The value of the property can be set at the Attribute, Type, or Class level.

If you set the value at the Attribute level, the function specified are only used for that specific attribute.

If you did not set the value at the Attribute level, Rational Rhapsody checks whether the attribute is based on a type or on a class. If the attribute is based on a type, then Rational Rhapsody takes the value of the property defined at the Type level. If the attribute is based on a class, Rational Rhapsody takes the value of the property defined at the Class level.

The property value should consist only of the name of the function. You must make sure that your code contains the include statements that are necessary to find the serialization function.

*Default = Blank*

## **AnimUnserializeOperation**

The animation feature is capable of animating attributes that are of primitive types such as integers or that are one-dimensional arrays of such types. If you want the animation to also include attributes that are based on types or classes you have defined, you must write serialization and unserialization functions for handling these types and classes.

The AnimUnserializeOperation property is used to tell Rational Rhapsody the name of the function that should be used for unserialization.

The value of the property can be set at the Attribute, Type, or Class level.

If you set the value at the Attribute level, the function specified are only used for that specific attribute.

If you did not set the value at the Attribute level, Rational Rhapsody checks whether the attribute is based on a type or on a class. If the attribute is based on a type, then Rational Rhapsody takes the value of the property defined at the Type level. If the attribute is based on a class, Rational Rhapsody takes the value of the property defined at the Class level.

The property value should consist only of the name of the function. You must make sure that your code contains the include statements that are necessary to find the unserialization function.

*Default = Blank*

## AnimUseMultipleSerializationFunctions

The AnimSerializeOperation and AnimUnserializeOperation properties are used to specify user-provided functions for serialization/unserialization of objects to allow inclusion of such objects in animation.

Because Rational Rhapsody allows you to fine-tune code generation of arguments by using the In, Out, InOut, and TriggerArgument properties, you might need to provide multiple serialization/unserialization functions to handle these different types of arguments. You can use the AnimUseMultipleSerializationFunctions property to instruct Rational Rhapsody to use multiple user-provided serialization/unserialization functions.

If you set the value of this property to Checked, Rational Rhapsody searches for user-provided serialization functions whose names consist of the string entered for the AnimSerializeOperation property and the suffixes "In", "Out", "InOut", and "TriggerArgument".

The same is true for unserialization functions. However, for unserialization, Rational Rhapsody cannot handle Out arguments, so the relevant suffixes are "In", "InOut", and "TriggerArgument".

Since the AnimSerializeOperation and AnimUnserializeOperation properties exist under both the Type metaclass and the Class metaclass, the AnimUseMultipleSerializationFunctions property also exists under both these metaclasses.

*Default = Cleared*

## BaseNumberOfInstances

The BaseNumberOfInstances property is a string that specifies the size of the local heap memory pool allocated for either:

- Instances of the class (CPP\_CG::Class)
- Instances of the event (CPP\_CG::Event)
- This property provides support for static architectures found in hard real-time and safety-critical systems without memory management capabilities during run time. All instances of events are dynamically allocated during initialization. Once allocated, the event queue for a thread remains static in size.

Triggered operations use the properties defined for events. When the memory pool is exhausted, an additional amount, specified by the AdditionalNumberOfInstances property, is allocated. Memory pools for classes can be used only with the Flat statechart implementation scheme. The possible values are as follows:

- An empty string (blank) - Memory is always dynamically allocated.
- BaseNumberOfInstances - An array is allocated in this size for instances.

The related properties are as follows:

- AdditionalNumberOfInstances - Specifies the number of instances to allocate if the pool runs out.
- ProtectStaticMemoryPool - Specifies whether the pool should be protected (to support a multithreaded environment)
- EmptyMemoryPoolCallback - Specifies a user callback function to be called when the pool is empty.

This property should be used instead of the `AdditionalNumberOfInstance` property for error handling.

- `EmptyMemoryPoolMessage` - When set to true, this property causes a message to be displayed if the pool runs out of memory in instrumented mode.

*Default = Empty string*

## CodeGeneratorTool

The `CodeGeneratorTool` property specifies which code generation tool to use for the given configuration. The possible values are as follows:

- `Classic` - refers to the older "non-respect" code generation tool.
- `Advanced` - refers to the Rational Rhapsody newer code-respect-oriented code generation tool.
- `External` - instructs Rational Rhapsody to use the registered external code generator.

*Default = Advanced*

## ComplexityForInlining

The `ComplexityForInlining` property specifies the upper bound for the number of lines in user code that are allowed to be inlined. User code is the action part of transitions in statecharts.

For example, when you use the value 3, all transitions with actions consisting of three lines or fewer of code are automatically inlined in the calling function.

Inlining is replacing a function call in the generated code with the actual code statements that make up the body of the function. This optimizes the code execution at the expense of an increase in code size.

For example, increasing the number of lines that can be inlined from 3 to 5 has shortened the code execution time in some cases up to 10%. This property applies only to the Flat implementation scheme for statecharts.

*Default = 0*

## DeclarationModifier

The `DeclarationModifier` property adds a string to the class or event declaration. The string is displayed between the class keyword and the class name in the generated code. For example, for a class A, the `DeclarationModifier` would appear as follows: `class DeclarationModifier> A {...}`; This property adds a modifier to the class declaration. For example, if you have a class `myExportableClass` that is exported from a DLL by using the `MYDLL_API` macro, you can set the `DeclarationModifier` property to "MYDLL\_API." The generated code would then be as follows: `class MYDLL_API myExportableClass {...}`; This property supports two keywords: `$component` and `$class`.

*Default = Empty string*

## DefaultValue

When you use ports, it is possible to have a situation where your application tries to call an operation that is part of a required interface for a port, but the service is not available. In such cases, this might result in problems with the class/type that is supposed to be returned by that operation.

The DefaultValue property allows you to define a value, such as null, that can be returned in such situations.

*Default = Blank*

## DependenciesAutoArrange

The DependenciesAutoArrange property determines the criterion used to organize usage dependencies in the generated code. If the value of the property is set to True, the dependencies are ordered alphabetically in the code. If you set the value to False, the order of dependencies in the code is in accordance with the order that you specified in the "Edit usage dependencies order" window. This property corresponds to the "Use default order" check box in the "Edit usage dependencies order" window.

*Default = True*

## DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- \$Name - the element name
- \$FullName - the full path of the element (P1::P2::C.a)
- \$Description - the text provided on the Description tab of the Features window for the element
- \$Requirements - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the \$Requirements keyword in the DescriptionTemplate property to get such comments generated for specific model elements.
- \$Login - the login name of the user who generated the file
- \$CodeGeneratedDate - the date on which the code was generated
- \$CodeGeneratedTime - the time at which the code was generated
- \$RhapsodyVersion - the version of Rhapsody that was used to generate the file



- For operations: \$Signature - the operation signature, \$Arguments - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: \$Type - the type of the attribute
- For arguments: \$Type - the argument type, \$Direction - the argument direction (In, Out, or InOut)
- For events: \$Arguments - the text provided on the Description tab of the Features window for the event arguments
- For association ends: \$Target - the other end of the association
- For requirements: \$ID - the ID specified for the requirement, \$Specification - the specification text for the requirement
- The values of tags can be included by using the syntax \$tag\_name. For example, if you use a tag called author for your classes, you can use the keyword \$author in the value of DescriptionTemplate in order to have the value of the author tag included in the element description.
- The values of properties can be included by using the syntax \$property\_name. For example, you can use the keyword \$Animate in the value of DescriptionTemplate in order to have the value of the Animate property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as \$Name)
- Tag keywords
- Property keywords

If the value of a keyword is a MultiLine value, then each new line (except the first one) starts with the value of the <lang>\_CG::Configuration::DescriptionBeginLine property, and each line ends with the value of the <lang>\_CG::Configuration::DescriptionEndLine property.

*Default = Blank*

## Destructor

The Destructor property controls the generation of virtual destructors in C++. The property exists for C for historical reasons, with a single value of auto, but it has no effect on the generated C code. The possible values are as follows:

- auto - A virtual destructor is generated for an object only if it has at least one virtual function.
- virtual - A virtual destructor is generated in all cases.
- abstract - A virtual destructor is generated as a pure virtual function.
- common - A nonvirtual destructor is generated.

*Default = auto*

## Embeddable

The Embeddable property is a Boolean property that specifies whether a class can be allocated by value (nested) inside another class or package. For example, if the Embeddable property is True, 20 instances of

a class A can be allocated inside another class by using the following syntax: A itsA[20]; The possible values are as follows:

- **Checked** - The object can be allocated by value inside a composite object or package. The object declaration and definition are generated in the specification file of the composite.
- **Cleared** - The object cannot be embedded inside another object (not supported in RiC). The object declaration and definition are generated in the implementation file of the composite.

The Embeddable property is used with the EmbeddedScalar and EmbeddedFixed properties to determine how to generate code for an embedded object. It is also closely related to the ImplementWithStaticArray property, which also needs to be set in order to support by-value allocation.

Relations can be generated by value only under the following circumstances:

- The multiplicity of the relation is well-defined (not “\*”).
- The ImplementWithStaticArray property of the component relation is set to FixedAndBounded.

When the Embeddable property is Cleared (RiC only):

- The attributes of the object are encapsulated. Clients of the object are forced to use it only by way of its operations, because there is no direct access to its attributes.
- Dynamic allocation must be used. The compiler does not know how to statically allocate an object when its declaration is not visible.

*Default = Cleared*

## EnableDynamicAllocation

The EnableDynamicAllocation property specifies whether to use dynamic memory allocation for objects. The possible values are as follows:

- **Checked** - Dynamic allocation of events is enabled. Create() and Destroy() operations are generated for the object or object type.
- **Cleared** - Events are dynamically allocated during initialization, but not during run time. Create() and Destroy() operations are not generated for the object. This setting is recommended for static architectures that do not use dynamic memory management during run time.

*Default = Checked*

## EnableUseFromCPP

The EnableUseFromCPP property specifies whether to wrap C operations with an appropriate extern C { } wrapper to prevent problems when code is compiled with a C++ compiler. Wrapping C code with extern C includes C code in a C++ application.

Note that the structure definition for the object is not wrapped - only the functions are.

For example, if the EnableUseFromCPP is set to Checked for an object, the following wrapper code is generated for its operations:

```
#ifdef __cplusplus extern "C" { #endif /* __cplusplus */ /* Operations */ #ifdef __cplusplus } #endif /*
```

`__cplusplus */`

*Default = Cleared*

## ExpandNameKeywordWithTemplateParameters

Prior to release 8.1.5, when you used properties to define standard operations, the keywords \$Name and \$NameWithTemplateParams would both result in the generation of the class name together with the template parameters of a template class. This issue was corrected in release 8.1.5.

In order to preserve the previous code generation behavior for pre-8.1.5 models, the property ExpandNameKeywordWithTemplateParameters was added to the backward compatibility settings for C++, with a value of True.

## Final

The Final property, when set to False, specifies that the generated record for the class is a tagged record. This property applies to Ada95.

*Default = False*

## Friend

The Friend property specifies friends to be added to class declarations. For example, if you specify “int t(); class x”, the following lines is generated in the public section of all class declarations: friend int t(); friend class x; Separate multiple friends with semicolons.

*Default = Empty string*

## GenClassAsStruct

When generating C++ code, Rational Rhapsody generates classes in your model as C++ classes in the code. While this is the default behavior, it is also possible to have Rational Rhapsody generate classes as structs in your C++ code. The GenClassAsStruct property allows you to specify that a class should be generated as a struct.

*Default = False*

## GenDefaultVisibility

When Rhapsody generates C++ code, the visibility of class members is explicitly stated, even when a member uses the default visibility (private in class, public in struct). If you prefer to have the generated code leave out the visibility in such cases, set the value of the property GenDefaultVisibility to False.

*Default = True*

## GenerateAccessType

The GenerateAccessType property determines which access types are generated for the class. The possible values are as follows:

- None - Access types are not generated.
- Standard - An access type is generated.
- General - General access types are generated.

*Default = General*

## GenerateClassDescriptionInSpecification

Prior to release 8.1 of Rhapsody, if the \$requirements keyword was used to include requirement text in the class description that is generated in implementation files, there were cases where the text ended up instead in the class description in the specification file. This issue was fixed in release 8.1. To preserve the previous code generation behavior for pre-8.1 models, the property GenerateClassDescriptionInSpecification was added to the 8.1 backward compatibility settings for C++ with a value of True.

## GenerateDestructor

The GenerateDestructor property specifies whether to generate a destructor for a class.

*Default = Checked*

## GenerateLocalDeclareGuard

If a class that contains guarded operations is derived from multiple base classes, each of which contains guarded operations, you may encounter build errors that refer to ambiguity of the operation 'getGuard'. To resolve these build errors, set the value of the GenerateLocalDeclareGuard property to True for the derived class.

Note that you may also encounter these errors if your class is derived from a user-defined class that contains guarded operations, and also derives from a class from the Rhapsody framework that contains guarded operations, for example, OMThread, which is a base class for any class specified as "active".

*Default = False*

## GenerateRecordType

The GenerateRecordType property determines whether the class record is generated.

*Default = True*

## HasUnknownDiscriminant

The HasUnknownDiscriminant property determines whether an unknown discriminant ) is generated for this class.

*Default = False*

## ImplIncludes

The ImplIncludes property specifies the names (including full paths) of header files to be included at the top of implementation files generated for classes, objects or object types, or packages. Separate multiple file names by using commas, without spaces.

*Default = Empty string*

## ImplementationEpilog

The ImplementationEpilog property enables you to add any code that you want to be added as verbatim text (to be ignored by Rational Rhapsody) to the end of the definition of a model element.

For example, you could wrap a section of code with an #ifdef-#endif pair, add compiler-specific keywords, or add a #pragma statement. For example, to specify that an operation is available only when the code is compiled with \_DEBUG, set the following properties for the operation:

- Set SpecificationProlog to #ifdef \_DEBUG cr.
- Set SpecificationEpilog to #endif.
- Set ImplementationProlog to #ifdef \_DEBUG cr.
- Set ImplementationEpilog to #endif.

The following table lists whether leading and trailing linefeeds are generated.

Metaclass Leading Linefeed Added? Class Yes Package No

*Default = Empty MultiLine*

## ImplementationPragmas

*The ImplementationPragmas property specifies the user-defined pragmas to generate in the body. Default = Empty MultiLine*

## ImplementationPragmasInContextClause

*The ImplementationPragmasInContextClause property specifies the user-defined pragmas to generate in the context clause of the body. Default = Empty MultiLine*

## ImplementationProlog

The ImplementationProlog property enables you to add any code that you want to be added as verbatim text (to be ignored by the software) to the beginning of the definition of a model element.

For example, you could wrap a section of code with an `#ifdef-#endif` pair, add compiler-specific keywords, or add a `#pragma` statement. For example, to specify that an operation is available only when the code is compiled with `_DEBUG`, set the following properties for the operation:

- Set SpecificationProlog to `#ifdef _DEBUG cr.`
- Set SpecificationEpilog to `#endif.`
- Set ImplementationProlog to `#ifdef _DEBUG cr.`
- Set ImplementationEpilog to `#endif.`

The following table lists whether leading and trailing linefeeds are generated.

Metaclass Trailing Linefeed Added? Class No Package Yes

*Default = Empty MultiLine*

## In

The In property specifies how code is generated when the type is used with an argument that has the modifier In. When a class is used with the "In" modifier, the default is "const \$type&" in C++.

## InitCleanUpRelations

The InitCleanUpRelations property specifies whether to generate `initRelations()` and `cleanUpRelations()` operations for sets of related global instances. This property applies only to composites and global relations.

*Default = True*

## InitializationCode

The InitializationCode property adds the specified initialization code in the body of the class. A non-abstract class can have initialization code that is executed during elaboration of the associated package. (empty MultiLine)

## InOut

The InOut property specifies how code is generated when the type is used with an argument that has the modifier InOut. When a class is used with the "InOut" modifier, the default is "\$type&" in C++.

## InstanceDeclaration

The InstanceDeclaration property specifies how instances are declared in code. The default value for C is as follows: struct \$cname\$suffix In the generated code, the variable \$cname is replaced with the object (or object type) name. The variable \$suffix is replaced with the type suffix "\_t," if the object is of implicit type.

*Default = \$cname\$suffix*

## IsCompletedOperation

The IsCompletedOperation specifies whether state\_IS\_COMPLETED operations are generated as functions or macros (by using #define). The possible values are as follows:

- Plain - state\_IS\_COMPLETED operations are generated as functions (pre-V4.2 behavior). This is the default value.
- Inline - state\_IS\_COMPLETED operations are generated by using #define macros, if the body contains only a return statement.

*Default = Plain*

## IsInOperation

The IsInOperation specifies how state\_IN methods are generated.

In Rational Rhapsody Developer for C++, this property specifies whether state\_IN methods are virtual or nonvirtual. For classes with state machines (statecharts or activity diagrams), Rational Rhapsody generates state\_IN operations.

By default, these operations are nonvirtual, but you can make them virtual by setting this property to Virtual. This value is needed to support statechart inheritance in Flat mode.

*Default = Default*

## IsLimited

The IsLimited property determines whether the class or record type is generated as limited.

*Default = False*

## IsNested

The IsNested property specifies whether to generate the class or package as nested.

*Default = False*

## IsPrivate

The IsPrivate property specifies whether to generate the class or package as private.

*Default = False*

## IsReactiveInterface

The IsReactiveInterface property modifies the way reactive classes are generated. It has the following effects:

- Virtual inheritance from OMReactive
- Prevents instrumentation
- Prevents the thread argument and the initialization code (setting the active context) in the class constructor
- Creates a pure-virtual destructor (by default)

This property affects only classes that declare themselves as interfaces by having a stereotype with a name that contains the word “interface” (case-insensitive).

In previous versions of Rational Rhapsody, a class could inherit from a single reactive class only, regardless of whether it was an interface or implementation class. Beginning with Version 4.0.1 MR2, a class can inherit (implement) several reactive interfaces.

In Rational Rhapsody Developer for C++, you must explicitly designate reactive interfaces because the code generator applies special translation rules involving multiple inheritance from the Rational Rhapsody framework. You can designate a reactive interface in two ways:

- Set the CPP\_CG::Class::IsReactiveInterface property to checked.
- Use the predefined stereotype Reactive\_interface. This stereotype uses stereotype-based code generation in order to automatically apply the correct property value.

Alternatively, you can define another stereotype (such as PortSpec) that sets IsReactiveInterface to true and use that stereotype. A class is considered reactive if it meets all the following conditions:

- The CPP\_CG::Framework::ReactiveBase property is not empty.
- The CPP\_CG::Framework::ReactiveBaseUsage property is set to Checked.
- One or more of the following conditions are checked:
  - The class has a statechart or activity diagram.
  - The class is a composite class.
  - The class has event receptions or triggered operations.

*Default = Cleared*

Rational Rhapsody Developer for C++ A reactive interface:

- Automatically uses virtual inheritance from its base reactive class.
- Does not override reactive methods (such as startBehavior().)



- Does not call reactive initialization methods (such as `setThread()`).
- By default, has a pure virtual destructor (when the `CPP_CG::Class::Destructor` property is set to `auto`).
- Cannot have a statechart or activity diagram.
- Cannot be a composite class.

A class that inherits from a reactive interface:

- Overrides reactive methods (such as `startBehavior()`.)
- Ignores the reactive interfaces when overriding reactive methods. It calls reactive initialization methods directly (such as calling `setThread()` in its constructor).

## MangleNestedInAnimation

The `MangleNestedInAnimation` property is used to specify that when nested classes are animated, the name used for the inner class should be mangled. For example, if class B is nested in class A, the name `A_B` would be used.

*Default = Cleared*

## MarkPrologEpilogInAnnotations

The `MarkPrologEpilogInAnnotations` property specifies whether to generate ignore annotations for the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties so they are ignored during roundtrip. When you insert code element declarations (variables, types, functions, and so on) in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, after a full roundtrip those elements are added to the model and are duplicated on the next code generation.

When you use the `MarkPrologEpilogInAnnotations` property, you can have Rational Rhapsody automatically ignore the information specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties instead of adding the ignore annotations manually. The possible values for the `MarkPrologEpilogInAnnotations` property are:

- None - Rational Rhapsody does not generate any annotations. Any models created before Version 4.1 automatically have this property setting.
- Ignore - Rational Rhapsody generates the `/// ignore annotation before the code specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, and generates the /// annotation after the code specified in those properties.`
- Auto - If the code in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties is one line (it does not contain any newline characters `(\n)`), no annotations are generated (the same behavior has the None setting). If there is more than one line, Rational Rhapsody generates the `/// ignore annotation before the code specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, and generates the /// annotation after the code specified in those properties (the same behavior as the Ignore setting).`

*During roundtrip, any ignore annotations in the comments of the element are not included in its description. Because the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties are generated between the annotation and its declaration for the element, you cannot rename those elements on roundtrip. If you change the name of an element, it is*

*removed from the model and added with the new name. Some model information (for example, property settings) might be lost. Default = None*

## MaximumPendingEvents

The MaximumPendingEvents property specifies the maximum number of events that can be simultaneously pending in the event queue of the active class. The possible values are:

- -1 - Memory is dynamically allocated.
- Positive integer - Specifies the maximum number of events.

*Default = -1*

## MultiLineInitializerList

The MultiLineInitializerList property specifies how the code for the class initialization list should be formatted.

If the property is set to False, the class initialization list is generated on a single line.

If the property is set to True, the class initialization list is generated on multiple lines such that each attribute appears on a different line.

*Default = False*

## NestingVisibility

*The NestingVisibility property specifies the visibility of the generated specification of the nested class or package. Default = Public*

## ObjectTypeAsSingleton

The ObjectTypeAsSingleton property generates singleton code for object-types and actors. This functionality saves a singleton-type (actor) in its own repository unit, and manage that unit by using a configuration management tool. Set this property for a single object-type or higher. An object-type is generated as a singleton when all of the following conditions are met:

- The object-type has the «Singleton» stereotype.
- There is one and only one object of the object-type and the object multiplicity is 1.
- The ObjectTypeAsSingleton property is set to True.

Note that when you expose a singleton object (for example, by creating a singleton object-type), Rational Rhapsody also modifies the code that is generated for the singleton.

*Default = False*

## OpeningBraceStyle

The OpeningBraceStyle property controls where the opening brace of the code block is positioned - on the same line as the element name (SameLine) or on the following line (NewLine).

## OptimizeStatechartsWithoutEventsMemoryAllocation

The OptimizeStatechartsWithoutEventsMemoryAllocation property determines whether the generated code uses dynamic memory allocation for statecharts that use only triggered operations.

*Default = False*

## Out

The Out property specifies how code is generated when the type is used with an argument that has the modifier "Out."

When a class is used with the "Out" modifier, the default is "\$type\*&" in C++.

## ReactiveInterfaceScheme

The property ReactiveInterfaceScheme determines which framework class serves as the base class for a reactive interface.

If the property value is set to Full, the interface inherits from OMReactive.

If the property value is set to Thin, the interface inherits from IOxfEventSender.

Note that IOxfEventSender includes only operations related to event sending, while OMReactive includes also attributes and operations related to statechart behavior.

*Default = Full*

## ReactiveThreadSettingPolicy

The ReactiveThreadSettingPolicy property specifies how threads are set for reactive classes. The possible values are as follows:

- Default - During code generation, Rational Rhapsody adds a thread argument to the constructor.
- MainThread - Rational Rhapsody does not add an argument; the thread is set to the main thread.
- UserDefined - Rational Rhapsody does not add an argument; you must set the value for the thread yourself.

*Default = Default*

## RecordTypeName

*The RecordTypeName property specifies the name of the class record type. If this is not set, Rational Rhapsody uses class\_name\_t. Default = Empty string*

## RefactorRenameRegularExpression

When you use the Refactor:Rename feature, Rhapsody searches the user-defined code in the model for all occurrences of the name of the specified element, including places where the element name appears as part of the name of elements whose name is derived from the element that is being renamed.

To locate these elements, Rhapsody uses a set of regular expressions.

The regular expressions used for this search are defined by the property RefactorRenameRegularExpression.

Different values are used for this property, depending on what type of element you are renaming. For example, you will see that this property has a different value under Class than it does under Attribute. For some types of elements, Rhapsody uses a more generic expression, taken from the value of the property under ModelElement.

In general, the value of this property is a regular expression that consists of a number of "Or" components. For the renaming to work properly, each such component must contain three groupings, with the middle grouping using the variable \$keyword, which represents the name of the element that is being renamed. In some cases, the expression takes this information from the value of another property rather than using the \$keyword variable.

```
Default = (^its|[^A-Za-z0-9_]its |^$<CG::Relation::Add>Its |[^A-Za-z0-9_] $<CG::Relation::Add>Its  
|^$<CG::Relation::Clear>Its |[^A-Za-z0-9_] $<CG::Relation::Clear>Its  
|^$<CG::Relation::CreateComponent>Its |[^A-Za-z0-9_] $<CG::Relation::CreateComponent>Its  
|^$<CG::Relation::DeleteComponent>Its |[^A-Za-z0-9_] $<CG::Relation::DeleteComponent>Its  
|^$<CG::Relation::Find>Its |[^A-Za-z0-9_] $<CG::Relation::Find>Its |^$<CG::Relation::Get>Its  
|[^A-Za-z0-9_] $<CG::Relation::Get>Its |^$<CG::Relation::Remove>Its  
|[^A-Za-z0-9_] $<CG::Relation::Remove>Its |^$<CG::Relation::Set>Its  
|[^A-Za-z0-9_] $<CG::Relation::Set>Its |^$<CG::Relation::GetKey>Its  
|[^A-Za-z0-9_] $<CG::Relation::GetKey>Its |^$<CG::Relation::GetAt>Its  
|[^A-Za-z0-9_] $<CG::Relation::GetAt>Its |^$<CG::Relation::RemoveKey>Its  
|[^A-Za-z0-9_] $<CG::Relation::RemoveKey>Its ) ($keyword:c)([^A-Za-z0-9_] |_[1-9])+[^A-Za-z0-9_] |$)  
|([A-Za-z0-9_] |^)( $keyword)([A-Za-z0-9_] |$)
```

## RelativeEventDataRecordTypeComponentsNaming

The RelativeEventDataRecordTypeComponentsNaming property enables relative naming of event data record type components that represent events and triggered operation parameters. If this is True, no events or triggered operations share argument names because they would generate record components with the same name (which would not compile).

*Default = False*

## Renames

The Renames property enables one element to rename another element of the same type. You can also rename an element by using a renames dependency. In the case of a conflict, the dependency has precedence. Note the following:

- For attributes, this property works only for static attributes in a class or for attributes in a package.
- For operations, this property contains the name of the operation this operation in renaming. The signatures of the two operations must match.

*Default = Empty string*

## ReturnType

The ReturnType property specifies how code is generated when the type is used as a return type.

When a class is used with the "ReturnType" modifier, the default is "\$type\*" in C++.

## Simplify

If you are using the Rational Rhapsody customizable code generation mechanism, the Simplify property can be used to change the way specific types of elements are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - The element is ignored.
- Copy - The element is copied from the original to the simplified model. It is not modified in any way.
- Default - Uses the standard simplification for this item, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user.
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for the element has been applied.

*Default = "Default"*

## SimplifyConstructors

If you are using the Rational Rhapsody customizable code generation mechanism, the SimplifyConstructors property can be used to change the way constructors are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - The constructors is ignored.
- Copy - The constructorare copied from the original to the simplified model. They do not be modified in any way.

- Default - Uses the standard simplification for constructors, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user.
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for constructors has been applied.

*Default = "Default"*

## **SimplifyDestructors**

If you are using the Rational Rhapsody customizable code generation mechanism, the SimplifyDestructors property can be used to change the way destructors are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - The destructors are ignored.
- Copy - The destructors are copied from the original to the simplified model. They are not modified in any way.
- Default - Uses the standard simplification for destructors, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user.
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for destructors has been applied.

*Default = "Default"*

## **SimplifyPackageFiles**

If you are using the Rational Rhapsody customizable code generation mechanism, the SimplifyPackageFiles property can be used to change the way File elements are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - File elements are ignored.
- Copy - File elements are copied from the original to the simplified model. They are not modified in any way.
- Default - Uses the standard simplification for File elements, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user.
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for File elements has been applied.

*Default = "Default"*

## **SingletonExposeThis**

The SingletonExposeThis property, when set to False, specifies that all non-static methods are considered

as static methods and does not pass in this parameter.

*Default = False*

## SpecificationEpilog

The SpecificationEpilog property allows you to add code to the end of the declaration of a model element.

For example, you can use this property to add an `#ifdef` to indicate that an operation is available only when the code is compiled with `_DEBUG`, by setting the following properties:

- For SpecificationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For SpecificationEpilog, enter the value `#endif`
- For ImplementationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For ImplementationEpilog, enter the value `#endif`

*Default = Blank*

## SpecificationPragmas

*The SpecificationPragmas property specifies the user-defined pragmas to generate in the specification.*

*Default = Empty MultiLine*

## SpecificationPragmasInContextClause

*The SpecificationPragmasInContextClause property specifies the user-defined pragmas to generate in the context clause of the specification. Default = Empty MultiLine*

## SpecificationProlog

The SpecificationProlog property allows you to add code to the beginning of the declaration of a model element.

For example, you can use this property to add an `#ifdef` to indicate that an operation is available only when the code is compiled with `_DEBUG`, by setting the following properties:

- For SpecificationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For SpecificationEpilog, enter the value `#endif`
- For ImplementationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For ImplementationEpilog, enter the value `#endif`

*Default = Blank*

## SpecIncludes

*The SpecIncludes property specifies the names (including full paths) of header files to be included at the top of specification files generated for classes (C++ and Java), objects or object types (C), and packages. Separate multiple file names by using commas, without spaces. Default = Empty string*

## TaskBody

*The TaskBody property defines an alternate task body for Ada Task and Ada Task Type classes. Default = Empty string*

## TriggerArgument

The TriggerArgument property specifies how the type should be passed in when used as an argument for events/triggered operations. By default, classes that are used as an argument are passed by way of a pointer. There are 4 other properties that effect how types are passed into and returned by regular operations: "in", "out", "in/out" and "return."

*Default = \$type\**

See also:

- In
- InOut
- Out

## Visibility

The Visibility property specifies the visibility of the model element. Code generation maps the visibility specified for an element to the same visibility in the generated language. The possible values are as follows:

- Public - The model element is public.
- Protected - The model element is protected.
- Private - The element is private.

See "Visibility" for more information.

*Default = Public*

## Configuration

The Configuration metaclass contains properties that affect the configuration.

## ClassesPerCGCall



The `ClassesPerCGCall` property can be used to specify the maximum number of classes that Rational Rhapsody should include in a single code generation "chunk". Above this number, the code generation action will be broken into a number of smaller code generation actions. For example, if you specify 500 for the value of the property, then if your model has 501-1000 classes, Rational Rhapsody will try to break the code generation action into two smaller code generation actions.

Note that while the property name includes the term `Classes`, this number also takes into account similar model elements, such as actors and files.

If the value is set to -1, the chunking mechanism is not used, meaning that the code generation action will not be broken into a number of smaller actions regardless of how large the model is.

*Default = -1*

## ClassStateDeclaration

The `ClassStateDeclaration` property supports C compilers that cannot handle enum declarations inside struct declaration. The possible values are as follows:

- `InClassDeclaration` - Generate the reactive statechart enum declaration in the class declaration (as in Rational Rhapsody 3.0.1).
- `BeforeClassDeclaration` - Generate the reactive class statechart enum declaration before the declaration of the class.

*Default = InClassDeclaration*

## CodeGenerationDirectoryLevel

The `CodeGenerationDirectoryLevel` property is found in the pre-72 compatibility profiles for C and C++.

Before version 7.2 of Rational Rhapsody, the directories specified with the `DefaultSpecificationDirectory` and `DefaultImplementationDirectory` properties were created at the beginning of the path to the generated files, for example, `..\spec_directory\package_a\subpackage_1` and `..\impl_directory\package_a\subpackage_1`.

Beginning with version 7.2 of Rational Rhapsody, the directories specified with `DefaultSpecificationDirectory` and `DefaultImplementationDirectory` are created at the end of the path to the generated files, for example, `..\package_a\subpackage_1\spec_directory` and `..\package_a\subpackage_1\impl_directory`.

To provide the old code generation behavior for pre-72 models, the compatibility profiles include the `CodeGenerationDirectoryLevel` property, with the default value of the property set to `Top`. If you want your pre-72 models to use the new behavior that was introduced in version 7.2, change the value of this property to `Bottom`.

*Default = Top*

## CodeGeneratorTool

The CodeGeneratorTool property specifies which code generation tool to use for the given configuration. The possible values are as follows:

- Advanced - Rational Rhapsody uses its internal code generator to generate code
- External - instructs Rational Rhapsody to use the registered external code generator

*Default = Advanced*

## CodeUpdate

The CodeUpdate property is responsible for the selective code generation used in code-centric mode.

In code-centric mode, the code-generation behavior is based on the premise that if you add any code-related elements to your model, you would prefer that Rational Rhapsody make as few changes as possible to your code.

*Default = True (in code-centric settings)*

## ContainerSet

The ContainerSet property specifies the container set used to implement relations. The possible C++ values are as follows:

OMContainers (default) OMCorba2CorbaContainers OMCpp2CorbaContainers  
OMCppOfCorbaContainers OMUContainers STLContainers

## DefaultActiveGeneration

The DefaultActiveGeneration property specifies whether the default active class is created, as well as the classes for which it acts as the active context. The possible values are as follows:

- Disable - The default active singleton is not created.
- ReactiveWithoutContext - The default active singleton is created if there are reactive classes that consume events and do not have an active context explicitly specified. The default active singleton can handle only these classes.
- All - The default active singleton is generated if there is at least one event-consuming reactive class and the active singleton can handle all reactive classes that consume events - even those reactive classes that specify another active class as their active context.

*Default = ReactiveWithoutContext*

## DefaultImplementationDirectory

The DefaultImplementationDirectory property specifies the relative path to the default directory for generated implementation files. The value of this property is added after the configuration path. Consider the following case:

- File C.cpp is an implementation of class C mapped to a folder Foo.

- The active configuration (cfg) is under component cmp.
- DefaultImplementationDirectory is set to “src”

Rational Rhapsody generates C.cpp to root>\cmp\cfg\src\Foo. Note the following limitations:

- This feature is not supported in COM- or CORBA-related components (C++ only).
- The predefined OSE environments (OsePPCDiab and OseSfk) are not supported due to makefile flexibility issues.
- This feature is not supported by the INTEGRITY adapter build file generator.

*Default = Empty string*

## DefaultSpecificationDirectory

The DefaultSpecificationDirectory property specifies the relative path to the default directory for generated specification files. The value of this property is added after the configuration path. Consider the following case:

- File B.h is a specification of class B that is not mapped to any file.
- The active configuration (cfg) is under component cmp.
- DefaultSpecificationDirectory is set to “inc”

Rational Rhapsody generates B.h to root>\cmp\cfg\inc. Note the following limitations:

- This feature is not supported in COM- or CORBA-related components (C++ only).
- The predefined OSE environments (OsePPCDiab and OseSfk) are not supported due to makefile flexibility issues.
- This feature is not supported by the INTEGRITY adapter build file generator.

*Default = Empty string*

## DependencyRuleScheme

The DependencyRuleScheme property specifies how dependency rules should be generated in the makefile. The possible values are as follows:

- Basic - Generates only the local implementation and specification files in the dependency rule in the makefile.
- ByScope - In addition to generating the same files as the Basic option, generates the specification files of related elements (dependencies, associations, generalizations, and so on) that are in the scope of the active component.
- This option corresponds to the Rational Rhapsody 5.0.1 behavior.
- Extended - In addition to generating the same files as the ByScope option, generates the specification files of related external elements (specified by using the CG::Class:UseAsExternal and CG::Package:UseAsExternal properties) and elements that are not in the scope of the active component.

*Default = ByScope*

## DescriptionBeginLine

This property specifies the prefix for the beginning of comment lines in the generated code.

The property can be used to customize the generation of comments for element descriptions so that the comments match the comment styles required by documentation generators such as Doxygen.

This property affects only the code that is generated for descriptions of model elements; other auto-generated comments are not affected.

*Default = //*

## DescriptionEndLine

This property specifies the symbols for the end of comment lines in the generated code.

The property can be used to customize the generation of comments for element descriptions so that the comments match the comment styles required by documentation generators such as Doxygen.

This property affects only the code that is generated for descriptions of model elements; other auto-generated comments are not affected.

## EmptyArgumentListName

The EmptyArgumentListName specifies the string generated for the argument list when an operation has no arguments. For example, if you set this value to "void," for an operation foo that has no arguments, Rational Rhapsody generates the following code:

```
int foo (void){ ... }
```

*Default = Empty string*

## Environment

The Environment property determines the target environment for a configuration. Generated code is targeted for that environment. See the Release Notes for the environments supported by Rational Rhapsody “out-of-the-box.”

“Out-of-the-box” support means that Rational Rhapsody includes a set of preconfigured code generation properties for the environment and precompiled versions of the relevant OXF libraries. The precompiled OXF libraries have been fully tested. You can also add new environments, for example if you want to generate code for another RTOS.

This involves retargeting the OS wrapper files in the Rational Rhapsody framework and creating a new set of code generation properties for the target environment.

*Default = Microsoft*

## ExternalGenerationTimeout

The ExternalGenerationTimeout property specifies how long, in seconds, Rational Rhapsody waits for the each class in the configuration scope to complete so you can once again make changes to the model.

This property applies to both the full-featured external generator and makefile generator. For example, if you set this property to 2 and you have 10 classes, Rational Rhapsody sets a timeout of 20.

If the external code generator does not complete generation in this timeframe, Rational Rhapsody displays a message in the output window saying that the generator is not responding, and you are allowed to make changes to the model.

If you set this property to 0, Rational Rhapsody does not time out the generation session, and waits for the code generator to complete its task - even if it takes forever. Rhapsody waits for a notification from the full-featured external code generator, or for the process termination of a makefile generator.

*Default = 0*

## ExternalGeneratorFileMappingRules

The ExternalGeneratorFileMappingRules property specifies whether the external code generator uses the same file mapping and naming scheme ( mapping rules ) as Rational Rhapsody.

If the mapping rules are different , the external generator must implement handlers to the GetFileName, GetMainFileName, and GetMakefileName events that Rational Rhapsody runs to get a requested file name and path. The possible values are as follows:

- AsRhapsody - The external generator uses the same mapping rules as Rational Rhapsody.
- DefinedByGenerator - The external generator has its own mapping rules.

*Default = AsRhapsody*

## GenerateAnnotationsForNonSPARKConfigurations

The GenerateAnnotationsForNonSPARKConfigurations property specifies whether

*Default = False*

## GenerateDirectory

The GenerateDirectory property is used to specify that the code files for classes and files in a package should be generated in a separate directory that has the same name as the package.

If the property is set to False, the code files will be generated in a single directory that contains the generated files for all packages for which this property is set to False.

Note that if this property is set to False and your model contains classes with the same name in different

packages, the generated files for these classes will overwrite the previous file with the same name. This will leave you with only one generated file even though the model contains a number of classes with that name.

*Default = False*

## GenerateSingleIDLFile

Beginning in version 8.0.3, a separate IDL file is generated for each package that contains DDS elements. To preserve the previous code generation behavior for pre-8.0.3 models, the `CPP_CG::Configuration::GenerateSingleIDLFile` property was added to the backward compatibility settings for C++ with a value of `True`.

## GeneratorExtraPropertyFiles

The `GeneratorExtraPropertyFiles` property opens the default Text Editor allowing the user to edit the `$OMROOT\CodeGenerator\GenerationRules\LangC\ReC_CG.ini` file.

## GeneratorRulesSet

*The `GeneratorRulesSet` property specifies your own rules set. Default = Empty MultiLine*

## GeneratorScenarioName

*The `GeneratorScenarioName` property specifies the scenario name for the rule, if you write your own set of code generation rules. Default = Empty string*

## GenericEventHandling

The `GenericEventHandling` property is a Boolean value that determines whether to generate generic event-handling code.

This property supports large-scale collaboration, where you might not be aware of which classes consume a base event of your part in the event hierarchy, and might not have access to parts of the model that use base events.

Beginning with Rational Rhapsody 4.0, the framework base event class includes a new, virtual method that checks the event ID against the specified ID, thereby supplying a generic mechanism for events without super events. The language-specific methods are as follows: C:

```
#define RiCEvent_isTypeOf(event, id) ((event)-IId == (id)) C++: virtual OMBoolean isTypeOf(short id)
const {return IId ==id;}
```

In addition, C++ includes a new macro, `IS_EVENT_TYPE_OF(id)`, to support both reusable and flat code generation schemes.

Java:

```
Boolean isTypeOf(long id) {return lId == id;}
```

Each generated event that has a super event overrides the method to check the ID against its own ID, then calls its base event directly to continue the check. An event without a base event returns Cleared if the ID does not equal its own.

When you set the GenericEventHandling property to Cleared, event consumption code is generated as in version 3.0.1. Setting this property affects only the way events are consumed - the override on the isTypeOf() method is still generated, to allow handling of events in components that use the generic event handling.

To support complete generic event handling, you should regenerate the code for all events and reactive classes.

*Default = Checked*

## ImplementationEpilog

The ImplementationEpilog property enables you to add any code that you want to be added as verbatim text (to be ignored by Rational Rhapsody) to the end of the definition of a model element. For example, you could wrap a section of code with an #ifdef-#endif pair, add compiler-specific keywords, or add a #pragma statement.

For example, to specify that an operation is available only when the code is compiled with \_DEBUG, set the following properties for the operation:

- Set SpecificationProlog to #ifdef \_DEBUG cr.
- Set SpecificationEpilog to #endif.
- Set ImplementationProlog to #ifdef \_DEBUG cr.
- Set ImplementationEpilog to #endif.

The following table lists whether leading and trailing linefeeds are generated.

Metaclass	Leading	Linefeed	Added?	Class	Yes	Package	No
-----------	---------	----------	--------	-------	-----	---------	----

*Default = Empty MultiLine*

## ImplementationProlog

The ImplementationProlog property enables you to add any code that you want to be added as verbatim text (to be ignored by the software) to the beginning of the definition of a model element. For example, you could wrap a section of code with an #ifdef-#endif pair, add compiler-specific keywords, or add a #pragma statement.

For example, to specify that an operation is available only when the code is compiled with \_DEBUG, set the following properties for the operation:

- Set SpecificationProlog to `#ifdef _DEBUG cr.`
- Set SpecificationEpilog to `#endif.`
- Set ImplementationProlog to `#ifdef _DEBUG cr.`
- Set ImplementationEpilog to `#endif.`

The following table lists whether leading and trailing linefeeds are generated.

Metaclass Trailing Linefeed Added? Class No Package Yes

*Default = Empty MultiLine*

## Indentation

The Indentation property specifies the number of spaces that should be used for indentation when Rhapsody generates code.

Note that this property does not affect the indentation used in operation bodies since that code is provided by the user.

*Default = 4*

## InitializeEmbeddableObjectsByValue

The InitializeEmbeddableObjectsByValue property specifies whether embeddable classes and object types selected in the configuration initial instances list should be allocated by value in the main() routine.

*Default = Cleared*

## LocalVariablesDeclaration

*The LocalVariablesDeclaration property specifies variables that you want to appear in the declaration of the entrypoint or operation. Default = Empty MultiLine*

## MainFunctionArgList

This property provides a list of the main function arguments. The default list is "int argc, char\* argv[]."

*Default = int argc, char\* argv[]*

## MainFunctionRetType

The MainFunctionRetType property determines the return type for the "main" function generated by Rational Rhapsody. Note that if the value of the property is a blank string, the return type generated will be "int".



*Default = Blank*

## MarkPrologEpilogInAnnotations

The MarkPrologEpilogInAnnotations property specifies whether to generate ignore annotations for the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties so they are ignored during roundtrip.

When you insert code element declarations (variables, types, functions, and so on) in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, after a full roundtrip those elements are added to the model and are duplicated on the next code generation.

When you use the MarkPrologEpilogInAnnotations property, you can have Rational Rhapsody automatically ignore the information specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties instead of adding the ignore annotations manually. The possible values for the MarkPrologEpilogInAnnotations property are:

- None - Rational Rhapsody does not generate any annotations. Any models created before Version 4.1 automatically have this property setting.
- Ignore - Rational Rhapsody generates the `///  
[ ignore annotation before the code specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, and generates the ///  
[ annotation after the code specified in those properties.`
- Auto - If the code in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties is one line (it does not contain any newline characters `\n`), no annotations are generated (the same behavior has the None setting). If there is more than one line, Rational Rhapsody generates the `///  
[ ignore annotation before the code specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, and generates the ///  
[ annotation after the code specified in those properties (the same behavior as the Ignore setting).`

During roundtrip, any ignore annotations in the comments of the element are not included in its description. Because the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties are generated between the annotation and its declaration for the element, you cannot rename those elements on roundtrip. If you change the name of an element, it is removed from the model and added with the new name. Some model information (for example, property settings) might be lost.

*Default = None*

## ImplicitObjectTypeSuffix

For each implicit object in your model, generated C++ code includes a class that the object instantiates. Generated C code contains a struct that is the type of the implicit object. To differentiate the class/struct from the object, the name of the class/struct is composed of the name of the object plus a suffix. You can use the property ImplicitObjectTypeSuffix to customize the suffix that is used.

*Default = \_C*

## ShowCgSimplifiedModelPackage

The first step of the code generation process consists of the building of a simplified model based on the Rational Rhapsody model.

By default, the simplified model is not displayed in Rational Rhapsody. To have the simplified model displayed in the browser, set the ShowCgSimplifiedModelPackage property property to True. Once you have done so, the next time you generate code, the simplified model is added automatically at the top of the project tree in the browser.

*Default = Cleared*

## **SimplifyMainFiles**

If you are using the Rational Rhapsody customizable code generation mechanism, the SimplifyMainFiles property can be used to change the way main files are handled by Rational Rhapsody when it transforms the model into a simplified model. This allows you to customize code generation for main files, beyond the initialization code you can specify in Rational Rhapsody at the configuration level.

The property can take any of the following values:

- None - Main files are ignored.
- Copy - Main fileare copied from the original to the simplified model. They are not modified in any way.
- Default - Uses the standard simplification for main files, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user.
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for main files has been applied.

*Default = "Default"*

## **SimplifyMakeFile**

If you are using the Rational Rhapsody customizable code generation mechanism, the SimplifyMakeFile property can be used to change the way makefiles are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - Makefile elements are ignored.
- Copy - Makefile elementare copied from the original to the simplified model. They are not modified in any way.
- Default - Uses the standard simplification for makefiles, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user.
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for makefiles has been applied.

*Default = "Default"*

## SourceListFile

*The SourceListFile property specifies the name of the file containing a list of .java source files to be compiled with javac. The batch file used by the Build command (jdkmake.bat) can use the following call, rather than including a long list of source files: javac -g @files.lst This same command is generated from the following line in the MakeFileContent property for Java: javac -g @\$SourceListFile If the SourceListFile property is empty, \$SourceListFile is replaced with a string containing all source file names, separated by spaces (for example, "A.java B.java"). This means that if the MakeFileContent default value is not changed, you get: javac -g @A.java B.java ... If you do not want to use the file containing the list of sources, you must also change the MakeFileContent property to replace "javac -g @\$SourceListFile" with "javac -g \$SourceListFile". Default = files.lst*

## SpecificationEpilog

The SpecificationEpilog property allows you to add code to the end of the declaration of a model element.

For example, you can use this property to add an #ifdef to indicate that an operation is available only when the code is compiled with \_DEBUG, by setting the following properties:

- For SpecificationProlog, enter the value #ifdef \_DEBUG and a new line.
- For SpecificationEpilog, enter the value #endif
- For ImplementationProlog, enter the value #ifdef \_DEBUG and a new line.
- For ImplementationEpilog, enter the value #endif

*Default = Blank*

## SpecificationProlog

The SpecificationProlog property allows you to add code to the beginning of the declaration of a model element.

For example, you can use this property to add an #ifdef to indicate that an operation is available only when the code is compiled with \_DEBUG, by setting the following properties:

- For SpecificationProlog, enter the value #ifdef \_DEBUG and a new line.
- For SpecificationEpilog, enter the value #endif
- For ImplementationProlog, enter the value #ifdef \_DEBUG and a new line.
- For ImplementationEpilog, enter the value #endif

*Default = Blank*

## UsePre80DdsCG

Prior to version 8.0, dependencies with the <<qualityOfService>> stereotype were drawn from the quality of service entity to the DDS entity. Beginning with release 8.0, dependencies with the <<qualityOfService>> stereotype are drawn from the DDS entity (such as DataWriter) to the quality of service entity.

To allow correct code generation for existing models that use the pre-8.0 approach for such dependencies, the `CPP_CG::Configuration::UsePre80DdsCG` property was added to the backward compatibility settings for C++ with a value of `True`.

Note, however, that even with this property, the correct code will be generated only if you turn off the "load on demand" and "parallel code generation" options. If you want to use one of these options, you must reverse the direction of the `<<qualityOfService>>` dependencies in your older model.

To have Rational Rhapsody make the necessary model changes automatically, select Convert pre-8.0 DDS model from the pop-up menu that is displayed for DDS models.

## UsePre81GlobalAttributeLocation

Prior to release 8.1 of Rhapsody, there were rare situations where attributes would not be generated in the correct order. This issue was fixed in release 8.1. To preserve the previous code generation behavior for pre-8.1 models, the property `UsePre81GlobalAttributeLocation` was added to the 8.1 backward compatibility settings for C++ with a value of `True`.

## Cygwin

The Cygwin metaclass controls the environment settings (Compiler, framework libraries, and so on) for Cygwin.

## AdaptorSearchPath

The `AdaptorSearchPath` property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the `AdaptorSearchPath` property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)/LangCpp/osconfig/Cygwin*

## AdditionalReservedWords

The `AdditionalReservedWords` property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use. In general, this is an environment property that can be

contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = Empty string*

## **AnimIncludeDirectories**

The AnimIncludeDirectories property is used to specify the directories that must be referenced in the makefile for projects where Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with INST\_INCLUDES.

*Default = \$(INCLUDE\_QUALIFIER)\$(OMROOT)/LangCpp/aom  
\$(INCLUDE\_QUALIFIER)\$(OMROOT)/LangCpp/tom*

## **AnimInstLibs**

The AnimInstLibs property is used to specify the static libraries that are required when Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with INST\_LIBS.

*Default = \$(OMROOT)/LangCpp/lib/cygwinanim\$(CPU)\$(LIB\_EXT)  
\$(OMROOT)/LangCpp/lib/cygwinomcomanim\$(CPU)\$(LIB\_EXT)*

## **AnimOxfLibs**

The AnimOxfLibs property is used to specify the framework libraries that are required when Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with OXF\_LIBS.

*Default = \$(OMROOT)/LangCpp/lib/cygwinomcomanim\$(CPU)\$(LIB\_EXT)  
\$(OMROOT)/LangCpp/lib/cygwinomcomanim\$(CPU)\$(LIB\_EXT)*

## **AnimPreprocessor**

The AnimPreprocessor property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with INST\_FLAGS.

*Default = \$(DEFINE\_QUALIFIER)OMANIMATOR \$(DEFINE\_QUALIFIER)\_\_USE\_W32\_SOCKETS*

## **BuildArgumentsInIDE**

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the CPP\_CG::[environment]::BuildCommandInIDE property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = \$makefile*

## **BuildCommandInIDE**

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = cygwinmake.bat*

## **BuildCommandSet**

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration.

To change this property, use the Configuration window in the browser - do not change it by using the Properties tab in the Features window or by modifying the site.prp file.

Note that this property also affects the names of the framework libraries used in the link. The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*Default = Debug*

## **buildFrameworkCommand**

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings

tab of the Features window for the active configuration. The `buildFrameworkCommand` property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\"\\etc\\cygwinmake.bat cygwinbuild.mak build \\\"BUILD\_SET=\$BuildCommandSet\\\" \\\"CPU=\$CPU\\\"\""*

## BuildInIDE

The `BuildInIDE` property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Checked*

## CompileCommand

The `CompileCommand` property sets the Cygwin compiler name.

*Default = g++*

## CompilerFlags

The `CompilerFlags` property allows you to define additional compilation flags. The value of the property is inserted into the value of the `CompileSwitches` property (Linux) or `CPPCompileSwitches` (cygwin). In the generated makefile, you can see the value of this property in the line that begins with `ConfigurationCPPCompileSwitches=`.

*Default = Blank*

## CPPCompileCommand

The `CPPCompileCommand` property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The default is as follows:

```
@echo Compiling $OMFileImpPath @$ (CC) $OMFileCPPCompileSwitches -o $OMFileObjPath  
$OMFileImpPath
```

## CPPCompileDebug

The `CPPCompileDebug` property modifies the makefile compile command with switches for building a

debug version of the component.

*Default = -g*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*Default = -O*

## CPPCompileSwitches

The CPPCompileSwitches property specifies the compiler switches.

*Default =*

*\$IncludeDirectories \$DefinedSymbols \$(INST\_FLAGS) \$(INCLUDE\_PATH) \$(INST\_INCLUDES)  
\$CompilerFlags \$OMCPPCompileCommandSet -c*

## CPU

The CPU property is used to specify the CPU of the target environment.

When building applications or framework libraries for MSVC9 environments, Rhapsody uses a variable called \$CPU which represents the value provided for the CPU property for these environments. This variable is provided as a parameter to the msvc9make.bat batch file, and is also used in the makefiles that are generated.

When building 32-bit applications, use the value x86 for this property.

When building 64-bit applications, use the value x64 for this property.

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

*Default = \$OMFileObjPath : \$OMFileImpPath \$OMFileSpecPath \$OMFileDependencies*

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: *\$OMFileObjPath : \$OMFileImpPath "\*.bmp" "\*.ico" "\*.rc2"*

## EnableDebugIntegrationWithIDE



When you use Rational Rhapsody in conjunction an IDE such as Eclipse, you can use the `EnableDebugIntegrationWithIDE` property to specify whether or not the IDE debugger is to be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to `Checked`, the IDE debugger is used.

*Default = Checked*

## EndApplicationCode

When the main function ends, the system's `exit()` function is called automatically. The system function calls the destructors of framework instances in the order it determines. The `EndApplicationCode` property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an `endApplication()` function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to `endApplication()`, it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The `EntryPoint` property specifies the name of the main program for a given environment.

*Default = main*

See also the definition of the `EntryPointDeclarationModifier` property for more information.

## ErrorMessageTokensFormat

The `ErrorMessageTokensFormat`, working with the `ParseErrorMessage` property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

`ErrorMessageTokensFormat` defines the number and location of tokens within the regular expression defined by the `ParseErrorMessage` property. `ErrorMessageTokens` has three parameters, each with an integer value:

- `TotalNumberOfTokens` - The number of tokens in the regular expression
- `FileTokenPosition` - The position of the file name token in the expression
- `LineTokenPosition` - The position of the line number token in the expression

*Default = TotalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2*

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the CPP\_CG::<Environment>::ExeName property plus the value of this property.

*(Default = .exe)*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the CPP\_CG::<Environment>::ExeExtension property.

*(Default = Blank)*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements. The file inclusions are generated in the makefile.

*Default = \$OMSpecIncludeInElements \$OMImpIncludeInElements*

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled).

During code generation, these libraries are added to the generated makefile.

Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

*Default = \$(OMROOT)\LangCpp\lib\cygwinWebComponents\$(CPU)\$(LIB\_EXT),  
\$(OMROOT)\lib\cygwinWebServices\$(CPU)\$(LIB\_EXT), -lws2\_32*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated

implementation files for a given language and environment.

*Default = .cpp*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*Default = include*

## InvokeAnimatedExecutable

The AnimationPortRange value in the rhapsody.ini file makes it possible to use animation with multiple instances of Rhapsody running simultaneously.

Ordinarily, when you run an application with animation, the executable is launched using the command specified by the InvokeExecutable property. However, if you have specified a value greater than zero for AnimationPortRange in your rhapsody.ini file, the executable is launched using the command specified by the InvokeAnimatedExecutable property. By default, the value of InvokeAnimatedExecutable includes a reference to the port to use for animation.

*Default = "\"\$OMROOT/etc/Executer.exe\" \"\"\$OMROOT\|\"|etc\cygwinrun.bat \$executable -port \$port\"\""*

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT/etc/cygwinrun.bat\" \$executable"*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

*"rhapsody\_dir>\share/etc/Executer.exe" "dir; dir | more"*

The InvokeMake property might include the value of any other property in the relevant environment.

To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

*"\$OMROOT/etc/Executer.exe" "\"\$OMROOT/etc/vx6make.bat\" \$makefile \$maketarget 6.2 \$BSP gnu"*

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\|\"|etc/cygwinmake.bat \$makefile \$maketarget"*

`\ "CPU=$CPU\" \" "`

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .a*

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

*Default = -g*

## LinkerFlags

The LinkerFlags property allows you to define linker flags. The value of the property is inserted into the value of the LinkSwitches property. In the generated makefile, you can see the value of this property in the line that begins with LINK\_FLAGS=.

*Default = Blank*

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

*Default = -O*

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

*Default = \$OMLinkCommandSet \$LinkerFlags*

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the CPP\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*(Default = .mak)*

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration.

The default is as follows: ##### Target type (Debug/Release) #####  
#####  
CPPCompileDebug=\$OMCPPCompileDebug CPPCompileRelease=\$OMCPPCompileRelease  
LinkDebug=\$OMLinkDebug LinkRelease=\$OMLinkRelease BuildSet=\$OMBuildSet  
SUBSYSTEM=\$OMSubSystem COM=\$OMCOM RPFrameWorkDll=\$OMRPFrameWorkDll  
ConfigurationCPPCompileSwitches= \$OMReusableStatechartSwitches  
\$OMConfigurationCPPCompileSwitches !IF "\$(RPFrameWorkDll)" == "True"  
ConfigurationCPPCompileSwitches= \$(ConfigurationCPPCompileSwitches) /D "FRAMEWORK\_DLL"  
!ENDIF !IF "\$(COM)" == "True" SUBSYSTEM=/SUBSYSTEM:windows !ENDIF

**Compilation Flags** The compilation flags section of the makefile contains the default compilation flags stored in the CompileSwitches property. For example, the default content of the compilation flags section of a C++ makefile for the Microsoft environment is as follows: ##### Compilation flags  
#####  
INCLUDE\_QUALIFIER=/I LIB\_PREFIX=MS

Commands Definitions The commands definition section of the makefile specifies programs to execute from the makefile. For example, the default commands definition section of a C++ makefile for the Microsoft environment is as follows: ##### Commands definition #####  
##### RMDIR = rmdir LIB\_CMD=link.exe -lib  
LINK\_CMD=link.exe LIB\_FLAGS=\$OMConfigurationLinkSwitches  
LINK\_FLAGS=\$OMConfigurationLinkSwitches \$(SUBSYSTEM) / MACHINE:I386

Generated Macros The generated macros section of the makefile contains a variable that expands to the Rational Rhapsody-generated macros in the makefile. For example: ##### Generated macros #####  
##### \$OMContextMacros  
OBJ\_DIR=\$OMObjectsDir !IF "\$(OBJ\_DIR)"!=" CREATE\_OBJ\_DIR=if not exist \$(OBJ\_DIR) mkdir  
\$(OBJ\_DIR) CLEAN\_OBJ\_DIR= if exist \$(OBJ\_DIR) \$(RMDIR) \$(OBJ\_DIR) !ELSE  
CREATE\_OBJ\_DIR= CLEAN\_OBJ\_DIR= !ENDIF

The \$OMContextMacros keyword expands several macros in the makefile. Each makefile macro has its own keyword. You can use these keywords separately to customize the makefile.

The \$OMContextMacros variable modifies target-specific variables. Replace the \$OMContextMacros line in the MakeFileContent property with the following:

```
FLAGSFILE=$OMFlagsFile RULESFILE=$OMRulesFile OMROOT=$OMROOT
CPP_EXT=$OMImplExt H_EXT=$OMSpecExt OBJ_EXT=$OMObjExt EXE_EXT=$OMExeExt
LIB_EXT=$OMLibExt INSTRUMENTATION=$OMInstrumentation TIME_MODEL=$OMTimeModel
TARGET_TYPE=$OMTargetType TARGET_NAME=$OMTargetName $OMAllDependencyRule
TARGET_MAIN=$OMTargetMain LIBS=$OMLibs INCLUDE_PATH=$OMIncludePath
ADDITIONAL_OBJS=$OMAdditionalObjs OBJS= $OMObjs
```

Predefined Macros The predefined macros section of the makefile contains other macros than the Rational Rhapsody-generated macros specified in the generated macros section. For example, the default predefined macros section of a C++ makefile for the Microsoft environment is as follows:

```
##### Predefined macros #####
##### $(OBJS) : $(INST_LIBS)
$(OXF_LIBS) LIB_POSTFIX= !IF "$(BuildSet)"=="Release" LIB_POSTFIX=R !ENDIF !IF
"$(TARGET_TYPE)" == "Executable" LinkDebug=$(LinkDebug) /DEBUG
LinkRelease=$(LinkRelease) /OPT:NOREF !ELSEIF "$(TARGET_TYPE)" == "Library"
LinkDebug=$(LinkDebug) /DEBUGTYPE:CV !ENDIF !IF "$(INSTRUMENTATION)" == "Animation"
INST_FLAGS=/D "OMANIMATOR" INST_INCLUDES=/I $(OMROOT)\LangCpp\aoom /I
$(OMROOT)\LangCpp\tom !IF "$(RPFrameWorkDll)" == "True" INST_LIBS=
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfanimdll$(LIB_POSTFIX) $(LIB_EXT)
!ELSE INST_LIBS= $(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomanim$(LIB_POSTFIX) $(LIB_EXT)
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfinst$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)" == "Tracing" INST_FLAGS=/D
"OMTRACER" INST_INCLUDES=/I $(OMROOT)\LangCpp\aoom /I $(OMROOT)\LangCpp\tom !IF
"$(RPFrameWorkDll)" == "True" INST_LIBS= OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfracedll$(LIB_POSTFIX) $(LIB_EXT) !ELSE
INST_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)tomtrace$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomtrace$(LIB_POSTFIX) $(LIB_EXT) OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfinst$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)" == "None" INST_FLAGS=
INST_INCLUDES= INST_LIBS= !IF "$(RPFrameWorkDll)" == "True"
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfdll$(LIB_POSTFIX) $(LIB_EXT) !ELSE
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
```

```
SOCK_LIB= !ELSE !ERROR An invalid Instrumentation $(INSTRUMENTATION) is specified.
!ENDIF
```

**Generated Dependencies** The generated dependencies section of the makefile contains a variable that expands to Rational Rhapsody-generated dependencies and compilation instructions. For example, the generated dependencies section of a C++ makefile for the Microsoft environment is as follows:

```
##### Generated dependencies #####
##### SOMContextDependencies
$OMFileObjPath : $OMMainImplementationFile $(OBJS) $(CPP) $(ConfigurationCPPCompileSwitches)
/Fo"$OMFileObjPath" $OMMainImplementationFile
```

**Linking Instructions** The linking instructions section of the makefile contains the predefined linking instructions. For example, the default linking instructions section of a C++ makefile for the Microsoft environment is as follows:

```
##### Linking instructions #####
#####
$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS) $OMFileObjPath
$OMMakefileName $OMModelLibs @echo Linking $(TARGET_NAME)$(EXE_EXT) $(LINK_CMD)
$OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \ $(OXF_LIBS) \
$(SOCK_LIB) \ $(LINK_FLAGS) /out:$(TARGET_NAME)$(EXE_EXT)
$(TARGET_NAME)$(LIB_EXT) : $(OBJS) $(ADDITIONAL_OBJS) $OMMakefileName @echo
Building library $@ $(LIB_CMD) $(LIB_FLAGS) /out:$(TARGET_NAME)$(LIB_EXT) $(OBJS)
$(ADDITIONAL_OBJS) clean: @echo Cleanup $OMCleanOBJS if exist $OMFileObjPath erase
$OMFileObjPath if exist *$(OBJ_EXT) erase *$(OBJ_EXT) if exist $(TARGET_NAME).pdb erase
$(TARGET_NAME).pdb if exist $(TARGET_NAME)$(LIB_EXT) erase
$(TARGET_NAME)$(LIB_EXT) if exist $(TARGET_NAME).ilk erase $(TARGET_NAME).ilk if exist
$(TARGET_NAME)$(EXE_EXT) erase $(TARGET_NAME)$(EXE_EXT) $(CLEAN_OBJ_DIR)
```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the CPP\_CG::<Environment>::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## NoneIncludeDirectories

The NoneIncludeDirectories property is used to specify the directories that must be referenced in the makefile for projects where Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with INST\_INCLUDES.

*Default = Blank*

## NoneInstLibs

The `NoneInstLibs` property is used to specify the static libraries that are required when Instrumentation Mode is set to `None`. In the makefile, these appear in the line that begins with `INST_LIBS`.

*Default = `$(OMROOT)/LangCpp/lib/cygwinosim$(CPU)$(LIB_EXT)`*

## **NoneOxfLibs**

The `NoneOxfLibs` property is used to specify the framework libraries that are required when Instrumentation Mode is set to `None`. In the makefile, these appear in the line that begins with `OXF_LIBS`.

*Default = `$(OMROOT)/LangCpp/lib/cygwinoxf$(CPU)$(LIB_EXT)`*

## **NonePreprocessor**

The `NonePreprocessor` property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to `None`. In the makefile, these appear in the line that begins with `INST_FLAGS`.

*Default = `Blank`*

## **ObjCleanCommand**

The `ObjCleanCommand` property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default = `$(RM) $OMFileObjPath`*

## **ObjectName**

The `ObjectName` property specifies an alternative name for the compiled object file in the generated makefile.

*Default = `Empty string`*

## **ObjectsDirectory**

The `ObjectsDirectory` property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = `Empty string`*

## **ObjExtension**

The `ObjExtension` property specifies the extension appended to compiled object files for a given environment.



*Default = .o*

## OSFileSystemCaseSensitive

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Checked*

## ParseErrorDescript

The ParseErrorDescript property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = ([^:]+)[:]( [0-9:]+)[:](error/warning)[:](.\*)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = ([^:]+)[:]( [0-9:]+)[:]*

## ParseErrorMoreInfo

The ParseErrorMoreInfo property is used to define a regular expression that represents the format of compiler error messages that follow the first line which contains the term 'error' or 'warning'.

This property is used to parse the compiler output and extract the additional information from the message so that this information can be displayed to the user in the build output window.

*Default = Blank*

## ParseMakeError

The ParseMakeError property is used to define a regular expression that represents the format of make process or linker error messages. This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = (make)[:](.\*) (Error)*

## ParseSeverityError

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = [:] (error/undefined/cannot find/multiple definition)*

## ParseSeverityWarning

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^:]+):([0-9:]+):[:] (warning)*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation.

*Default = \*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Checked*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running the product, whereas the target is the machine running the application. To run remotely, the UseRemoteHost property must be set to True.

If UseRemoteHost is Checked and RemoteHost is blank, the current host name is used for the remote host. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = Empty string*

## ReusableStatechartSwitches

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts. See the upgrade history on the support site for information on this change.

*Default = -DOM\_REUSABLE\_STATECHART\_IMPLEMENTATION*

## **SockLib**

The SockLib property represents the name of the socket library.

For the Cygwin environment, the value entered for this property is used in the generated makefile.

When using the integration with Visual Studio, the value of this property is included in the "additional dependencies" for the Visual Studio project.

*Default = -lws2\_32*

## **SpecExtension**

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*Default = .h*

## **TraceIncludeDirectories**

The TraceIncludeDirectories property is used to specify the directories that must be referenced in the makefile for projects where Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with INST\_INCLUDES.

*Default = \$(INCLUDE\_QUALIFIER)\$(OMROOT)/LangCpp/aom  
\$(INCLUDE\_QUALIFIER)\$(OMROOT)/LangCpp/tom*

## **TraceInstLibs**

The TraceInstLibs property is used to specify the static libraries that are required when Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with INST\_LIBS.

*Default = \$(OMROOT)/LangCpp/lib/cygwintomtrace\$(CPU)\$(LIB\_EXT)  
\$(OMROOT)/LangCpp/lib/cygwinaomtrace\$(CPU)\$(LIB\_EXT)  
\$(OMROOT)/LangCpp/lib/cygwinomsiminst\$(CPU)\$(LIB\_EXT)*

## **TraceOxfLibs**

The TraceOxfLibs property is used to specify the framework libraries that are required when Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with OXF\_LIBS.

*Default = \$(OMROOT)/LangCpp/lib/cygwinomxfinst\$(CPU)\$(LIB\_EXT)  
\$(OMROOT)/LangCpp/lib/cygwinomcomappl\$(CPU)\$(LIB\_EXT)*

## TracePreprocessor

The TracePreprocessor property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with INST\_FLAGS.

*Default = \$(DEFINE\_QUALIFIER)OMTRACER*

## UnixLineTerminationStyle

The UnixLineTerminationStyle property specifies whether generated files use the UNIX end-of-line style. If this property is set to Cleared, the end-of-line style depends on the host type (for example, DOS style on Windows machines, and UNIX style on Solaris machines).

*Default = Cleared*

## UnixPathNameForOMROOT

The UnixPathNameForOMROOT property specifies whether the makefile must include UNIXstyle path names.

The pRISM compilers do not tolerate DOS-style path conventions. If you do not set this property correctly, there might be many compilation problems.

*Default = Cleared*

## UpdateBuildSettingsInIDE

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## UseNewBuildOutputWindow

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the CG::General::ShowLogViewAfterBuild property to True.

*Default = Checked*

## UseRemoteHost

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Checked*

## UserVariables

The UserVariables property allows you to add user variables to be set in Eclipse integrated projects.

*Default = CYGWIN=nodosfilewarning*

## UseTemplateName

In the declarations for members of a template class, some compilers require that you use precede the member type with the keyword "typename" for types with dependent names. Since support for this keyword varies between compilers, the UseTemplateName property is used to specify whether the "typename" keyword should be included in the generated code.

*Default = Checked*

## DDS

Contains properties relating to code for DDS models.

### Pre803MultiplicityWarning

Beginning in version 8.0.3, for elements with bounded multiplicity in DDS models, Rational Rhapsody generates arrays by default, rather than sequences.

If you want to use the new code generation behavior for an existing model, but do not want to see a warning about the change each time you generate code, change the value of the CPP\_CG::DDS::Pre803MultiplicityWarning property to False.

## Dependency

The Dependency metaclass controls the dependency for a package that defines a namespace.

## CreateUseStatement

The CreateUseStatement property determines whether a use statement is added to the code after the with statement. The supplier of the dependency must be a class or type.

*Default = False*

## GenerateForwardDeclarations

The GenerateForwardDeclarations property is a Boolean property that specifies whether forward declarations are generated.

When set to True, the generation of forward declarations is carried out according to the value of the property CG::Dependency::UsageType.

When set to False, the specification file will not contain a forward declaration even if the value of UsageType is set to Existence or Implementation.

*Default = True*

## GenerateOriginComment

When set to Checked, generates a comment before #include statements that indicate which element "caused" the #include.

*Default = Checked*

## GeneratePragmaElaborate

The GeneratePragmaElaborate property determines whether to generate an elaborate pragma for the supplier class in the client class or package.

*Default = False*

## GeneratePragmaElaborateAll

The GeneratePragmaElaborateAll property determines whether to generate a pre-elaborate pragma for the supplier class in the client class or package.

*Default = False*

## GenerateWithClause

The GenerateWithClause property determines whether with clauses are generated for Usage dependencies.

For example, you can generate a with clause for a package, P1, in the specification of another package, P2, by using a dependency, D1, and generate a use clause for P1 in the body of P2. In addition, this functionality is useful for modeling inherited annotations across classes and packages.

*Default = True*

## ImplementationEpilog

The ImplementationEpilog property enables you to add any code that you want to be added as verbatim text (to be ignored by Rational Rhapsody) to the end of the definition of a model element.

For example, you could wrap a section of code with an #ifdef-#endif pair, add compiler-specific keywords, or add a #pragma statement. For example, to specify that an operation is available only when the code is compiled with \_DEBUG, set the following properties for the operation:

- Set SpecificationProlog to #ifdef \_DEBUG cr.
- Set SpecificationEpilog to #endif.
- Set ImplementationProlog to #ifdef \_DEBUG cr.
- Set ImplementationEpilog to #endif.

The following table lists whether leading and trailing linefeeds are generated.

Metaclass	Leading Linefeed Added?	Class	Yes	Package	No
-----------	-------------------------	-------	-----	---------	----

*Default = Empty MultiLine*

## ImplementationProlog

The ImplementationProlog property enables you to add any code that you want to be added as verbatim text (to be ignored by the software) to the beginning of the definition of a model element.

For example, you could wrap a section of code with an #ifdef-#endif pair, add compiler-specific keywords, or add a #pragma statement. For example, to specify that an operation is available only when the code is compiled with \_DEBUG, set the following properties for the operation:

- Set SpecificationProlog to #ifdef \_DEBUG cr.
- Set SpecificationEpilog to #endif.
- Set ImplementationProlog to #ifdef \_DEBUG cr.
- Set ImplementationEpilog to #endif.

The following table lists whether leading and trailing linefeeds are generated.

Metaclass	Trailing Linefeed Added?	Class	No	Package	Yes
-----------	--------------------------	-------	----	---------	-----

*Default = Empty MultiLine*

## IncludeStyle

The IncludeStyle property controls the style of #include statements. When you use this property, you can control the style of a specific dependency, or the entire configuration/component/project.

To set the style for include files that are synthesized based on associations between model elements (for example, setting the type of some attribute to a class), add a «Usage» dependency between the elements and set this property to the appropriate value. The possible values are as follows:

- Default - Use angle brackets for include statements for external elements, and quotes for include statements for other elements.
- Quotes - Enclose include files in quotation marks. For example: #include "A.h"
- When a compiler encounters an include file in quotes, it searches for the file in both the current directory and the directories specified in the include path. Note that the specific algorithm used is compiler-dependent.
- AngledBrackets - Enclose include files in angle brackets. For example: #include A.h
- When a compiler encounters an include file in angle brackets, it searches for the file only in the directories specified in the include path.
- If you set the property to AngledBrackets at the configuration level, you must also change the CG::File::IncludeScheme property to RelativeToConfiguration to ensure successful compilation.

*Default = Default*

## MarkPrologEpilogInAnnotations

The MarkPrologEpilogInAnnotations property specifies whether to generate ignore annotations for the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties so they are ignored during roundtrip.

When you insert code element declarations (variables, types, functions, and so on) in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, after a full roundtrip those elements are added to the model and are duplicated on the next code generation.

When you use the MarkPrologEpilogInAnnotations property, you can have Rational Rhapsody automatically ignore the information specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties instead of adding the ignore annotations manually.

The possible values for the MarkPrologEpilogInAnnotations property are:

- None - Rational Rhapsody does not generate any annotations. Any models created before Version 4.1 automatically have this property setting.
- Ignore - Rational Rhapsody generates the `/// ignore annotation before the code specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, and generates the /// annotation after the code specified in those properties.`
- Auto - If the code in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties is one line (it does not contain any newline characters (`\n`)), no annotations are generated (the same behavior has the None setting). If there is more than one line, Rational Rhapsody generates the `/// ignore annotation before the code specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, and generates the /// annotation after the code specified in those properties (the same behavior as the Ignore setting).`



During roundtrip, any ignore annotations in the comments of the element are not included in its description. Because the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties are generated between the annotation and its declaration for the element, you cannot rename those elements on roundtrip. If you change the name of an element, it is removed from the model and added with the new name.

Some model information (for example, property settings) might be lost.

*Default = None*

## NamespaceAlias

The NamespaceAlias property allows you to take advantage of the C++ namespace alias feature.

The value of the property should be the string you would like to use as the alias for the namespace of the package that the element is dependent upon.

For example, if you have specified a dependency on the nested namespace Hardware, as defined below:

```
namespace Equipment
```

```
{
```

```
namespace Hardware
```

```
{
```

```
class Printer { ... };
```

```
}
```

```
}
```

you can enter hw for the value of the NamespaceAlias property, and then the generated code includes the following statement:

```
namespace hw = Equipment::Hardware;
```

Because namespace aliases are an alternative to the use of "using" directives, Rational Rhapsody ignores the value of the property UseNameSpace if you have entered a value for the NamespaceAlias property for the same dependency.

*Default = Blank*

## Simplify

If you are using the Rational Rhapsody customizable code generation mechanism, the Simplify property can be used to change the way specific types of elements are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - The element is ignored.
- Copy - The element is copied from the original to the simplified model. It is not modified in any way.
- Default - Uses the standard simplification for this item, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for the element has been applied.

*Default = "Default"*

## SpecificationEpilog

The SpecificationEpilog property allows you to add code to the end of the declaration of a model element.

For example, you can use this property to add an `#ifdef` to indicate that an operation is available only when the code is compiled with `_DEBUG`, by setting the following properties:

- For SpecificationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For SpecificationEpilog, enter the value `#endif`
- For ImplementationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For ImplementationEpilog, enter the value `#endif`

*Default = Blank*

## SpecificationProlog

The SpecificationProlog property allows you to add code to the beginning of the declaration of a model element.

For example, you can use this property to add an `#ifdef` to indicate that an operation is available only when the code is compiled with `_DEBUG`, by setting the following properties:

- For SpecificationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For SpecificationEpilog, enter the value `#endif`
- For ImplementationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For ImplementationEpilog, enter the value `#endif`

*Default = Blank*

## UseNameSpace

*The UseNameSpace property models namespace usage. When you set a dependency to a package that defines a namespace and set this property to Checked, Rational Rhapsody generates a "using namespace" statement to the package namespace. Default = Cleared*

# EnumerationLiteral

The EnumerationLiteral metaclass contains properties related to the generation of code for enumerations.

## DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- `$Name` - the element name
- `$FullName` - the full path of the element (P1::P2::C.a)
- `$Description` - the text provided on the Description tab of the Features window for the element
- `$Requirements` - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the `$Requirements` keyword in the DescriptionTemplate property to get such comments generated for specific model elements.
- `$Login` - the login name of the user who generated the file
- `$CodeGeneratedDate` - the date on which the code was generated
- `$CodeGeneratedTime` - the time at which the code was generated
- `$RhapsodyVersion` - the version of Rhapsody that was used to generate the file
- For operations: `$Signature` - the operation signature, `$Arguments` - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: `$Type` - the type of the attribute
- For arguments: `$Type` - the argument type, `$Direction` - the argument direction (In, Out, or InOut)
- For events: `$Arguments` - the text provided on the Description tab of the Features window for the event arguments
- For association ends: `$Target` - the other end of the association
- For requirements: `$ID` - the ID specified for the requirement, `$Specification` - the specification text for the requirement
- The values of tags can be included by using the syntax `$tag_name`. For example, if you use a tag called

author for your classes, you can use the keyword \$author in the value of DescriptionTemplate in order to have the value of the author tag included in the element description.

- The values of properties can be included by using the syntax \$property\_name. For example, you can use the keyword \$Animate in the value of DescriptionTemplate in order to have the value of the Animate property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as \$Name)
- Tag keywords
- Property keywords

If the value of a keyword is a MultiLine value, then each new line (except the first one) starts with the value of the <lang>\_CG::Configuration::DescriptionBeginLine property, and each line ends with the value of the <lang>\_CG::Configuration::DescriptionEndLine property.

*Default = Blank*

## Event

The Event metaclass contains properties that control events.

### AnimInstanceCreate

The AnimInstanceCreate property affects event creation. If you set the C\_CG::Event::NoDynamicAllocAnimCreate property to False, Rational Rhapsody does not generate the event creation method, effectively disabling the ability to inject the event in animation. To enable the injection of the event, you can specify a different method to obtain an instance of the event by setting this property to the name of the method to use.

*Default = Empty string*

### DeclarationModifier

The DeclarationModifier property adds a string to the class or event declaration. The string is displayed between the class keyword and the class name in the generated code. For example, for a class A, the DeclarationModifier would appear as follows: class DeclarationModifier> A {...}; This property adds a modifier to the class declaration. For example, if you have a class myExportableClass that is exported from a DLL by using the MYDLL\_API macro, you can set the DeclarationModifier property to “MYDLL\_API.” The generated code would then be as follows: class MYDLL\_API myExportableClass {...}; This property supports two keywords: \$component and \$class.

*Default = Empty string*

## DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- `$Name` - the element name
- `$FullName` - the full path of the element (`P1::P2::C.a`)
- `$Description` - the text provided on the Description tab of the Features window for the element
- `$Requirements` - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the `$Requirements` keyword in the DescriptionTemplate property to get such comments generated for specific model elements.
- `$Login` - the login name of the user who generated the file
- `$CodeGeneratedDate` - the date on which the code was generated
- `$CodeGeneratedTime` - the time at which the code was generated
- `$RhapsodyVersion` - the version of Rhapsody that was used to generate the file
- For operations: `$Signature` - the operation signature, `$Arguments` - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: `$Type` - the type of the attribute
- For arguments: `$Type` - the argument type, `$Direction` - the argument direction (In, Out, or InOut)
- For events: `$Arguments` - the text provided on the Description tab of the Features window for the event arguments
- For association ends: `$Target` - the other end of the association
- For requirements: `$ID` - the ID specified for the requirement, `$Specification` - the specification text for the requirement
- The values of tags can be included by using the syntax `$tag_name`. For example, if you use a tag called `author` for your classes, you can use the keyword `$author` in the value of DescriptionTemplate in order to have the value of the `author` tag included in the element description.
- The values of properties can be included by using the syntax `$property_name`. For example, you can use the keyword `$Animate` in the value of DescriptionTemplate in order to have the value of the `Animate` property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as \$Name)
- Tag keywords
- Property keywords

If the value of a keyword is a MultiLine value, then each new line (except the first one) starts with the value of the `<lang>_CG::Configuration::DescriptionBeginLine` property, and each line ends with the value of the `<lang>_CG::Configuration::DescriptionEndLine` property.

*Default = Blank*

## EnableDynamicAllocation

The EnableDynamicAllocation property specifies whether to use dynamic memory allocation for objects. The possible values are as follows:

- Checked - Dynamic allocation of events is enabled. Create() and Destroy() operations are generated for the object or object type.
- Cleared - Events are dynamically allocated during initialization, but not during run time. Create() and Destroy() operations are not generated for the object. This setting is recommended for static architectures that do not use dynamic memory management during run time.

If you are managing your own memory pools, set this property to False and call CPPReactive\_gen() directly. The following example shows how to call RiCReactive\_gen() directly to send a static event to a reactive object A, when you use a member function of A genStaticEv2A():

```
void A_genStaticEv2A(struct A_t* const me) { { /*#[ operation genStaticEv2A() */ static struct ev _ev;  
ev_Init(_ev); RiCEvent_setDeleteAfterConsume(((RiCEvent*)_ev), RiCFALSE); (void)  
RiCReactive_gen(me-ric_reactive, ((RiCEvent*)_ev), RiCFALSE); /*#]*/ } }
```

Alternatively, you can use internal memory pools by setting the BaseNumberOfInstances property, which results in the use of framework memory pools. If you use the framework memory pools, do not disable the Create() and Destroy() methods because these methods are used to manage the memory pool.

When you disable the generation of the Create() and Destroy() methods, you can still inject events in animation by supplying an alternate function to get an event instance. To do this, set the AnimInstanceCreate property.

*Default = Checked*

## EventIdAsDefine

When Rational Rhapsody generates code for an event, it creates an ID number for the event. If the value of the EventIdAsDefine property is set to True, the ID is assigned by using a macro definition. If the value of the property is set to False, the ID is assigned by using a variable.

*Default = Checked*

## In

The In property determines the exact syntax used when an event is used as an "in" parameter for an operation.

*Default = const \$type&*

## InitializationStyle

The InitializationStyle property determines whether event arguments are initialized in the initialization list or by assignment in the body of the constructor.

The possible values for the property are ByInitializer and ByAssignment.

The default value of this property was changed in version 7.6. In the backward compatibility settings for C++ in version 7.6, the value of this property is set to ByAssignment in order to maintain the previous behavior for pre-7.6 models.

*Default = ByInitializer*

## InOut

The InOut property determines the exact syntax used when an event is used as an "in/out" parameter for an operation.

*Default = \$type&*

## Out

The Out property determines the exact syntax used when an event is used as an "out" parameter for an operation.

*Default = \$type\*&*

## ReturnType

The ReturnType property determines the exact syntax used when an event is used as the return type of an operation.

*Default = \$type\**

## Simplify

If you are using the Rational Rhapsody customizable code generation mechanism, the Simplify property can be used to change the way specific types of elements are handled by Rational Rhapsody when it

transforms the model into a simplified model.

The property can take any of the following values:

- None - The element is ignored.
- Copy - The element is copied from the original to the simplified model. It is not modified in any way.
- Default - Uses the standard simplification for this item, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user.
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for the element has been applied.

*Default = "Default"*

## File

The File metaclass contains properties that control the generated code files.

### DiffDelimiter

When code is generated, new files are generated only where a before vs. after comparison indicates that the code has changed. When dealing with comments in the code, there are cases where you may decide that a change is not significant enough to justify regeneration of the file. For example, if you record the author of a file as a comment in the file, you may decide that the file should not be regenerated if the only change is the name of the author. The property DiffDelimiter can be used to mark such insignificant changes. When you use the string specified for the DiffDelimiter property somewhere in your code, the text to the right of the delimiter will be ignored when the code comparison is done prior to the regeneration of files.

For example, the default value of the property CPP\_CG::File::SpecificationHeader includes the following text:

```
//! Generated Date: $CodeGeneratedDate
```

So if the only change in the code is the code generation date, the file will not be regenerated.

Note that the delimiter can be used at the beginning of a line (in which case the entire line will be ignored in the comparison) or in the middle of a line (in which case only the text to the right of the delimiter will be ignored).

If you do not want to use this feature in your model, change the value of the property to blank.

*Default = //!*

### Footer



The Footer property specifies a multiline footer that is added to the end of generated Java files.

*Default =*

```
"/***** File Path:
$FullCodeGeneratedFileName *****/"
```

Footer format strings can contain any of the following keywords:

- `$ProjectName` - The project name.
- `$ComponentName` - The component name.
- `$ConfigurationName` - The configuration name.
- `$ModelElementName` - The name of the element mapped to the file. If there is more than one, this is the name of the first element.
- `$FullModelElementName` - The name of the element mapped to the file, including the full path. If there is more than one, this is the name of the first element.
- `$CodeGeneratedDate` - The generation date.
- `$CodeGeneratedTime` - The generation time.
- `$RhapsodyVersion` - The version of Rational Rhapsody that generated the file.
- `$Login` - The user who generated the file.
- `$CodeGeneratedFileName` - The name of the generated file.
- `$FullCodeGeneratedFileName` - The full file name.
- `$Tag` - The value of the specified the element tag.
- `$Property` - The value of the element property with the specified name.

To avoid redundant compilation, Rational Rhapsody avoids unnecessary changes to specific lines prefixed with a special string, defined by the `CPP_CG::File::DiffDelimiter` property. The default `DiffDelimiter` value is `"//!"`.

## Generate

The `Generate` property is used to specify whether code should be generated for a Class or File element. For Java code generation, this is a Boolean property. For C and C++, there are also property values that can be used to generate only the specification file or only the implementation file.

The possible values are:

- `True` - for C and C++ both specification and implementation files are generated
- `False` - no files are generated
- `Specification (C, C++)` - only the specification file is generated
- `Implementation (C, C++)` - only the implementation file is generated

*Default = True*

## Header

The Header property specifies a multiline header that is added to the top of all generated Java files.

*Default =*

```
/****** Rhapsody : $RhapsodyVersion
Login : $Login Component : $ComponentName Configuration : $ConfigurationName Model Element :
$FullModelElementName //! Generated Date : $CodeGeneratedDate File Path :
$FullCodeGeneratedFileName *****/
```

Header format strings can contain any of the following keywords:

- `$ProjectName` - The project name.
- `$ComponentName` - The component name.
- `$ConfigurationName` - The configuration name.
- `$ModelElementName` - The name of the element mapped to the file. If there is more than one, this is the name of the first element.
- `$FullModelElementName` - The name of the element mapped to the file, including the full path. If there is more than one, this is the name of the first element.
- `$CodeGeneratedDate` - The generation date.
- `$CodeGeneratedTime` - The generation time.
- `$RhapsodyVersion` - The version of Rational Rhapsody that generated the file.
- `$Login` - The user who generated the file.
- `$CodeGeneratedFileName` - The name of the generated file.
- `$FullCodeGeneratedFileName` - The full file name.
- `$Tag` - The value of the specified the element tag.
- `$Property` - The value of the element property with the specified name.

To avoid redundant compilation, Rational Rhapsody avoids unnecessary changes to specific lines prefixed with a special string, defined by the `CPP_CG::File::DiffDelimiter` property. The default `DiffDelimiter` value is `"//!"`.

## ImplementationEpilog

The `ImplementationEpilog` property enables you to add any code that you want to be added as verbatim text (to be ignored by Rational Rhapsody) to the end of the definition of a model element. For example, you could wrap a section of code with an `#ifdef-#endif` pair, add compiler-specific keywords, or add a `#pragma` statement. For example, to specify that an operation is available only when the code is compiled with `_DEBUG`, set the following properties for the operation:

- Set `SpecificationProlog` to `#ifdef _DEBUG cr.`
- Set `SpecificationEpilog` to `#endif.`
- Set `ImplementationProlog` to `#ifdef _DEBUG cr.`
- Set `ImplementationEpilog` to `#endif.`

The following table lists whether leading and trailing linefeeds are generated.

Metaclass Leading Linefeed Added? Class Yes Package No

*Default = Empty MultiLine*

## ImplementationFooter

The ImplementationFooter property specifies the multiline footer to be generated at the end of implementation files. The default footer template for C++ is as follows:

```
/****** File Path:
$FullCodeGeneratedFileName *****/
```

Footer format strings can contain any of the following keywords:

- \$ProjectName - The project name.
- \$ComponentName - The component name.
- \$ConfigurationName - The configuration name.
- \$ModelElementName - The name of the element mapped to the file. If there is more than one, this is the name of the first element.
- \$FullModelElementName - The name of the element mapped to the file, including the full path. If there is more than one, this is the name of the first element.
- \$CodeGeneratedDate - The generation date.
- \$CodeGeneratedTime - The generation time.
- \$RhapsodyVersion - The version of Rational Rhapsody that generated the file.
- \$Login - The user who generated the file.
- \$CodeGeneratedFileName - The name of the generated file.
- \$FullCodeGeneratedFileName - The full file name.
- \$Tag - The value of the specified the element tag.
- \$Property - The value of the element property with the specified name.

To avoid redundant compilation, Rational Rhapsody avoids unnecessary changes to specific lines prefixed with a special string, defined by the CPP\_CG::File::DiffDelimiter property. The default DiffDelimiter value is “//!”. The keywords are resolved in the following order:

- Predefined keywords
- Property keywords
- Tag keywords

Note the following:

- Keyword names can be written in parentheses. For example: \$(Name)
- If the value of a keyword is a MultiLine, each new line (except the first one) starts with the value of the CPP\_CG::Configuration::DescriptionBeginLine property; each line ends with the value of the CPP\_CG::Configuration::DescriptionEndLine property.

## ImplementationHeader

The ImplementationHeader property specifies the multiline header that is generated at the beginning of implementation files. The default header template for C++ is as follows:

```
/****** Rhapsody: $RhapsodyVersion
Login: $Login Component: $ComponentName Configuration: $ConfigurationName Model Element:
$FullModelElementName //! Generated Date: $CodeGeneratedDate File Path:
$FullCodeGeneratedFileName *****/
```

Header format strings can contain any of the following keywords:

- \$ProjectName - The project name.
- \$ComponentName - The component name.
- \$ConfigurationName - The configuration name.
- \$ModelElementName - The name of the element mapped to the file. If there is more than one, this is the name of the first element.
- \$FullModelElementName - The name of the element mapped to the file, including the full path. If there is more than one, this is the name of the first element.
- \$CodeGeneratedDate - The generation date.
- \$CodeGeneratedTime - The generation time.
- \$RhapsodyVersion - The version of Rational Rhapsody that generated the file.
- \$Login - The user who generated the file.
- \$CodeGeneratedFileName - The name of the generated file.
- \$FullCodeGeneratedFileName - The full file name.
- \$Tag - The value of the specified the element tag.
- \$Property - The value of the element property with the specified name.

To avoid redundant compilation, Rational Rhapsody avoids unnecessary changes to specific lines prefixed with a special string, defined by the CPP\_CG::File::DiffDelimiter property. The default DiffDelimiter value is “//!”. The keywords are resolved in the following order:

- Predefined keywords
- Property keywords
- Tag keywords

Note the following:

- Keyword names can be written in parentheses. For example: \$(Name)
- If the value of a keyword is a MultiLine, each new line (except the first one) starts with the value of the CPP\_CG::Configuration::DescriptionBeginLine property; each line ends with the value of the CPP\_CG::Configuration::DescriptionEndLine property.

## ImplementationProlog

The ImplementationProlog property enables you to add any code that you want to be added as verbatim text (to be ignored by the software) to the beginning of the definition of a model element. For example, you could wrap a section of code with an #ifdef-#endif pair, add compiler-specific keywords, or add a #pragma statement. For example, to specify that an operation is available only when the code is compiled

with `_DEBUG`, set the following properties for the operation:

- Set `SpecificationProlog` to `#ifdef _DEBUG cr.`
- Set `SpecificationEpilog` to `#endif.`
- Set `ImplementationProlog` to `#ifdef _DEBUG cr.`
- Set `ImplementationEpilog` to `#endif.`

The following table lists whether leading and trailing linefeeds are generated.

Metaclass	Trailing Linefeed Added?	Class	No	Package	Yes
-----------	--------------------------	-------	----	---------	-----

*Default = Empty MultiLine*

## MarkPrologEpilogInAnnotations

The `MarkPrologEpilogInAnnotations` property specifies whether to generate ignore annotations for the `Specification Prolog`, `Implementation Prolog`, `Specification Epilog`, and `Implementation Epilog` properties so they are ignored during roundtrip.

When you insert code element declarations (variables, types, functions, and so on) in the `Specification Prolog`, `Implementation Prolog`, `Specification Epilog`, and `Implementation Epilog` properties, after a full roundtrip those elements are added to the model and are duplicated on the next code generation. When you use the `MarkPrologEpilogInAnnotations` property, you can have Rational Rhapsody automatically ignore the information specified in the `Specification Prolog`, `Implementation Prolog`, `Specification Epilog`, and `Implementation Epilog` properties instead of adding the ignore annotations manually.

The possible values for the `MarkPrologEpilogInAnnotations` property are:

- **None** - Rational Rhapsody does not generate any annotations. Any models created before Version 4.1 automatically have this property setting.
- **Ignore** - Rational Rhapsody generates the `///  
[ ignore annotation before the code specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, and generates the ///  
] annotation after the code specified in those properties.`
- **Auto** - If the code in the `Specification Prolog`, `Implementation Prolog`, `Specification Epilog`, and `Implementation Epilog` properties is one line (it does not contain any newline characters `(\n)`), no annotations are generated (the same behavior has the `None` setting). If there is more than one line, Rational Rhapsody generates the `///  
[ ignore annotation before the code specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, and generates the ///  
] annotation after the code specified in those properties (the same behavior as the Ignore setting).`

*During roundtrip, any ignore annotations in the comments of the element are not included in its description. Because the `Specification Prolog`, `Implementation Prolog`, `Specification Epilog`, and `Implementation Epilog` properties are generated between the annotation and its declaration for the element, you cannot rename those elements on roundtrip. If you change the name of an element, it is removed from the model and added with the new name. Some model information (for example, property settings) might be lost. Default = Auto*

## Simplify

If you are using the Rational Rhapsody customizable code generation mechanism, the Simplify property can be used to change the way specific types of elements are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - The element is ignored.
- Copy - The element is copied from the original to the simplified model. It is not modified in any way.
- Default - Uses the standard simplification for this item, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user.
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for the element has been applied.

Note that this property refers to the simplification of component files.

*Default = "Default"*

## SpecificationEpilog

The SpecificationEpilog property allows you to add code to the end of the declaration of a model element.

For example, you can use this property to add an `#ifdef` to indicate that an operation is available only when the code is compiled with `_DEBUG`, by setting the following properties:

- For SpecificationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For SpecificationEpilog, enter the value `#endif`
- For ImplementationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For ImplementationEpilog, enter the value `#endif`

*Default = Blank*

## SpecificationFooter

The SpecificationFooter property specifies the multiline footer to be generated at the end of specification files.

The default is as follows:

```
/****** File Path:
$FullCodeGeneratedFileName *****/
```

Footer format strings can contain any of the following keywords:

- \$ProjectName - The project name.
- \$ComponentName - The component name.
- \$ConfigurationName - The configuration name.
- \$ModelElementName - The name of the element mapped to the file. If there is more than one, this is

the name of the first element.

- `$FullModelElementName` - The name of the element mapped to the file, including the full path. If there is more than one, this is the name of the first element.
- `$CodeGeneratedDate` - The generation date.
- `$CodeGeneratedTime` - The generation time.
- `$RhapsodyVersion` - The version of Rational Rhapsody that generated the file.
- `$Login` - The user who generated the file.
- `$CodeGeneratedFileName` - The name of the generated file.
- `$FullCodeGeneratedFileName` - The full file name.
- `$Tag` - The value of the specified the element tag.
- `$Property` - The value of the element property with the specified name.

To avoid redundant compilation, Rational Rhapsody avoids unnecessary changes to specific lines prefixed with a special string, defined by the `CPP_CG::File::DiffDelimiter` property. The default `DiffDelimiter` value is `“//!”`. The keywords are resolved in the following order:

- Predefined keywords (such as `$Name`)
- Property keywords
- Tag keywords

Note the following:

- Keyword names can be written in parentheses. For example: `$(Name)`
- If the value of a keyword is a `MultiLine`, each new line (except the first one) starts with the value of the `CPP_CG::Configuration::DescriptionBeginLine` property; each line ends with the value of the `CPP_CG::Configuration::DescriptionEndLine` property.

## SpecificationHeader

The `SpecificationHeader` property specifies the multiline header to be generated at the beginning of specification files.

The default is as follows:

```
/****** Rhapsody: $RhapsodyVersion
Login: $Login Component: $ComponentName Configuration: $ConfigurationName Model Element:
$FullModelElementName //! Generated Date: $CodeGeneratedDate File Path:
$FullCodeGeneratedFileName *****/
```

Header format strings can contain any of the following keywords:

- `$ProjectName` - The project name.
- `$ComponentName` - The component name.
- `$ConfigurationName` - The configuration name.
- `$ModelElementName` - The name of the element mapped to the file. If there is more than one, this is the name of the first element.
- `$FullModelElementName` - The name of the element mapped to the file, including the full path. If there

is more than one, this is the name of the first element.

- `$CodeGeneratedDate` - The generation date.
- `$CodeGeneratedTime` - The generation time.
- `$RhapsodyVersion` - The version of Rational Rhapsody that generated the file.
- `$Login` - The user who generated the file.
- `$CodeGeneratedFileName` - The name of the generated file.
- `$FullCodeGeneratedFileName` - The full file name.
- `$Tag` - The value of the specified the element tag.
- `$Property` - The value of the element property with the specified name.

To avoid redundant compilation, Rational Rhapsody avoids unnecessary changes to specific lines prefixed with a special string, defined by the `CPP_CG::File::DiffDelimiter` property. The default `DiffDelimiter` value is `“//!”`. The keywords are resolved in the following order:

- Predefined keywords (such as `$Name`)
- Property keywords
- Tag keywords

Note the following:

- Keyword names can be written in parentheses. For example: `$(Name)`
- If the value of a keyword is a `MultiLine`, each new line (except the first one) starts with the value of the `CPP_CG::Configuration::DescriptionBeginLine` property; each line ends with the value of the `CPP_CG::Configuration::DescriptionEndLine` property.

## SpecificationProlog

The `SpecificationProlog` property adds code to the beginning of the declaration of a model element (such as a configuration or class).

For example, to create an abstract class in Java, you can set the `SpecificationProlog` property for the class to "abstract." You must include the space after the word "abstract." If the visibility for the class is set to default, the following class declaration is generated in the .java file:

abstract class classname { ... } The `SpecificationProlog` property allows you to add compiler-specific keywords, add a `#pragma` statement, or wrap a section of code with an `#ifdef-#endif` pair.

For example, to specify that an operation is available only when the code is compiled with `_DEBUG`, set the following properties for the operation:

- Set `SpecificationProlog` to `#ifdef _DEBUG cr.`
- Set `SpecificationEpilog` to `#endif.`
- Set `ImplementationProlog` to `#ifdef _DEBUG cr.`
- Set `ImplementationEpilog` to `#endif.`

The following table lists whether leading and trailing linefeeds are generated.

Metaclass	Leading Linefeed Added?	Trailing Linefeed Added?	Class	Yes	No	Package	Yes	Yes
-----------	-------------------------	--------------------------	-------	-----	----	---------	-----	-----



*Default = Empty MultiLine*

## flowPort

The flowPort metaclass controls whether code is generated for flowports.

### InvokeRelay

Use the InvokeRelay property to control the relay of data from flowports.

This possible values are:

- UponValueChange - Data is sent through the flowport only if a change from the previous data happens.
- Always - Data is always sent through the flowport.

*Default = UponValueChange*

### OptimizeCode

Code generation for ports and flow ports was optimized in version 7.5.3 of Rhapsody, relative to the code generated in previous versions. A new property named OptimizeCode was added with a default value of True. In the C++ backward compatibility profile for 7.5.3., the value for this property is set to False so that the old code generation mechanism will be used for ports and flow ports in older models.

### ReceiveRelay

Use the ReceiveRelay property to control the notification event (the 'chXXX' event) called when data has arrived through a flowport.

This possible values are:

- UponValueChange - The notification event is called only if a change of data happens.
- Always - Notification event is called even if no change of data occurs.

*Default = UponValueChange*

### SupportMulticast

Use this property to control the multicasting ability of a flowport. Use the property to enable data or events to be sent from one sender flowport to many.

The possible values are:

- Always - Rational Rhapsody always generates multicasting ability to each flowport in the model.

- Smart - Rational Rhapsody identifies flowports that are connected to more than 1 flowport and generates code to support multicasting to those flowports only.
- Never - Rational Rhapsody never generates code supporting multicasting of data/event through flowports. This is the value for models created before Rational Rhapsody 7.5, so the old behavior is the same.

*Default = Smart*

## Framework

The Framework metaclass contains properties that affect the Rational Rhapsody framework.

### ActivateFrameworkDefaultEventLoop

The ActivateFrameworkDefaultEventLoop property specifies the framework call that initializes the framework main event loop.

*Default = OXF::start(\$Fork);*

The value of \$Fork is calculated from the CG::Configuration::StartFrameworkInMainThread property for regular applications and from the CORBA::Configuration::StartFrameworkInMainThread property for CORBA servers.

This property can be set at the configuration level or higher.

### ActiveBase

The ActiveBase property specifies the superclass from which to specialize all threads, if the ActiveBaseUsage property is set to Checked.

*Default = OMThread*

### ActiveBaseUsage

The ActiveBaseUsage property specifies whether to use the superclass specified by the ActiveBase property as the superclass for all threads.

*Default = Checked*

### ActiveDestructorGuard

The ActiveDestructorGuard property specifies the macro that starts protection for an active user object destructor.

*Default = START\_DTOR\_THREAD\_GUARDED\_SECTION*

## ActiveExecuteOperationName

The ActiveExecuteOperationName property sets the user object virtual table for an active object and passes it to a task in the task initialization function (RiCTask\_init()). Follow these steps:

- Create a method with the following signature: struct RiCReactive \* operation name> (RiCTask \* const)
- Set the operation name in the ActiveExecuteOperationName property.
- Start the execution of the active object task by calling the RICTASK\_START() macro on the object.

The virtual function table member name is stored in the ActiveVtblName property.

*Default = Empty string*

## ActiveGuardInitialization

The ActiveGuardInitialization property specifies the call that makes the active object event dispatching guarded.

*Default = setToGuardThread*

## ActiveIncludeFiles

The ActiveIncludeFiles property specifies the base class for threads when you use selective framework includes. If a class is active and this property is defined, the file specified by the property is included in the class specification file.

*Default = oxf/omthread.h*

## ActiveInit

The ActiveInit property specifies the format of the declaration generated for the initializer for an active class.

*(Default =*

## ActiveMessageQueueSize

The ActiveMessageQueueSize property specifies the size of the message queue allocated for active objects, if the ActiveMessageQueueSize property for classes is left blank.

*Default = OMOSThread::DefaultMessageQueueSize*

## ActiveStackSize

The ActiveStackSize property specifies the size of the stack allocated for active objects, if the ActiveStackSize property for classes is left blank.

*Default = is OMOSThread::DefaultStackSize*

## ActiveThreadName

The ActiveThreadName property specifies the name of threads, if the ActiveThreadName property for classes is left blank.

*Default = ""*

## ActiveThreadPriority

The ActiveThreadPriority priority specifies the priority of threads, if the ActiveThreadPriority property for classes is left blank.

*Default = OMOSThread::DefaultThreadPriority*

## ActiveVtblName

*The ActiveVtblName property stores the name of the virtual function table that is associated with a task (the RiCTask member of the structure). Default = \$ObjectName\_activeVtbl*

## BooleanType

*The BooleanType property specifies the Boolean type used by the framework. Default = RhpBoolean*

## CurrentEventId

*The CurrentEventId property specifies the call or macro used to obtain the ID of the currently consumed event. Default = OM\_CURRENT\_EVENT\_ID*

## DefaultProvidedInterfaceName

*The DefaultProvidedInterfaceName property specifies the interface that must be implemented by the "in" part of a rapid port. See the Rational Rhapsody help for more information on rapid ports. Default = DefaultProvidedInterface*

## DefaultReactivePortBase

*The `DefaultReactivePortBase` property stores the base class for the generic rapid port (or default reactive port). This base class relays all events. See the Rational Rhapsody help for more information on rapid ports. Default = `OMDefaultReactivePort`*

## **DefaultReactivePortIncludeFiles**

The `DefaultReactivePortIncludeFiles` property specifies the include files that are referenced in the generated file that implements the class with the rapid ports. See the Rational Rhapsody help for more information on rapid ports.

*Default = `<oxf/OMDefaultReactivePort.h>`*

## **DefaultRequiredInterfaceName**

The `DefaultRequiredInterfaceName` property specifies the interface that must be implemented by the "out" part of a rapid port. See the Rational Rhapsody help for more information on rapid ports.

*Default = `DefaultRequiredInterface`*

## **EnableDirectReactiveDeletion**

The `EnableDirectReactiveDeletion` property specifies the call to the framework that supports direct deletion of reactive instances (by using the delete operator) instead of graceful framework termination (by using the reactive `destroy()` method).

When you use `destroy()`, the object waits in a zombie mode until all the events that are designated to it are removed from the active context queue, and then self -destructs.

In this scheme, there is no need to traverse the queue of the active context to cancel pending events, and there is no need to make the reactive destructor guarded to ensure safe deletion. A reactive object can be either in a graceful termination or forced deletion (by using the delete operator) state: you cannot use graceful deletion on an object that allows forced deletion, and vice versa.

You can set a single reactive object in a forced deletion state, or set the entire system (all reactive instances) in a forced deletion state (as is done for compatibility with earlier versions). Graceful termination should not be used when a reactive part (of a composite class) runs in a context of an active object that is not part of, and different from, the composite active context.

If you are using a Rational Rhapsody library component as part of an application where the main is not generated by Rational Rhapsody (for example, GUI applications), the framework initializes itself in full compatibility mode on the call to `OXF::init()`. If you want to remove part or all of the compatibility features, call `OXF::initialize()` instead of `OXF::init()` (the operation takes the same arguments) and add independent activation calls that are compatible with earlier versions, prior to the `initialize()` call. Note that the `CPP_CG::Framework::UseDirectReactiveDeletion` property must be set to `True` for this property to take effect. When it is set to `True`, the code specified in the `EnableDirectReactiveDeletion` is generated in the main prior to the call to `OXF::init()`.

*Default = `OXF::supportExplicitReactiveDeletion();`*

## EventBase

The EventBase property specifies the base class for all events, if the EventBaseUsage property is set to Checked.

*Default = OMEvent*

## EventBaseUsage

The EventBaseUsage property specifies whether to use the event superclass specified by the EventBase property as the parent of all events.

*Default = Checked*

## EventGenerationPattern

The EventGenerationPattern property supplies some of the information needed to generate code for Send Action elements in C, C++, and Java.

For each language, code generation for this element is determined by the following properties:

- `CPP_CG::Framework::EventGenerationPattern` - general format
- `CPP_CG::Framework::EventToPortGenerationPattern` - used when sending even to a port

Note: Rational Rhapsody does not support roundtripping for Send Action elements.

## EventIdType

When Rational Rhapsody generates code for an event, it creates an ID number for the event. The EventIDType property allows you to specify the type that should be used for this number if you do not want to use the type that Rational Rhapsody generates by default. In C and C++, the value of this property affects code generation only if the EventIdAsDefine property is set to False.

*Default = short*

## EventIncludeFiles

The EventIncludeFiles property specifies the base class for events when you use selective framework includes. If events are defined in a package, the file specified by this property is included in the package specification file to enable the use of events and timeouts in the package.

*Default = <oxf/event.h>*

## EventSender

The `EventSender` property specifies the base class to use for reactive interfaces when the `ReactiveInterfaceScheme` property is set to `Thin`. This allows you to define your own interface for event sending behavior instead of the framework class `IOxfEventSender`.

*(Default = `IOxfEventSender`)*

## EventSetParamsStatement

The `EventSetParamsStatement` property specifies a template for the body of the `setParams()` method, provided by the Rational Rhapsody framework for Java, to set the parameters of an event. For example, for an event of type `evOn()`, the default template would generate the following code in the body of the `setParams()` method: `evOn params = (evOn) event;` The default value is as follows: `$eventType params = ($eventType) event;`

## FrameworkInitialization

The `FrameworkInitialization` property specifies the framework initialization code that is called by the `main`.

The default is as follows:

```
OXF::initialize($(Argc)$(Argv)$(AnimationPortNumber)$(RemoteHost)$(TimerResolution)$(TimerMaxTimeouts)$(TimeM
```

## HeaderFile

The `HeaderFile` property specifies the framework header files to be included in objects that are within the scope of a particular configuration.

*Default = `oxf/oxf.h`*

## IncludeHeaderFile

The `IncludeHeaderFile` property specifies whether to include the framework header files specified by the `CG::Framework::HeaderFile` property in the project.

*Default = `Checked`*

## InnerReactiveClassName

The `InnerReactiveInstanceName` property specifies the name of a reactive class that serves as a bridge between a reactive class in your model and the framework. The implementation scheme of reactive classes is different in Java than in C++. Java does not allow inheritance from the reactive framework classes because that would mean that you would not be able to inherit from an additional base class that might not be reactive.

The chosen alternative is to delegate an inner class instance that inherits from `RiJStateReactive`.

Delegation is the implementation of an interface that forwards relevant messages to the inner class instance.

*Default = Reactive*

## InnerReactiveInstanceName

The InnerReactiveInstanceName property specifies the name of a reactive instance that serves as a bridge between a reactive class in your model and the framework. The implementation scheme of reactive classes is different in Java than in C++. Java does not allow inheritance from the reactive framework classes because that would mean that you would not be able to inherit from an additional base class that might not be reactive.

The chosen alternative is to delegate an inner class instance that inherits from RiJStateReactive. Delegation is the implementation of an interface that forwards relevant messages to the inner class instance.

*Default = reactive*

## InstrumentationHeaderFile

The InstrumentationHeaderFile property specifies the header file required for animation.

*Default = aom/aom.h*

## InstrumentVtblName

The InstrumentVtblName property specifies the name of the virtual function table that is associated with animation objects. Each animated object has its own virtual function table (Vtbl). This table creates your own framework, with its own virtual instrumentation functions, and connect it to Rational Rhapsody.

*Default = \$ObjectName\_instrumentVtbl*

## InvokePortMultiPattern

When using Call Operation elements, you can specify that the target should be a port on an object. In cases where the multiplicity of the port is greater than 1, the InvokePortMultiPattern property specifies the code that should be generated to invoke the operation via the port.

*Default = OUT\_PORT\_AT(\$target, \$index)->*

## InvokePortPattern

When using Call Operation elements, you can specify that the target should be a port on an object. For such cases, the InvokePortPattern property specifies the code that should be generated to invoke the operation via the port.



*Default = OUT\_PORT(\$target)->*

## **IsCompletedCall**

*The IsCompletedCall property specifies the call or macro that determines whether the state reached a final state so it can be exited on a null transition. The property supports the \$State keyword so you can use state-based calls. The keyword is resolved to the state implementation (code) name. Default = IS\_COMPLETED(\$State)*

## **IsInCall**

*The IsInCall property specifies the query that determines whether the state is in the current active configuration. The property supports the \$State keyword so you can use state-based calls. The keyword is resolved to the state implementation (code) name. Default = IS\_IN(\$State)*

## **MakeFileName**

The MakeFileName property specifies a new name for the makefile. To use this property, add the following line to the .prp file: Property MakeFileName String "MyFileName"

In this syntax, MyFileName specifies the name of the makefile.

## **NullTransitionId**

*The NullTransitionId property specifies the ID reserved for null transition consumption. Default = OMNullEventId*

## **OperationGuard**

*The OperationGuard property specifies the macro that guards an operation. Default = GUARD\_OPERATION*

## **ProtectedBase**

The ProtectedBase property specifies the base class for protected objects, if the ProtectedBaseUsage property is set to Checked.

*Default = Empty string*

## **ProtectedBaseUsage**

The ProtectedBaseUsage property specifies whether to use the class specified by the ProtectedBase property as the base class for protected objects.

*Default = Checked*

## ProtectedClassDeclaration

*The ProtectedClassDeclaration property affects how protected classes are implemented. Beginning with Rational Rhapsody 4.0, instead of inheriting from OMProtected, the class embeds an aggregate OMProtected. The aggregate member and helper methods are defined in the macro OMDECLARE\_GUARDED (defined in omprotected.h). Default = OMDECLARE\_GUARDED*

## ProtectedIncludeFiles

The ProtectedIncludeFiles property specifies the base class for protected classes when you use selective framework includes.

Default value = <oxf/omprotected.h>

## ProtectedInit

The ProtectedInit property specifies the declaration generated for the initializer for guarded objects. The default value for Ada is an empty string. The default value for C is as follows: \$base\_init(\$member)

## ReactiveBase

The ReactiveBase property specifies the base class for all reactive classes, if the ReactiveBaseUsage property is set to Checked.

*Default = OMReactive*

## ReactiveBaseUsage

The ReactiveBaseUsage property specifies whether to use the class specified by the ReactiveBase property as the base class for all reactive objects.

*Default = Checked*

## ReactiveConsumeEventOperationName

The ReactiveConsumeEventOperationName property sets the user object virtual table for a reactive object. Follow these steps:

- Create a method with the following signature: void operation name>(RiCReactive \* const, RiCEvent\*)
- Set the operation name in the ReactiveConsumeEventOperationName property.

*Rational Rhapsody Developer for Ada ignores all the values for the properties under the Framework metaclass except for this one. Default = Empty string*

## ReactiveCtorActiveArgDefaultValue

The constructor for a reactive class contains a single argument, which specifies the active object (thread and message queue) for processing incoming events. The ReactiveCtorActiveArgDefaultValue property specifies the default value for this argument.

If you are modifying the default value, you can use a UML tag on the class for this purpose. For example, you can define a tag named `activeObject`, and set the value of ReactiveCtorActiveArgDefaultValue to `$activeObject`. If you then set the tag value as `p_active` for a class named `Sensor`, you would end up with the following code for the constructor:

```
Sensor(IOxfActive* theActiveContext = p_active) { }
```

*Default = 0*

## ReactiveCtorActiveArgName

The constructor for a reactive class contains a single argument, which specifies the active object (thread and message queue) for processing incoming events. The ReactiveCtorActiveArgName property specifies the name used for this argument.

*Default = theActiveContext*

## ReactiveCtorActiveArgType

The constructor for a reactive class contains a single argument, which specifies the active object (thread and message queue) for processing incoming events. The ReactiveCtorActiveArgType property specifies the type of this argument.

*Default = IOxfActive\**

## ReactiveDestructorGuard

*The ReactiveDestructorGuard property specifies the macro that starts protection of a section of code used for destruction of a reactive instance. This prevents a "race" (between the deletion and event dispatching) when deleting an active instance. Default = START\_DTOR\_REACTIVE\_GUARDED\_SECTION*

## ReactiveEnableAccessEventData

*The ReactiveEnableAccessEventData property specifies the code to be used to enable access to the specific event data in a transition (typically by assigning a local variable of the appropriate type). The property supports the \$Event keyword so you can specify the event type. Default = OMSETPARAMS(\$Event);*

## ReactiveGetStateCall

The `ReactiveGetStateCall` property specifies the framework code used for getting the current state of a reactive class.

*Default = `OMReactive::getReactiveInternalState()`;*

## ReactiveGuardInitialization

*The `ReactiveDestructorGuard` property specifies the framework call that makes the event consumption of a specific reactive class guarded. Default = `setToGuardReactive`*

## ReactiveHandleEventNotConsumed

*The `ReactiveHandleEventNotConsumed` property registers a method to handle unconsumed events in a reactive class. Specify the method name as value for this property. Default = Empty string*

## ReactiveHandleTONotConsumed

*The `ReactiveHandleTONotConsumed` property registers a method to handle unconsumed trigger operations in a reactive class. Specify the method name as the value for this property. Default = Empty string*

## ReactiveIncludeFiles

The `ReactiveIncludeFiles` property specifies the base classes for reactive classes when you use selective framework includes. If a class is reactive and this property is defined, the file specified by the property is included in the class specification file. For reactive classes, the header files specified by the following properties are also included:

- `EventIncludeFiles` - For the event base class
- `ActiveIncludeFiles` - If the class is guarded or instrumented

*Default = `<oxf/omreactive.h,oxf/state.h>`*

## ReactiveInit

The `ReactiveInit` property specifies the declaration for the initializer generated for reactive objects. The default pattern for C is as follows: `$base_init($member, (void*)$mePtr, $task, $VtblName)`; The `$base` variable is replaced with the name of the reactive object during code generation. The string “\_init” is appended to the object name in the name of the operation. For example, if the reactive object is named A, the initializer generated for A is named `A_init()`. The `$member` variable is replaced with the name of the reactive member (equivalent to the base class) of the object during code generation.

The `$mePtr` variable is replaced with the name of the user object (the value of the `Me` property). The member and `mePtr` objects are not equivalent if the user object is active. The `$VtblName` variable is replaced with the name of the virtual function table for an object, specified by the `ReactiveVtblName` property. The default value for Ada is an empty string. The default for C is as follows:

```
$base_init($member, (void*)$mePtr, $task, $VtblName);
```

The default for Java is as follows: `reactive = new Reactive($task);`

## ReactiveInterface

*The ReactiveInterface property specifies the name of the interface class that forwards messages to an inner class instance of a reactive class in order to implement its reactive behavior. Default = RiJStateConcept*

## ReactiveSetEventHandlingGuard

*The ReactiveSetEventHandlingGuard property controls the code that is generated within the constructor of a reactive class. When you use this property with guarded triggered operations, it enables guarding of the event handling (in order to provide mutual exclusion between the event and TO handling). Default = setEventGuard(getGuard());*

## ReactiveSetStateCall

The ReactiveSetStateCall property specifies the framework code used for setting the current state of a reactive class.

*Default = OMReactive::setReactiveInternalState(oxfReactiveState);*

## ReactiveSetTask

The ReactiveSetTask property specifies the string that tells a reactive object whether it is an active or a sequential instance.

*Default = setActiveContext(\$task, \$isActive);*

## ReactiveStateType

The ReactiveStateType property is used for serialization to define the oxfstate type.

*Default = unsigned long*

## ReactiveTakeEventStatusType

For classes with statecharts, you can use the property ReactiveTakeEventStatusType to specify the type that is returned by the event-processing operations, such as `rootState_processEvent`, and `rootState_handleEvent`.

If the property is left blank, `IOxfReactive::TakeEventStatus` is used as the return type for these operations.

In order to have an effect on the generated code, this property must be modified at the project level or for specific Configurations. It will not affect the generated code if you change the value at the package or class level.

Note that if you modify the value of this property, you must also change the return type defined for the virtual function `rootState_processEvent` in the framework class `OMReactive` so that it matches the type that you specified for the property.

*Default = Blank*

## ReactiveVtblName

*The ReactiveVtblName property specifies the name of the virtual function table (Vtbl) associated with a reactive object. Each reactive object has its own Vtbl, which creates your own framework and connect it to Rational Rhapsody. Default = \$ObjectName\_reactiveVtbl*

## SetManagedTimeoutCanceling

The `SetManagedTimeoutCanceling` property is a property for backward compatibility that specifies whether the framework uses the pre-Rational Rhapsody 6.0 scheme of timeout creation and cancellation (where `OMTimerManager` is responsible for cancellation of timeouts) or the Rational Rhapsody 6.0 scheme.

In Rational Rhapsody 6.0, the framework moves the responsibility for a timeout cancellation from the timer manager to the timeout client (the reactive object). This change reduces the timer manager responsibilities and the overhead in timeout management (thus improving timeout scheduling performance).

The change also includes changes in the generated code (the user reactive objects hold pointers to the waiting timeouts in order to enable canceling).

If you are using a Rational Rhapsody library component as part of an application where the main is not generated by Rational Rhapsody (for example, GUI applications), the framework initializes itself in full compatibility mode on the call to `OXF::init()`.

If you want to remove part or all of the compatibility features, call `OXF::initialize()` instead of `OXF::init()` (the operation takes the same arguments) and add independent, backward-compatibility activation calls prior to the `initialize()` call.

*Default = `OXF::setManagedTimeoutCanceling(true);`*

## SetRhp5CompatibilityAPI

*The `SetRhp5CompatibilityAPI` property specifies the call that configures models created before Rational Rhapsody 6.0 so they use the 5. x version of the framework instead of the new one. See `UseRhp5CompatibilityAPI` for more information on Version 5. x compatibility mode. Default = `OXF::setRhp5CompatibleAPI(true);`*

## StaticMemoryIncludeFiles

The StaticMemoryIncludeFiles property specifies the files to be included in the package specification file if static memory management is enabled and you are you use selective framework includes.

*Default = <oxf/MemAlloc.h>*

## StaticMemoryPoolDeclaration

The StaticMemoryPoolDeclaration property specifies the declaration of the memory pool for timeouts.

*Default = DECLARE\_MEMORY\_ALLOCATOR(\$Class, \$BaseNumberOfInstances)*

## StaticMemoryPoolImplementation

The StaticMemoryPoolImplementation property specifies the generated code in the implementation file for a memory pool implementation (see the BaseNumberOfInstances property).

*Default = IMPLEMENT\_MEMORY\_ALLOCATOR(\$Class, \$BaseNumberOfInstances, \$AdditionalNumberOfInstances, \$ProtectStaticMemoryPool)*

## TestEventTypeCall

*The TestEventTypeCall property specifies the test used in event consumption code to check if the currently consumed event is of a given type. Default = IS\_EVENT\_TYPE\_OF(\$Id)*

## TimeoutId

*The TimeoutId property specifies the ID reserved for timeout events. Default = OMTimeoutEventId*

## TimerMaxTimeouts

*The TimerMaxTimeouts property specifies the maximum number of timeouts allowed simultaneously in the system, if the TimerMaxTimeouts property for the configuration is not overridden. In the framework, the default number of timers is 100. Default = Empty string*

## TimerResolution

The TimerResolution property allows you to override the default tick time used.

The number entered is the number of milliseconds used for the tick time.

The default tick time (currently 100 milliseconds) is defined by OMTimerManagerDefaults::defaultTicktime in the file OMTimerManagerDefaults.cpp

*Default = Blank*

## UseDirectReactiveDeletion

*The UseDirectReactiveDeletion property determines whether direct deletion of reactive instances (by using the delete operator) is used instead of graceful framework termination (by using the reactive destroy() method). When this property is set to Checked, the code specified in the EnableDirectReactiveDeletion is generated in the main prior to the call to OXF::init(). See EnableDirectReactiveDeletion and the upgrade history on the support site for more information on this functionality. Default = Cleared*

## UseManagedTimeoutCanceling

The UseManagedTimeoutCanceling property specifies whether the framework uses the pre-Rational Rhapsody 6.0 scheme of timeout creation and cancellation (so OMTimerManager is responsible for cancellation of timeouts). In Rational Rhapsody 6.0, the framework moves the responsibility for a timeout cancellation from the timer manager to the timeout client (the reactive object).

This change reduces the timer manager responsibilities and the overhead in timeout management (thus improving timeout scheduling performance). The change also includes changes in the generated code (the user reactive objects hold pointers to the waiting timeouts in order to enable canceling). When loading a pre-6.0 model, Rational Rhapsody sets the project CPP\_CG::Framework::UseManagedTimeoutCanceling to Checked to set the system-compatibility mode.

See the upgrade history on the support site for more information.

*Default = Cleared*

## UseRhp5CompatibilityAPI

The UseRhp5CompatibilityAPI property specifies whether to use the virtual functions of the core implementation classes that existed in the pre-Rational Rhapsody 6.0 framework. The Rational Rhapsody 6.0 framework introduces a set of interfaces for the core behavioral framework. The interfaces define a concise API for the framework and enable you to replace the actual implementation of these interfaces while maintaining the framework behavior.

As a result of the interfaces' introduction, the framework behavioral classes (OMReactive, OMThread, and OMEvent) use a new set of virtual operations to implement the interfaces and provide the behavioral infrastructure. To support existing customizations of these classes (made by inheriting and overriding the virtual operations), the framework can work in a mode where the pre-6.0 API virtual operations are called.

When loading a pre-6.0 model, Rational Rhapsody sets the project property CPP\_CG::Framework::UseRhp5CompatibilityAPI to True to set the system-compatibility mode. If this is set to Checked, the pre-6.0 API is called by the framework instead of the interface-based API. Without this flag, user customizations can compile but is not called. See the upgrade history on the support site for more information on the Version 5. x compatibility mode.

*Default = Cleared*



## General

The General metaclass contains a property that specifies the Friend implementation scheme.

### <<Friend>>ImplementationScheme

The <<Friend>>ImplementationScheme property specifies how a friendship relation between classes is generated into code. According to the UML version 1.3 standard, a dependency that is stereotyped as <<Friend>> from a class A to a class B means that A is a friend of B.

Therefore, in the C++ implementation, class B grants friendship to class A. Rhapsody 3.0 and higher supports both semantics. Select the semantics you want by setting the <<Friend>>ImplementationScheme property to the appropriate value:

- UML1.3 - Use the UML <<Friend>> implementation semantics.
- Rhapsody2.3 - Use the Rational Rhapsody 2.3 <<Friend>> implementation semantics.

When you use the UML1.3 scheme, you can set the UsageType property so #include macros are generated in the source file for the friend. By default, this property is set to Specification, meaning that the #include is generated in the specification file. The default value of the <<Friend>>ImplementationScheme property UML1.3.

However, when you first load a version 2.3 model into Rational Rhapsody version 3.0 or higher, this property is set to Rational Rhapsody 2.3 at the project level. This enables the implementation scheme to be consistent for the entire model.

You can override this property at the package level, so some packages use the UML1.3 scheme whereas others use the Rational Rhapsody V2.3 scheme. Packages that had been previously imported into the model do not return to their old property value. Instead, they receive the project value, unless the property was overridden before the package was imported.

*Default = UML1.3*

## Generalization

The Generalization metaclass contains a property used to support generalization. See the Rational Rhapsody help for more information on generalization.

### Animate

The Animate property specifies whether animation code is generated for an element. You can specify your own animation function by using the CPP\_CG::Type::AnimSerializeOperation property. The semantics of the Animate property is always in favor of the owner settings:

- If a package Animate property is set to Cleared, all the classes owned by the package are not animated, regardless of the class Animate settings.
- If a class Animate property is set to Cleared, all the elements in the class (attributes, operations, relations, and so on) are not animated.
- If an operation Animate property is set to Cleared, all the arguments are not animated.
- If the AnimateArguments property is set to Cleared, all the arguments are not animated, regardless of the specific argument Animate property settings.

*Default = Checked*

## Simplify

If you are using the Rational Rhapsody customizable code generation mechanism, the Simplify property can be used to change the way specific types of elements are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - The element is ignored.
- Copy - The element is copied from the original to the simplified model. It is not modified in any way.
- Default - Uses the standard simplification for this item, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for the element has been applied.

*Default = "Default"*

## INTEGRITY

The INTEGRITY metaclass contains the environment settings (Compiler, framework libraries, and so on) for INTEGRITY 4.0.X .

### AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path. Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files. To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)/LangCpp/osconfig/INTEGRITY.*

## **AdditionalReservedWords**

The AdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = Empty string*

## **BLDAdditionalDefines**

The BLDAdditionalDefines property specifies additional compiler preprocessor flags.

*Default = Empty string*

## **BLDAdditionalOptions**

The BLDAdditionalOptions property specifies additional compilation switches.

*Default =*

*:optimizestrategy=space :driver\_opts=--diag\_suppress=14 :driver\_opts=--diag\_suppress=550*

## **BLDIncludeAdditionalBLD**

The IncludeAdditional property specifies additional build options.

*Default = Empty MultiLine*

## **BLDMainExecutableOptions**

The BLDMainExecutableOptions property specifies the options generated in the main build file of the executable component of the model. The default values for the C++ INTEGRITY metaclass are as follows:

*:defines=\_DEBUG :target\_os=integrity :integrity\_option=dynamic :staticlink=true*

## **BLDMainLibraryOptions**

The BLDMainLibraryOptions property specifies the options generated in the main build file of the library component of the model.

*Default =*

:defines=\_DEBUG :target\_os=integrity :integrity\_option=dynamic :staticlink=true

## **BLDTarget**

The BLDTarget property specifies the target BSP. For example, ":target=Win32". This property also affects the names of the framework libraries used in the link.

*Default = mbx800*

## **BuildArgumentsInIDE**

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the CPP\_CG::[environment]::BuildCommandInIDE property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## **BuildCommandInIDE**

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## **BuildCommandSet**

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration.

To change this property, use the Configuration window in the browser - do not change it by using the Properties tab in the Features window or by modifying the site.prp file. Note that this property also affects

the names of the framework libraries used in the link.

The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*Default = Debug*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT/etc/IntegrityMake.bat\" IntegrityBuild.bat  
buildLibs \$BLDTarget bld "*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## DebugSwitches

The DebugSwitches property sets the debug level used in debug switches. The default values are as follows:

Environment Possible Values Default Value INTEGRITY Default, Multi, None, Plain, and Stack Default  
OBJECTADA -ga, -gc, -ga -gc -ga RAVEN\_PPC -ga, -gc, -ga -gc -ga SPARK Empty string

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction with an IDE such as Eclipse, the EnableDebugIntegrationWithIDE property can be used to specify whether or not the IDE debugger should be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to True, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's exit() function is called automatically. The system function calls the destructors of framework instances in the order it determines. The EndApplicationCode property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an endApplication() function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to endApplication(), it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The EntryPoint property specifies the name of the main program for a given environment.

*Default = main*

See also the definition of the EntryPointDeclarationModifier property for more information.

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens has three parameters, each with an integer value:

- TotalNumberOfTokens - The number of tokens in the regular expression
- FileTokenPosition - The position of the file name token in the expression
- LineTokenPosition - The position of the line number token in the expression

*Default = TotalNumberOfTokens=3,FileTokenPosition=1,LineTokenPosition=2*

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the CPP\_CG::<Environment>::ExeName property plus the value of this property.

*(Default = .mod)*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the CPP\_CG::<Environment>::ExeExtension property.

*(Default = Blank)*

## FrameworkLibPrefix

The FrameworkLibPrefix property specifies the prefix of the Rational Rhapsody framework library linked with your application.

*Default = Integrity*

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile. Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

*Default =*

`$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)WebComponents$(LIB_POSTFIX)$(LIB_EXT),  
$(OMROOT)\lib\$(LIB_PREFIX)WebServices$(LIB_POSTFIX)$(LIB_EXT)`

## HasIDEInterface

The HasIDEInterface property is a Boolean value that specifies whether IDE support is enabled. If IDE support is enabled (Checked), the IDEInterfaceDLL property points to an IDE adapter that provides connection to the IDE. If the property is set to Cleared, IDE support is disabled and IDE services are not attempted. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = Cleared*

## IDEInterfaceDLL

The IDEInterfaceDLL property is a string that points to the IDE adapter DLL. You should not have any reason to modify this property. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = Empty string*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment.

*Default = .cpp*

## IntegrityRoot

If you are developing an application for one of the INTEGRITY environments, the IntegrityRoot property is used for makefile generation.

The value of this property can be the path to your INTEGRITY installation, such as C:\GHS\int504, or an environmental variable called \$INTEGRITY\_ROOT if you have defined a variable with this name to point to your INTEGRITY installation.

*Default = Blank*

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*Default = Empty string*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```



*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\IntegrityMake.bat\" \$makefile \$maketarget"*

## **InvokeMakeGenerator**

The InvokeMakeGenerator specifies the path to the executable program for an external makefile generator. This external generator is started each time you request a makefile generation. If the specified path is incorrect, Rational Rhapsody generates an error message. If you are using a full-featured external code generator, this property setting is ignored.

*Default = \$OMROOT/etc/IntegrityMakefileGenerator.bat*

## **IsFileNameShort**

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format.

If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## **LibExtension**

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .a*

## **MakeExtension**

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the CPP\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*(Default = .bld)*

## **MakeFileName**

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the `CPP_CG::<Environment>::MakeExtension` property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## **NullValue**

The `NullValue` property specifies an alternative expression for `NULL` in the generated code.

*Default = NULL*

## **ObjectName**

The `ObjectName` property specifies an alternative name for the compiled object file in the generated makefile.

*Default = Empty string*

## **ObjectsDirectory**

The `ObjectsDirectory` property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = work*

## **ObjExtension**

The `ObjExtension` property specifies the extension appended to compiled object files for a given environment.

*Default = .o*

## **OSFileSystemCaseSensitive**

The `OSFileSystemCaseSensitive` property specifies whether the OS file system for a given environment is case sensitive.

*Default = Cleared*

## **ParseErrorDescript**

The `ParseErrorDescript` property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be

displayed in the Description column on the Build tab of the Output window.

*Default = ([^"]+)"[,][ ]line ([0-9]+)[: ] (warning/error/catastrophic error) (.\*)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. The default values are as follows:

*Default = ([^"]+)"[,][ ]line ([0-9]+)[: ] (warning/error/catastrophic error)*

## ParseSeverityError

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^"]+)"[,][ ]line ([0-9]+)[: ] (error/catastrophic error)*

## ParseSeverityWarning

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^"]+)"[,][ ]line ([0-9]+)[: ] (warning)*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation.

*Default = /*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Cleared*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely.

In such configurations, the host is the machine running the product, whereas the target is the machine running the application.

To run remotely, the UseRemoteHost property must be set to Checked. If UseRemoteHost is Checked and RemoteHost is blank, the current host name is used for the remote host. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = Empty string*

## **ReusableStatechartSwitches**

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts. See the upgrade history on the support site for information on this change.

*Default = OM\_REUSABLE\_STATECHART\_IMPLEMENTATION.*

## **SpecExtension**

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*Default = .h*

## **UpdateBuildSettingsInIDE**

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## **UseActorsCode**

The UseActorsCode property specifies whether code is generated for actors. The value of the property should be synchronized with the configuration Generate Code For Actors check box (located in the configuration Initialization tab).

*Default = Cleared*

## **UseNewBuildOutputWindow**

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log

tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the `CG::General::ShowLogViewAfterBuild` property to `True`.

*Default = Checked*

## UseNonZeroStdInputHandle

The `UseNonZeroStdInputHandle` property is a Boolean value that specifies whether to use a non-zero standard input handle.

*Default = Cleared*

## UseRemoteHost

The `UseRemoteHost` property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Checked*

## UseTemplateName

In the declarations for members of a template class, some compilers require that you use precede the member type with the keyword "typename" for types with dependent names. Since support for this keyword varies between compilers, the `UseTemplateName` property is used to specify whether the "typename" keyword should be included in the generated code.

*Default = Cleared*

# INTEGRITY5

The `INTEGRITY5` metaclass contains the environment settings (Compiler, framework libraries, and so on) for `INTEGRITY 5.0.X`

## AdaptorSearchPath

The `AdaptorSearchPath` property specifies the path to the operating system configuration file. This path is added to the generated makefile search path. Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files. To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)/LangCpp/osconfig/INTEGRITY*

## **AdditionalReservedWords**

The AdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = Empty string*

## **AnimInstLibs**

The AnimInstLibs property is used to specify the static libraries that are required when Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with INST\_LIBS.

*Default = -l\$(LibPrefix)OxfInst\$(BLDTarget)\$OMLibSuffix\$LibExtension  
-l\$(LibPrefix)AomAnim\$(BLDTarget)\$OMLibSuffix\$LibExtension  
-l\$(FrameworkLibPrefix)OmComAppl\$(BLDTarget)\$OMLibSuffix\$LibExtension*

## **AnimPreprocessor**

The AnimPreprocessor property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with INST\_FLAGS.

*Default = -D\_OMINSTRUMENT*

## **BLDAdditionalDefines**

The BLDAdditionalDefines property specifies additional compiler preprocessor flags.

*Default = Empty string*

## **BLDAdditionalOptions**

The BLDAdditionalOptions property specifies additional compilation switches.

*Default =*

`-I. --diag_suppress 14,550,611,1795 :outputDirRelative=$ObjectsDirectory`

## **BLDIncludeAdditionalBLD**

The BLDIncludeAdditionalBLD specifies additional build options.

*Default = Empty MultiLine*

## **BLDMainExecutableOptions**

The BLDMainExecutableOptions property specifies the options generated in the main build file of the executable component of the model.

The default is as follows:

`-G-dynamic-non_shared-wantprototype-Ospace-tnone-delete`

## **BLDMainLibraryOptions**

The BLDMainLibraryOptions property specifies the options generated in the main build file of the library component of the model.

*Default =*

`-G -non_shared -wantprototype -Ospace -tnone -delete`

## **BLDTarget**

The BLDTarget property specifies the target BSP. For example, `":target=Win32"`. This property also affects the names of the framework libraries used in the link.

*Default = mbx800*

## **BuildArgumentsInIDE**

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the `CPP_CG::[environment]::BuildCommandInIDE` property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandInIDE

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration.

To change this property, use the Configuration window in the browser - do not change it by using the Properties tab in the Features window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link.

The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*Default = Debug*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT/etc/Integrity5Make.bat\" IntegrityBuild.bat buildLibs \$BLDTarget "*



## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## CompileDebug

The CompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*Default = Empty string*

## CompileSwitches

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property. The default values are as follows:

*Default = -DUSE\_Iostream*

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*Default = -D\_DEBUG -G*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*Default = Empty string*

## DebugLibSuffix

A suffix added to library names. OMLibSuffix keyword is replaced with this property or ReleaseLibSuffix according to the compilation to the build type: Release/Debug.

## DebugSwitches

The DebugSwitches property sets the debug level used in debug switches. The default values are as follows:

Environment	Possible Values	Default Value
INTEGRITY	Default, Multi, None, Plain, and Stack	Default
OBJECTADA	-ga, -gc, -ga -gc -ga	RAVEN_PPC -ga, -gc, -ga -gc -ga
SPARK	Empty string	

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction with an IDE such as Eclipse, the EnableDebugIntegrationWithIDE property can be used to specify whether or not the IDE debugger should be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to True, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's exit() function is called automatically. The system function calls the destructors of framework instances in the order it determines. The EndApplicationCode property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an endApplication() function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to endApplication(), it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The EntryPoint property specifies the name of the main program for a given environment.

*Default = main*

See also the definition of the EntryPointDeclarationModifier property for more information.

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens

has three parameters, each with an integer value:

- TotalNumberOfTokens - The number of tokens in the regular expression
- FileTokenPosition - The position of the file name token in the expression
- LineTokenPosition - The position of the line number token in the expression

*Default = TotalNumberOfTokens=3,FileTokenPosition=1,LineTokenPosition=2*

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the CPP\_CG::<Environment>::ExeName property plus the value of this property.

*(Default = .mod)*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the CPP\_CG::<Environment>::ExeExtension property.

*(Default = Blank)*

## ExtraFilesToCopy

A list of file names (with full paths) separated with commas. The generator copies this list of files to the folder of the makefile only if the file does not already exist.

*Default = \$OMROOT/MakeTpl/INTEGRITY5.ld*

## FrameworkLibPrefix

The FrameworkLibPrefix property specifies the prefix of the Rational Rhapsody framework library linked with your application.

*Default = Integrity5*

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile. Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

*Default =*

`$(LibPrefix)WebComponents$(BLDTarget)$OMLibSuffix$LibExtension ,  
$OMRoot/lib/$(FrameworkLibPrefix)WebServices$(BLDTarget)$OMLibSuffix$LibExtension`

## HasIDEInterface

The HasIDEInterface property is a Boolean value that specifies whether IDE support is enabled.

If IDE support is enabled (Checked), the IDEInterfaceDLL property points to an IDE adapter that provides connection to the IDE. If the property is set to Cleared, IDE support is disabled and IDE services are not attempted. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = Checked*

## IDEInterfaceDLL

The IDEInterfaceDLL property is a string that points to the IDE adapter DLL. You should not have any reason to modify this property. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = \$MULTI\_ROOT/rhapsody\_multi\_ide.dll*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment.

*Default = .cpp*

## IntegrityLinkFile

The name of the Integrity link file that should be added to the makefile template.

*Default = INTEGRITY5.ld*

## IntegrityRoot

If you are developing an application for one of the INTEGRITY environments, the IntegrityRoot property is used for makefile generation.

The value of this property can be the path to your INTEGRITY installation, such as C:\GHS\int504, or an environmental variable called \$INTEGRITY\_ROOT if you have defined a variable with this name to point to your INTEGRITY installation.

*Default = \$INTEGRITY\_ROOT*

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*Default = Empty string*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT/etc/vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

*Default =*

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT/etc/Integrity5Make.bat\" $makefile $maketarget"
```

## InvokeMakeGenerator

The InvokeMakeGenerator specifies the path to the executable program for an external makefile generator. This external generator is started each time you request a makefile generation. If the specified path is incorrect, Rational Rhapsody generates an error message. If you are using a full-featured external code generator, this property setting is ignored.

*Default = \$OMROOT/etc/Integrity5MakefileGenerator.bat*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format.

If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## **LibExtension**

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .a*

## **LibPrefix**

Combines all the prefixes of library names.

Default = \$(FrameworkLibPrefix)\$(OMMultipleAddressSpacesPrefix)

## **LinkDebug**

The LinkDebug property specifies the special link switches used to link in debug mode.

*Default = Empty string (blank)*

## **LinkRelease**

The LinkRelease property specifies the special link switches used to link in release mode.

*Default = Empty string*

## **LinkSwitches**

The LinkSwitches property specifies the standard link switches used to link in any mode.

*Default = Empty string*

## **MakeExtension**

The MakeExtension property can be used to specify the file extension you would like to use for the

makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the CPP\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*(Default = .gpj)*

## MakeFileContentForExe1

The content of the makefiles, in case of executable component type.

*Default =*

```
#!/gbuild primaryTarget=$PrimaryTarget [INTEGRITY Application] -bsp $BLDTarget -os_dir
$IntegrityRoot -o $OMTargetName$ExeExtension -object_dir=$ObjectsDirectory
$BLDMainExecutableOptions $OMMultipleAddressSpacesLibraries $KernelProject
$MakeFileNameForExe2 $IntegrateFile $BSPFile $ConnectionFile $ResourceFile
$OMMultipleAddressSpacesAdditionalFiles
```

## MakeFileContentForExe2

The content of the makefiles, in case of executable component type.

*Default =*

```
#!/gbuild [Program] -o $OMTargetName -object_dir=$ObjectsDirectory $BLDAdditionalOptions
-ISOmRoot/LangCpp -L$OMRoot/LangCpp/lib $OMUserIncludePath $LinkSwitches
$OMCompilationFlag $CompileSwitches $OMReusableFlag $OMInstrumentationFlags
$OMInstrumentationLibs $OMMultipleAddressSpacesLibraries $BLDAdditionalDefines $OMUserLibs
$OMMainFiles$ImpExtension $OMSrcFiles $IntegrityLinkFile $LinkerFile
```

## MakeFileContentForLib1

The content of the makefiles, in case of library component type.

The default is as follows:

```
#!/gbuild primaryTarget=$PrimaryTarget [Project] -bsp $BLDTarget -os_dir $IntegrityRoot
-object_dir=$ObjectsDirectory $BLDMainLibraryOptions $OMMultipleAddressSpacesSwitches
$KernelProject $MakeFileNameForLib2 $IntegrateFile $BSPFile $ConnectionFile $ResourceFile
$OMMultipleAddressSpacesAdditionalFiles
```

## MakeFileContentForLib2

The content of the makefiles, in case of library component type.

The default is as follows:

```
#!gbuild [Library] -o $OMTargetName$LibExtension -object_dir=$ObjectsDirectory  
$BLDAdditionalOptions -I$OMRoot/LangCpp $OMUserIncludePath $OMCompilationFlag  
$CompileSwitches $OMInstrumentationFlags $OMReusableFlag $BLDAdditionalDefines $OMSrcFiles
```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the CPP\_CG::<Environment>::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## MakeFileNameForExe1

The name of the makefiles, in case of executable component type.

*Default = \$(OMTargetName)\$MakeExtension*

## MakeFileNameForExe2

The name of the makefiles, in case of executable component type.

*Default = \$(OMTargetName)\_program\$MakeExtension*

## MakeFileNameForLib1

The name of the makefiles, in case of library component type.

*Default = \$(OMTargetName)\$MakeExtension*

## MakeFileNameForLib2

The name of the makefiles, in case of library component type.

*Default = \$(OMTargetName)\_library\$MakeExtension*

## MultipleAddressSpacesInFileContent



The content of the MultipleAddressSpacesIntFileName file.

*Default =*

```
# Integrate configuration file for compiling a list of address spaces that use also POSIX shared memory
manager Kernel Filename DynamicDownload EndKernel $OMSubComponentInfo AddressSpace
shared_memory_manager Filename posix_shm_manager MaximumPriority 200 Language $OMLanguage
Task Initial StartIt true EndTask EndAddressSpace
```

## MultipleAddressSpacesIntFileName

a file with this name is created in case of multiple address space compilation.

*Default = \$OMTargetName.int*

## MultipleAddressSpacesLibraries

names of libraries to add in case of multiple address space usage.

*Default =*

```
-l$(FrameworkLibPrefix)Dox$(BLDTarget)$LibExtension -llibposix$LibExtension
-llibshm_client$LibExtension
```

## MultipleAddressSpacesPrefix

A prefix that is added to libraries in case of multiple address space compilation.  
OMMultipleAddressSpacesPrefix keyword adds this prefix when needed.

*Default = Distributed*

## MultipleAddressSpacesSwitches

A switch for multiple address space compilation. The makefile template can add it directly but it is preferred to use the keyword OMMultipleAddressSpacesSwitches – that checks whether this switch should be added.

*Default = -DRIC\_DISTRIBUTED\_SYSTEM*

## NetAndSocketLibs

A list of library names that is added to OMWebLibs keyword if web-enabling flag is on or to OMInstrumentationFlags keyword if the instrumentation is in animation mode.

The default is as follows:

`-llibsocket.a -llibnet.a`

## NoneInstLibs

The NoneInstLibs property is used to specify the static libraries that are required when Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with INST\_LIBS.

*Default = -l\$(LibPrefix)Oxf\$(BLDTarget)\$OMLibSuffix\$LibExtension*

## NonePreprocessor

The NonePreprocessor property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with INST\_FLAGS.

*Default = Blank*

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## ObjectName

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = Empty string*

## ObjectsDirectory

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = work*

## ObjExtension

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*Default = .o*

## OSFileSystemCaseSensitive

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Cleared*

## ParseErrorDescript

The ParseErrorDescript property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = ([^"]+)"[,][ ]line ([0-9]+)[: ](warning/error/catastrophic error) (.\*)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. The default values are as follows:

*Default = ([^"]+)"[,][ ]line ([0-9]+)[: ](warning/error/catastrophic error)*

## ParseSeverityError

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^"]+)"[,][ ]line ([0-9]+)[: ](error/catastrophic error)*

## ParseSeverityWarning

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^"]+)"[,][ ]line ([0-9]+)[: ](warning)*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation.

*Default = /*

## **PosixSharedMemoryFiles**

This list is copied only in case of multiple address space compilation.

Default = \$OMROOT/MakeTmpl/posix\_shm\_manager.gpj,\$OMROOT/MakeTmpl/shm\_area.c

## **QuoteOMROOT**

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Cleared*

## **ReleaseLibSuffix**

A suffix added to library names. OMLibSuffix keyword is replaced with this property or DebugLibSuffix according to the compilation to the build type: Release/Debug.

*Default = R*

## **RemoteHost**

The RemoteHost property specifies the name of the host machine when you run an application remotely.

In such configurations, the host is the machine running the product, whereas the target is the machine running the application.

To run remotely, the UseRemoteHost property must be set to Checked. If UseRemoteHost is Checked and RemoteHost is blank, the current host name is used for the remote host. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = Empty string*

## **ReusableStatechartSwitches**

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts. See the upgrade history on the support site for information on this change.

*Default = OM\_REUSABLE\_STATECHART\_IMPLEMENTATION*

## **SpecExtension**

The SpecExtension property determines the extension that Rational Rhapsody appends to generated

specification (header) files for a given language and environment.

*Default = .h*

## TraceInstLibs

The TraceInstLibs property is used to specify the static libraries that are required when Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with INST\_LIBS.

*Default = -l\$(LibPrefix)OxfInstTrace\$(BLDTarget)\$OMLibSuffix\$LibExtension  
-l\$(LibPrefix)AomTrace\$(BLDTarget)\$OMLibSuffix\$LibExtension  
-l\$(LibPrefix)TomTrace\$(BLDTarget)\$OMLibSuffix\$LibExtension  
-l\$(FrameworkLibPrefix)OmComAppl\$(BLDTarget)\$OMLibSuffix\$LibExtension*

## TracePreprocessor

The TracePreprocessor property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with INST\_FLAGS.

*Default = -DOMTRACER*

## UpdateBuildSettingsInIDE

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## UseActorsCode

The UseActorsCode property specifies whether code is generated for actors. The value of the property should be synchronized with the configuration Generate Code For Actors check box (located in the configuration Initialization tab).

*Default = Cleared*

## UseNewBuildOutputWindow

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the `CG::General::ShowLogViewAfterBuild` property to `True`.

*Default = Checked*

## UseNonZeroStdInputHandle

The `UseNonZeroStdInputHandle` property is a Boolean value that specifies whether to use a non-zero standard input handle.

For `INTEGRITY`, `OBJECTADA`, `RAVEN_PPC`, and `SPARK` the default value is `Cleared`; for the other environments, the default value is `Checked`.

## UseRemoteHost

The `UseRemoteHost` property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Checked*

## UseTemplateName

In the declarations for members of a template class, some compilers require that you use precede the member type with the keyword `"typename"` for types with dependent names. Since support for this keyword varies between compilers, the `UseTemplateName` property is used to specify whether the `"typename"` keyword should be included in the generated code.

*Default = Cleared*

## WebInstLibs

A list of library names that is added to `OMWebLibs` keyword if web-enabling flag is on.

## Integrity5ESTL

The `Integrity5ESTL` metaclass contains the environment settings (Compiler, framework libraries, and so on) for `INTEGRITY 5.0.X. Embedded C++ with Templates`.

## AdaptorSearchPath

The `AdaptorSearchPath` property specifies the path to the operating system configuration file. This path is added to the generated makefile search path. Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme

makes it easier to add a custom adapter because you do not need to modify the framework files. To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the `AdaptorSearchPath` property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)/LangCpp/osconfig/INTEGRITY*

## **AdditionalReservedWords**

The `AdditionalReservedWords` property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = Empty string*

## **AnimInstLibs**

The `AnimInstLibs` property is used to specify the static libraries that are required when Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with `INST_LIBS`.

*Default = -l\$(LibPrefix)OxfInst\$(BLDTarget)\$OMLibSuffix\$LibExtension  
-l\$(LibPrefix)AomAnim\$(BLDTarget)\$OMLibSuffix\$LibExtension  
-l\$(FrameworkLibPrefix)OmComAppl\$(BLDTarget)\$OMLibSuffix\$LibExtension*

## **AnimPreprocessor**

The `AnimPreprocessor` property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with `INST_FLAGS`.

*Default = \$(DEFINE\_QUALIFIER)OMANIMATOR \$(DEFINE\_QUALIFIER)\_\_USE\_W32\_SOCKETS*

## **BLDAdditionalDefines**

The `BLDAdditionalDefines` property specifies additional compiler preprocessor flags.

*Default = Empty string*

## BLDAdditionalOptions

The BLDAdditionalOptions property specifies additional compilation switches.

The default is as follows:

```
-I. --diag_suppress 14,550,611,1795 --one_instantiation_per_object --ee --eele  
--std_cxx_include_directory $(MULTI_ROOT)/eecxx --std_cxx_include_directory  
$(MULTI_ROOT)/ansi
```

## BLDIncludeAdditionalBLD

The BLDIncludeAdditionalBLD specifies additional build options.

*Default = Empty MultiLine*

## BLDMainExecutableOptions

The BLDMainExecutableOptions property specifies the options generated in the main build file of the executable component of the model.

*Default = -G -dynamic -non\_shared -wantprototype -Ospace -tlocal -delete*

## BLDMainLibraryOptions

The BLDMainLibraryOptions property specifies the options generated in the main build file of the library component of the model.

*Default = -G -non\_shared -wantprototype -Ospace -tlocal -delete*

## BLDTarget

The BLDTarget property specifies the target BSP. For example, ":target=Win32". This property also affects the names of the framework libraries used in the link.

*Default = mbx800*

## BuildArgumentsInIDE

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the CPP\_CG::[environment]::BuildCommandInIDE property is not an empty string.



Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandInIDE

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration.

To change this property, use the Configuration window in the browser - do not change it by using the Properties tab in the Features window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link.

The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*Default = Debug*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT/etc/Integrity5Make.bat\" IntegrityBuild.bat buildLibs \$BLDTarget ESTL "*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## CompileDebug

The CompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*Default = Empty string*

## CompileSwitches

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property. The default values are as follows:

*Default = -DUSE\_Iostream -DOM\_STL*

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*Default = -D\_DEBUG -G*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*Default = Empty MultiLine*

## DebugLibSuffix

A suffix added to library names. OMLibSuffix keyword is replaced with this property or ReleaseLibSuffix according to the compilation to the build type: Release/Debug.

## DebugSwitches

The DebugSwitches property sets the debug level used in debug switches. The default values are as follows:

Environment	Possible Values	Default Value
INTEGRITY	Default, Multi, None, Plain, and Stack	Default
OBJECTADA	-ga, -gc, -ga -gc -ga	RAVEN_PPC -ga, -gc, -ga -gc -ga
SPARK	Empty string	

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction with an IDE such as Eclipse, the EnableDebugIntegrationWithIDE property can be used to specify whether or not the IDE debugger should be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to True, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's exit() function is called automatically. The system function calls the destructors of framework instances in the order it determines. The EndApplicationCode property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an endApplication() function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to endApplication(), it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The EntryPoint property specifies the name of the main program for a given environment.

*Default = main*

If applicable, see also the definition of the EntryPointDeclarationModifier property for more information.

## EnvironmentVarName

The EnvironmentVarName property specifies the name of the global variable that you must define in order to use the Embedded C++ compiler. It is used by the MultiMakefileGenerator. The value replaces the \$EnvironmentVarName value keyword inside the value for the BLDAdditionalOptions property.

*Default = MULTI\_ROOT*

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens has three parameters, each with an integer value:

- TotalNumberOfTokens - The number of tokens in the regular expression
- FileTokenPosition - The position of the file name token in the expression
- LineTokenPosition - The position of the line number token in the expression

*Default = TotalNumberOfTokens=3,FileTokenPosition=1,LineTokenPosition=2*

## ESTLCompliance

The ESTLCompliance property is a Boolean value that determines whether you are using the Embedded C++ (ESTL) environment and conform to its requirements. In instrumentation mode, the Rational Rhapsody code generator usually creates an OMAAnimatedUser Class> friend class for each user-defined class. This class inherits from AOMInstance, if its User Class> does not inherit from another class in the model. This inheritance is virtual and is needed for multiple inheritance support. Because ESTL does not support multiple inheritance (as far as virtual inheritance), the Rational Rhapsody Developer for C++ code generator does not create “virtual” inheritance if ESTLCompliance is set to True. To support ESTL compliance, Rational Rhapsody includes a new check to recognize the following elements of ESTL-noncompliance:

- Multiple inheritance, caused by the user model (several superclasses)
- Multiple inheritance, caused by Rational Rhapsody (an active reactive class is generated with two base classes: OMReactive and OMThread)
- Multiple inheritance, caused by a combination of the following factors:
  - An active class containing a superclass
  - A reactive class containing a superclass
  - Virtual inheritance, declared by the user in the features of the superclass

In these cases, Rational Rhapsody displays the following warning message for each problematic class: "ESTL does not support multiple/virtual inheritance" Note that this check runs only when the ESTLCompliance property is set to Checked.

*Default = Checked*

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the `CPP_CG::<Environment>::ExeName` property plus the value of this property.

*(Default = .mod)*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the `ExeName` property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the `CPP_CG::<Environment>::ExeExtension` property.

*(Default = Blank)*

## ExtraFilesToCopy

A list of file names (with full paths) separated with commas. The generator copies this list of files to the folder of the makefile only if the file does not already exist.

*Default = \$OMROOT/MakeTpl/INTEGRITY5.ld*

## FrameworkLibPrefix

The `FrameworkLibPrefix` property specifies the prefix of the Rational Rhapsody framework library linked with your application.

*Default = Integrity5ESTL*

## GetConnectedRuntimeLibraries

The `GetConnectedRuntimeLibraries` property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile. Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

*Default =*

`$(LibPrefix)WebComponents$(BLDTarget)$OMLibSuffix$LibExtension ,  
$OMRoot/lib/$(FrameworkLibPrefix)WebServices$(BLDTarget)$OMLibSuffix$LibExtension`

## HasIDEInterface

The HasIDEInterface property is a Boolean value that specifies whether IDE support is enabled.

If IDE support is enabled (Checked), the IDEInterfaceDLL property points to an IDE adapter that provides connection to the IDE. If the property is set to Cleared, IDE support is disabled and IDE services are not attempted. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = Checked*

## IDEInterfaceDLL

The IDEInterfaceDLL property is a string that points to the IDE adapter DLL. You should not have any reason to modify this property. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = \$MULTI\_ROOT/rhapsody\_multi\_ide.dll*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment. The default values are as follows:

*Default = .cpp*

## IntegrityLinkFile

The name of the Integrity link file that should be added to the makefile template.

*Default = INTEGRITY5.ld*

## IntegrityRoot

If you are developing an application for one of the INTEGRITY environments, the IntegrityRoot property is used for makefile generation.

The value of this property can be the path to your INTEGRITY installation, such as C:\GHS\int504, or an environmental variable called \$INTEGRITY\_ROOT if you have defined a variable with this name to point to your INTEGRITY installation.

*Default = \$INTEGRITY\_ROOT*

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*Default = Empty string*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe" "dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\Integrity5Make.bat\" \$makefile \$maketarget"*

## InvokeMakeGenerator

The InvokeMakeGenerator specifies the path to the executable program for an external makefile generator. This external generator is started each time you request a makefile generation. If the specified path is incorrect, Rational Rhapsody generates an error message. If you are using a full-featured external code generator, this property setting is ignored.

*Default = \$OMROOT/etc/Integrity5MakefileGenerator.bat*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format.

If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .a*

## LibPrefix

Combines all the prefixes of library names.

*Default = \$(FrameworkLibPrefix)\$(OMMultipleAddressSpacesPrefix)*

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

*Default = Empty string*

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

*Default = Empty string*

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

*Default = Empty MultiLine*

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the CPP\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*(Default = .gpj)*

## MakeFileContentForExe1

The content of the makefiles, in case of executable component type.

*Default =*

```
#!/gbuild primaryTarget=$PrimaryTarget [INTEGRITY Application] -bsp $BLDTarget -os_dir
$IntegrityRoot -o $OMTargetName$ExeExtension -object_dir=$ObjectsDirectory
$BLDMainExecutableOptions $OMMultipleAddressSpacesLibraries $KernelProject
$MakeFileNameForExe2 $BSPFile $ConnectionFile $ResourceFile
```



## MakeFileContentForExe2

The content of the makefiles, in case of executable component type.

*Default =*

```
#!/gbuild [Program] -o $OMTargetName -object_dir=$ObjectsDirectory $BLDAdditionalOptions
-I$OMRoot/LangCpp -L$OMRoot/LangCpp/lib $OMUserIncludePath $LinkSwitches
$OMCompilationFlag $CompileSwitches $OMReusableFlag $OMInstrumentationFlags
$OMInstrumentationLibs $OMMultipleAddressSpacesLibraries $BLDAdditionalDefines $OMUserLibs
$OMMainFiles$ImpExtension $OMSrcFiles $IntegrityLinkFile $LinkerFile
```

## MakeFileContentForLib1

The content of the makefiles, in case of library component type.

The default is as follows:

```
#!/gbuild primaryTarget=$PrimaryTarget [Project] -bsp $BLDTarget -os_dir $IntegrityRoot
-object_dir=$ObjectsDirectory $BLDMainLibraryOptions $MakeFileNameForLib2 $BSPFile
$ConnectionFile $ResourceFile
```

## MakeFileContentForLib2

The content of the makefiles, in case of library component type.

*Default =*

```
#!/gbuild [Library] -o $OMTargetName$LibExtension -object_dir=$ObjectsDirectory
$BLDAdditionalOptions -I$OMRoot/LangCpp $OMUserIncludePath $OMCompilationFlag
$CompileSwitches $OMInstrumentationFlags $OMReusableFlag $BLDAdditionalDefines $OMSrcFiles
```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the CPP\_CG::<Environment>::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## MakeFileNameForExe1

The name of the makefiles, in case of executable component type.

*Default = \$(OMTargetName)\$MakeExtension*

## **MakeFileNameForExe2**

The name of the makefiles, in case of executable component type.

*Default = \$(OMTargetName)\_program\$MakeExtension*

## **MakeFileNameForLib1**

The name of the makefiles, in case of library component type.

*Default = \$(OMTargetName)\$MakeExtension*

## **MakeFileNameForLib2**

The name of the makefiles, in case of library component type.

*Default = \$(OMTargetName)\_library\$MakeExtension*

## **MultipleAddressSpacesIntFileContent**

The content of the MultipleAddressSpacesIntFileName file.

*Default =*

```
# Integrate configuration file for compiling a list of address spaces that use also POSIX shared memory
manager Kernel Filename DynamicDownload EndKernel $OMSubComponentInfo AddressSpace
shared_memory_manager Filename posix_shm_manager MaximumPriority 200 Language $OMLanguage
Task Initial StartIt true EndTask EndAddressSpace
```

## **MultipleAddressSpacesIntFileName**

A file with this name is created in case of multiple address space compilation.

*Default = \$OMTargetName.int*

## **MultipleAddressSpacesLibraries**

Names of libraries to add in case of multiple address space usage.

*Default = -l\$(FrameworkLibPrefix)Dox\$(BLDTarget)\$LibExtension -llibposix\$LibExtension*

*-llibshm\_client\$LibExtension*

## MultipleAddressSpacesPrefix

A prefix that is added to libraries in case of multiple address space compilation. OMMultipleAddressSpacesPrefix keyword adds this prefix when needed.

*Default = Distributed*

## MultipleAddressSpacesSwitches

A switch for multiple address space compilation. The makefile template can add it directly but it is preferred to use the keyword OMMultipleAddressSpacesSwitches – that checks whether this switch should be added.

*Default = -DRIC\_DISTRIBUTED\_SYSTEM*

## NetAndSocketLibs

A list of library names that is added to OMWebLibs keyword if web-enabling flag is on or to OMInstrumentationFlags keyword if the instrumentation is in animation mode.

*Default = -llibsocket.a -llibnet.a*

## NoneInstLibs

The NoneInstLibs property is used to specify the static libraries that are required when Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with INST\_LIBS.

*Default = -l\$(LibPrefix)Oxf\$(BLDTarget)\$OMLibSuffix\$LibExtension*

## NonePreprocessor

The NonePreprocessor property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with INST\_FLAGS.

*Default = Blank*

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## ObjectName

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = Empty string*

## ObjectsDirectory

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = work*

## ObjExtension

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

Default = .o

## OSFileSystemCaseSensitive

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Cleared*

## ParseErrorDescript

The ParseErrorDescript property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = ([^"]+)"[,][ ]line ([0-9]+)[: ](warning/error/catastrophic error) (.\*)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = ([^"]+)"[,][ ]line ([0-9]+)[: ](warning/error/catastrophic error)*

## ParseSeverityError

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^"]+)"[,][ ]line ([0-9]+)[: ](error/catastrophic error)*

## ParseSeverityWarning

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^"]+)"[,][ ]line ([0-9]+)[: ](warning)*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation.

*Default = /*

## PosixSharedMemoryFiles

This list is copied only in case of multiple address space compilation.

*Default = \$OMROOT/MakeTmpl/posix\_shm\_manager.gpj,\$OMROOT/MakeTmpl/shm\_area.c*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Cleared*

## ReleaseLibSuffix

A suffix added to library names. OMLibSuffix keyword is replaced with this property or DebugLibSuffix according to the compilation to the build type: Release/Debug.

*Default = R*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely.

In such configurations, the host is the machine running the product, whereas the target is the machine running the application.

To run remotely, the UseRemoteHost property must be set to Checked. If UseRemoteHost is Checked and RemoteHost is blank, the current host name is used for the remote host. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = Empty string*

## ReusableStatechartSwitches

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts. See the upgrade history on the support site for information on this change.

*Default = OM\_REUSABLE\_STATECHART\_IMPLEMENTATION.*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*Default = .h*

## TraceInstLibs

The TraceInstLibs property is used to specify the static libraries that are required when Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with INST\_LIBS.

*Default = -l\$(LibPrefix)OxfInstTrace\$(BLDTarget)\$OMLibSuffix\$LibExtension  
-l\$(LibPrefix)AomTrace\$(BLDTarget)\$OMLibSuffix\$LibExtension  
-l\$(LibPrefix)TomTrace\$(BLDTarget)\$OMLibSuffix\$LibExtension  
-l\$(FrameworkLibPrefix)OmComAppl\$(BLDTarget)\$OMLibSuffix\$LibExtension*

## TracePreprocessor

The TracePreprocessor property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with INST\_FLAGS.

*Default = -DOMTRACER*

## UpdateBuildSettingsInIDE

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an

IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## **UseActorsCode**

The UseActorsCode property specifies whether code is generated for actors. The value of the property should be synchronized with the configuration Generate Code For Actors check box (located in the configuration Initialization tab).

*Default = Cleared*

## **UseNewBuildOutputWindow**

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the CG::General::ShowLogViewAfterBuild property to True.

*Default = Checked*

## **UseNonZeroStdInputHandle**

The UseNonZeroStdInputHandle property is a Boolean value that specifies whether to use a non-zero standard input handle.

For INTEGRITY, OBJECTADA, RAVEN\_PPC, and SPARK the default value is Cleared; for the other environments, the default value is Checked.

## **UseRemoteHost**

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Checked*

## **UseTemplateName**

In the declarations for members of a template class, some compilers require that you precede the member type with the keyword "typename" for types with dependent names. Since support for this

keyword varies between compilers, the `UseTemplateName` property is used to specify whether the "typename" keyword should be included in the generated code.

*Default = Cleared*

## WebInstLibs

A list of library names that is added to `OMWebLibs` keyword if web-enabling flag is on.

## Link

The `Link` metaclass contains a property that controls how links are handled during model simplification.

## Simplify

If you are using the Rational Rhapsody customizable code generation mechanism, the `Simplify` property can be used to change the way specific types of elements are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- `None` - The element is ignored.
- `Copy` - The element is copied from the original to the simplified model. It is not modified in any way.
- `Default` - Uses the standard simplification for this item, as defined in Rational Rhapsody.
- `ByUser` - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- `ByUserPostDefault` - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for the element has been applied.

*Default = "Default"*

## IntegrityESTL

The `IntegrityESTL` metaclass contains the environment settings (Compiler, framework libraries, and so on) for INTEGRITY 4.0.X. Embedded C++ with Templates.

## AdaptorSearchPath

The `AdaptorSearchPath` property specifies the path to the operating system configuration file. This path is added to the generated makefile search path. Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme



makes it easier to add a custom adapter because you do not need to modify the framework files. To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the `AdaptorSearchPath` property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)/LangCpp/osconfig/INTEGRITY*

## **AdditionalReservedWords**

The `AdditionalReservedWords` property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = Empty string*

## **BLDAdditionalDefines**

The `BLDAdditionalDefines` property specifies additional compiler preprocessor flags.

*Default = OM\_ESTL*

## **BLDAdditionalOptions**

The `BLDAdditionalOptions` property specifies additional compilation switches.

*Default =*

```
:optimizestrategy=space :driver_opts=--diag_suppress=14 :driver_opts=--diag_suppress=550  
:cx_mode=extended_embedded :cx_lib=eec :stdcxxincdirs=$(MULTI_ROOT)\eecxx  
:stdcxxincdirs=$(MULTI_ROOT)\ansi
```

## **BLDIncludeAdditionalBLD**

The `BLDIncludeAdditionalBLD` specifies additional build options.

*Default = Empty MultiLine*

## BLDMainExecutableOptions

The BLDMainExecutableOptions property specifies the options generated in the main build file of the executable component of the model.

The default is as follows:

```
:defines=_DEBUG :target_os=integrity :integrity_option=dynamic :staticlink=true
```

## BLDMainLibraryOptions

The BLDMainLibraryOptions property specifies the options generated in the main build file of the library component of the model.

*Default =*

```
:defines=_DEBUG :target_os=integrity :staticlink=true
```

## BLDTarget

The BLDTarget property specifies the target BSP. For example, ":target=Win32". This property also affects the names of the framework libraries used in the link.

*Default = mbx800*

## BuildArgumentsInIDE

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the CPP\_CG::[environment]::BuildCommandInIDE property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandInIDE

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration.

To change this property, use the Configuration window in the browser - do not change it by using the Properties tab in the Features window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link.

The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*Default = Debug*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT/etc/IntegrityMake.bat\" IntegrityBuild.bat  
buildLibs bld \$BLDTarget ESTL"*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## CompileDebug

The CompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

## CompileSwitches

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property. The default values are as follows:

*Default = -DUSE\_Iostream*

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*Default = -D\_DEBUG -G*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*Default = Empty string*

## DebugLibSuffix

A suffix added to library names. OMLibSuffix keyword is replaced with this property or ReleaseLibSuffix according to the compilation to the build type: Release/Debug.

## DebugSwitches

The DebugSwitches property sets the debug level used in debug switches. The default values are as follows:

Environment	Possible Values	Default Value
INTEGRITY	Default, Multi, None, Plain, and Stack	Default
OBJECTADA	-ga, -gc, -ga -gc -ga	RAVEN_PPC -ga, -gc, -ga -gc -ga
SPARK	Empty string	

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction with an IDE such as Eclipse, the EnableDebugIntegrationWithIDE property can be used to specify whether or not the IDE debugger should be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to True, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's `exit()` function is called automatically. The system function calls the destructors of framework instances in the order it determines. The `EndApplicationCode` property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an `endApplication()` function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to `endApplication()`, it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The `EntryPoint` property specifies the name of the main program for a given environment.

*Default = main*

See also the definition of the `EntryPointDeclarationModifier` property for more information.

## EnvironmentVarName

The `EnvironmentVarName` property specifies the name of the global variable that you must define in order to use the Embedded C++ compiler. It is used by the `MultiMakefileGenerator`. The value replaces the `$EnvironmentVarName` value keyword inside the value for the `BLDAdditionalOptions` property.

*Default = MULTI\_ROOT*

## ErrorMessageTokensFormat

The `ErrorMessageTokensFormat`, working with the `ParseErrorMessage` property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. `ErrorMessageTokensFormat` defines the number and location of tokens within the regular expression defined by the `ParseErrorMessage` property. `ErrorMessageTokens` has three parameters, each with an integer value:

- `TotalNumberOfTokens` - The number of tokens in the regular expression
- `FileTokenPosition` - The position of the file name token in the expression
- `LineTokenPosition` - The position of the line number token in the expression

*Default = ToTalNumberOfTokens=3,FileTokenPosition=1,LineTokenPosition=2*

## ESTLCompliance

The ESTLCompliance property is a Boolean value that determines whether you are using the Embedded C++ (ESTL) environment and conform to its requirements. In instrumentation mode, the Rational Rhapsody code generator usually creates an `OMAnimatedUser Class` friend class for each user-defined class. This class inherits from `AOMInstance`, if its `User Class` does not inherit from another class in the model. This inheritance is virtual and is needed for multiple inheritance support. Because ESTL does not support multiple inheritance (as far as virtual inheritance), the Rational Rhapsody Developer for C++ code generator does not create “virtual” inheritance if ESTLCompliance is set to True. To support ESTL compliance, Rational Rhapsody includes a new check to recognize the following elements of ESTL-noncompliance:

- Multiple inheritance, caused by the user model (several superclasses)
- Multiple inheritance, caused by Rational Rhapsody (an active reactive class is generated with two base classes: `OMReactive` and `OMThread`)
- Multiple inheritance, caused by a combination of the following factors:
  - An active class containing a superclass
  - A reactive class containing a superclass
- Virtual inheritance, declared by the user in the features of the superclass

In these cases, Rational Rhapsody displays the following warning message for each problematic class: "ESTL does not support multiple/virtual inheritance" Note that this check runs only when the ESTLCompliance property is set to Checked.

*Default = Checked*

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the `CPP_CG::<Environment>::ExeName` property plus the value of this property.

*(Default = .mod)*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value

of the `CPP_CG::<Environment>::ExeExtension` property.

*(Default = Blank)*

## ExtraFilesToCopy

A list of file names (with full paths) separated with commas. The generator copies this list of files to the folder of the makefile only if the file does not already exist.

The default value = `$OMROOT/MakeTmp/INTEGRITY5.ld`

## FrameworkLibPrefix

The `FrameworkLibPrefix` property specifies the prefix of the Rational Rhapsody framework library linked with your application.

*Default = IntegrityESTL*

## GetConnectedRuntimeLibraries

The `GetConnectedRuntimeLibraries` property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile. Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

*Default =*

`$(OMRoot)\LangCpp\lib\$(LIB_PREFIX)WebComponents$(LIB_POSTFIX)$(LIB_EXT),  
$(OMRoot)\lib\$(LIB_PREFIX)WebServices$(LIB_POSTFIX)$(LIB_EXT)`

## HasIDEInterface

The `HasIDEInterface` property is a Boolean value that specifies whether IDE support is enabled.

If IDE support is enabled (Checked), the `IDEInterfaceDLL` property points to an IDE adapter that provides connection to the IDE. If the property is set to Cleared, IDE support is disabled and IDE services are not attempted. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = Cleared*

## IDEInterfaceDLL

The `IDEInterfaceDLL` property is a string that points to the IDE adapter DLL. You should not have any reason to modify this property. In general, this is an environment property that can be contained in any of

the environment metaclasses supported by Rational Rhapsody.

*Default = Empty string*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment.

*Default = .cpp*

## IntegrityLinkFile

The name of the Integrity link file that should be added to the makefile template.

*Default = INTEGRITY5.ld*

## IntegrityRoot

If you are developing an application for one of the INTEGRITY environments, the IntegrityRoot property is used for makefile generation.

The value of this property can be the path to your INTEGRITY installation, such as C:\GHS\int504, or an environmental variable called \$INTEGRITY\_ROOT if you have defined a variable with this name to point to your INTEGRITY installation.

*Default = Blank*

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file. The default values are as follows:

*Default = Empty string*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.



"\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\vx6make.bat\" \$makefile \$maketarget 6.2 \$BSP gnu"

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\Integrity5Make.bat\" \$makefile \$maketarg*

## InvokeMakeGenerator

The InvokeMakeGenerator specifies the path to the executable program for an external makefile generator. This external generator is started each time you request a makefile generation. If the specified path is incorrect, Rational Rhapsody generates an error message. If you are using a full-featured external code generator, this property setting is ignored.

*Default = \$OMROOT/etc/Integrity5MakefileGenerator.bat*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format.

If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .a*

## LibPrefix

Combines all the prefixes of library names.

*Default = \$(FrameworkLibPrefix)\$ (OMMultipleAddressSpacesPrefix)*

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

*Default = Empty string (blank)*

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

*Default = Empty string*

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

*Default = Empty string*

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the CPP\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*(Default = .bld)*

## MakeFileContentForExe1

The content of the makefiles, in case of executable component type.

*Default =*

```
#!/gbuild primaryTarget=$PrimaryTarget [INTEGRITY Application] -bsp $BLDTarget -os_dir
$IntegrityRoot -o $OMTargetName$ExeExtension -object_dir=$ObjectsDirectory
$BLDMainExecutableOptions $OMMultipleAddressSpacesLibraries $KernelProject
$MakeFileNameForExe2 $IntegrateFile $BSPFile $ConnectionFile $ResourceFile
$OMMultipleAddressSpacesAdditionalFiles
```

## MakeFileContentForExe2

The content of the makefiles, in case of executable component type.

*Default =*

```
#!/gbuild [Program] -o $OMTargetName -object_dir=$ObjectsDirectory $BLDAdditionalOptions
-I$OMRoot/LangCpp -L$OMRoot/LangCpp/lib $OMUserIncludePath $LinkSwitches
$OMCompilationFlag $CompileSwitches $OMReusableFlag $OMInstrumentationFlags
$OMInstrumentationLibs $OMMultipleAddressSpacesLibraries $BLDAdditionalDefines $OMUserLibs
$OMMainFiles$ImpExtension $OMSrcFiles $IntegrityLinkFile $LinkerFile
```

## MakeFileContentForLib1

The content of the makefiles, in case of library component type.

The default is as follows:

```
#!/gbuild primaryTarget=$PrimaryTarget [Project] -bsp $BLDTarget -os_dir $IntegrityRoot
-object_dir=$ObjectsDirectory $BLDMainLibraryOptions $OMMultipleAddressSpacesSwitches
$KernelProject $MakeFileNameForLib2 $IntegrateFile $BSPFile $ConnectionFile $ResourceFile
$OMMultipleAddressSpacesAdditionalFiles
```

## MakeFileContentForLib2

The content of the makefiles, in case of library component type.

The default is as follows:

```
#!/gbuild [Library] -o $OMTargetName$LibExtension -object_dir=$ObjectsDirectory
$BLDAdditionalOptions -I$OMRoot/LangCpp $OMUserIncludePath $OMCompilationFlag
$CompileSwitches $OMInstrumentationFlags $OMReusableFlag $BLDAdditionalDefines $OMSrcFiles
```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the CPP\_CG::<Environment>::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## MakeFileNameForExe1

The name of the makefiles, in case of executable component type.

*Default = \$(OMTargetName)\$MakeExtension*

## MakeFileNameForExe2

The name of the makefiles, in case of executable component type.

*Default = \$(OMTargetName)\_program\$MakeExtension*

## MakeFileNameForLib1

The name of the makefiles, in case of library component type.

*Default = \$(OMTargetName)\$MakeExtension*

## MakeFileNameForLib2

The name of the makefiles, in case of library component type.

*Default = \$(OMTargetName)\_library\$MakeExtension*

## MultipleAddressSpacesIntFileContent

The content of the MultipleAddressSpacesIntFileName file.

*Default =*

```
# Integrate configuration file for compiling a list of address spaces that use also POSIX shared memory
manager Kernel Filename DynamicDownload EndKernel $OMSubComponentInfo AddressSpace
shared_memory_manager Filename posix_shm_manager MaximumPriority 200 Language $OMLanguage
Task Initial StartIt true EndTask EndAddressSpace
```

## MultipleAddressSpacesIntFileName

a file with this name is created in case of multiple address space compilation.

*Default = \$OMTargetName.int*

## MultipleAddressSpacesLibraries

names of libraries to add in case of multiple address space usage.

*Default =*

```
-l$(FrameworkLibPrefix)Dox$(BLDTarget)$LibExtension -llibposix$LibExtension
-llibshm_client$LibExtension
```

## MultipleAddressSpacesPrefix

A prefix that is added to libraries in case of multiple address space compilation.  
OMMultipleAddressSpacesPrefix keyword adds this prefix when needed.

*Default = Distributed*

## MultipleAddressSpacesSwitches

A switch for multiple address space compilation. The makefile template can add it directly but it is preferred to use the keyword `OMMultipleAddressSpacesSwitches` – that checks whether this switch should be added.

*Default = -DRIC\_DISTRIBUTED\_SYSTEM*

## NetAndSocketLibs

A list of library names that is added to `OMWebLibs` keyword if web-enabling flag is on or to `OMInstrumentationFlags` keyword if the instrumentation is in animation mode.

The default is as follows:

`-llibsocket.a -llibnet.a`

## NullValue

The `NullValue` property specifies an alternative expression for `NULL` in the generated code.

*Default = NULL*

## ObjectName

The `ObjectName` property specifies an alternative name for the compiled object file in the generated makefile.

*Default = Empty string*

## ObjectsDirectory

The `ObjectsDirectory` property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = work*

## ObjExtension

The `ObjExtension` property specifies the extension appended to compiled object files for a given environment.

*Default = .o*

## OSFileSystemCaseSensitive

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Cleared*

## ParseErrorDescript

The ParseErrorDescript property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = ([^"]+)"[,][ ]line ([0-9]+)[: ] (warning/error/catastrophic error) (.\*)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. The default values are as follows:

*Default = ([^"]+)"[,][ ]line ([0-9]+)[: ] (warning/error/catastrophic error)*

## ParseSeverityError

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^"]+)"[,][ ]line ([0-9]+)[: ] (error/catastrophic error)*

## ParseSeverityWarning

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^"]+)"[,][ ]line ([0-9]+)[: ] (warning)*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation.

*Default = /*

## **PosixSharedMemoryFiles**

This list is copied only in case of multiple address space compilation.

Default = \$OMROOT/MakeTmpl/posix\_shm\_manager.gpj,\$OMROOT/MakeTmpl/shm\_area.c

## **QuoteOMROOT**

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Cleared*

## **ReleaseLibSuffix**

A suffix added to library names. OMLibSuffix keyword is replaced with this property or DebugLibSuffix according to the compilation to the build type: Release/Debug.

*Default = R*

## **RemoteHost**

The RemoteHost property specifies the name of the host machine when you run an application remotely.

In such configurations, the host is the machine running the product, whereas the target is the machine running the application.

To run remotely, the UseRemoteHost property must be set to Checked. If UseRemoteHost is Checked and RemoteHost is blank, the current host name is used for the remote host. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = Empty string*

## **ReusableStatechartSwitches**

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts. See the upgrade history on the support site for information on this change.

*Default = OM\_REUSABLE\_STATECHART\_IMPLEMENTATION*

## **SpecExtension**

The SpecExtension property determines the extension that Rational Rhapsody appends to generated

specification (header) files for a given language and environment.

*Default = .h*

## **UpdateBuildSettingsInIDE**

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## **UseActorsCode**

The UseActorsCode property specifies whether code is generated for actors. The value of the property should be synchronized with the configuration Generate Code For Actors check box (located in the configuration Initialization tab).

*Default = Cleared*

## **UseNewBuildOutputWindow**

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the CG::General::ShowLogViewAfterBuild property to True.

*Default = Checked*

## **UseNonZeroStdInputHandle**

The UseNonZeroStdInputHandle property is a Boolean value that specifies whether to use a non-zero standard input handle.

For INTEGRITY, OBJECTADA, RAVEN\_PPC, and SPARK the default value is Cleared; for the other environments, the default value is Checked.

## **UseRemoteHost**

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.



*Default = Checked*

## UseTemplateName

In the declarations for members of a template class, some compilers require that you use precede the member type with the keyword "typename" for types with dependent names. Since support for this keyword varies between compilers, the UseTemplateName property is used to specify whether the "typename" keyword should be included in the generated code.

*Default = Cleared*

## WebInstLibs

A list of library names that is added to OMWebLibs keyword if web-enabling flag is on.

## Linux

The Linux metaclass controls the environment settings (Compiler, framework libraries, and so on) for Linux.

## AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)/LangCpp/osconfig/Linux*

## AdditionalReservedWords

The AdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = Empty string*

## **AnimIncludeDirectories**

The AnimIncludeDirectories property is used to specify the directories that must be referenced in the makefile for projects where Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with INST\_INCLUDES.

*Default = \$(INCLUDE\_QUALIFIER)\$(OMROOT)/LangCpp/tom*

## **AnimInstLibs**

The AnimInstLibs property is used to specify the static libraries that are required when Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with INST\_LIBS.

*Default = \$(OMROOT)/LangCpp/lib/linuxaomanim\$(LIB\_EXT)*

## **AnimOxfLibs**

The AnimOxfLibs property is used to specify the framework libraries that are required when Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with OXF\_LIBS.

*Default = \$(OMROOT)/LangCpp/lib/linuxoxfinst\$(LIB\_EXT)  
\$(OMROOT)/LangCpp/lib/linuxomcomappl\$(LIB\_EXT)*

## **AnimPreprocessor**

The AnimPreprocessor property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with INST\_FLAGS.

*Default = \$(DEFINE\_QUALIFIER)OMANIMATOR*

## **BuildCommandSet**

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration.

To change this property, use the Configuration window in the browser - do not change it by using the

Properties tab in the Features window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link.

The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*Default = Debug*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = \$OMROOT/etc/linuxmake linuxbuild.mak build*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## CompilerFlags

The CompilerFlags property allows you to define additional compilation flags. The value of the property is inserted into the value of the CompileSwitches property (Linux) or CPPCompileSwitches (cygwin). In the generated makefile, you can see the value of this property in the line that begins with ConfigurationCPPCompileSwitches=.

*Default = Blank*

## CompileSwitches

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property.

*Default =*

```
-I. -I$OMDefaultSpecificationDirectory -I$(OMROOT) -I$(OMROOT)/LangCpp  
-I$(OMROOT)/LangCpp/oxf $(INST_FLAGS) $(INCLUDE_PATH) $(INST_INCLUDES)  
-DUSE_Iostream $OMCPPCompileCommandSet -c
```

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default =*

```
@echo Compiling $OMFileImpPath $(CREATE_OBJ_DIR) @$$(CC) $OMFileCPPCompileSwitches -o  
$OMFileObjPath $OMFileImpPath
```

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*Default = -g*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*Default = -O*

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

*Default = \$OMFileObjPath : \$OMFileImpPath \$OMFileSpecPath \$OMFileDependencies*

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: \$OMFileObjPath : \$OMFileImpPath "\*.bmp" "\*.ico" "\*.rc2"

## DuplicateLibsListInMakeFile

The DuplicateLibsListInMakeFile property is a Boolean value that specifies whether Rational Rhapsody should duplicate the libraries list in the generated makefile link command. This property supports linkers

that are sensitive to library order in the link command.

*Default = Checked*

## EndApplicationCode

When the main function ends, the system's `exit()` function is called automatically. The system function calls the destructors of framework instances in the order it determines. The `EndApplicationCode` property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an `endApplication()` function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to `endApplication()`, it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The `EntryPoint` property specifies the name of the main program for a given environment.

*Default = main*

See also the definition of the `EntryPointDeclarationModifier` property for more information.

## ErrorMessageTokensFormat

The `ErrorMessageTokensFormat`, working with the `ParseErrorMessage` property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. `ErrorMessageTokensFormat` defines the number and location of tokens within the regular expression defined by the `ParseErrorMessage` property. `ErrorMessageTokens` has three parameters, each with an integer value:

- `TotalNumberOfTokens` - The number of tokens in the regular expression
- `FileTokenPosition` - The position of the file name token in the expression
- `LineTokenPosition` - The position of the line number token in the expression

*Default = TotalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2*

## FileDependencies

The `FileDependencies` property specifies which framework specification files and implementation files should be included in model elements. The file inclusions are generated in the makefile.

*Default = \$OMSpecIncludeInElements \$OMImpIncludeInElements*

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile. Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

*Default =*

`$(OMROOT)/LangCpp/lib/linuxWebComponents$(LIB_EXT),  
$(OMROOT)/lib/linuxWebServices$(LIB_EXT)`

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment.

*Default = .cpp*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*Default = include*

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*Default = Empty string*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

*Default = Empty string*

## **IsFileNameShort**

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## **LibExtension**

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .a*

## **LinkDebug**

The LinkDebug property specifies the special link switches used to link in debug mode.

*Default = -g*

## **LinkerFlags**

The LinkerFlags property allows you to define linker flags. The value of the property is inserted into the value of the LinkSwitches property. In the generated makefile, you can see the value of this property in the line that begins with LINK\_FLAGS=.

*Default = -lpthread -lstdc++*

## **LinkRelease**

The LinkRelease property specifies the special link switches used to link in release mode.

*Default = -O*

## **LinkSwitches**

The LinkSwitches property specifies the standard link switches used to link in any mode.

*Default = \$OMLinkCommandSet*

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration.

```
Default = ##### Target type (Debug/Release) #####
#####
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease
ConfigurationCPPCompileSwitches=$OMReusableStatechartSwitches
$OMConfigurationCPPCompileSwitches #####
Predefined macros ##### RM=/bin/rm -rf MD=/bin/mkdir -p INCLUDE_QUALIFIER=-I
CC=gcc -DUSE_Iostream LIB_CMD=ar LINK_CMD=$(CC) LIB_FLAGS=rvu
LINK_FLAGS=-lpthread -lstdc++ $OMConfigurationLinkSwitches
##### Context macros #####
$OMContextMacros ##### Predefined macros
##### $(OBJS) : $(INST_LIBS) $(OXF_LIBS) OBJ_DIR=$OMObjectsDir ifeq
$(OBJ_DIR,) CREATE_OBJ_DIR= CLEAN_OBJ_DIR= else CREATE_OBJ_DIR= $(MD) $(OBJ_DIR)
CLEAN_OBJ_DIR= $(RM) $(OBJ_DIR) endif ifeq $(INSTRUMENTATION),Animation)
INST_FLAGS=-DOMANIMATOR INST_INCLUDES=-I$(OMROOT)/LangCpp/aom
-I$(OMROOT)/LangCpp/tom INST_LIBS= $(OMROOT)/LangCpp/lib/linuxaomanim$(LIB_EXT)
OXF_LIBS=$(OMROOT)/LangCpp/lib/linuxoxfinst$(LIB_EXT)
$(OMROOT)/LangCpp/lib/linuxomcomappl$(LIB_EXT) SOCK_LIB= else ifeq
$(INSTRUMENTATION),Tracing) INST_FLAGS=-DOMTRACER
INST_INCLUDES=-I$(OMROOT)/LangCpp/aom -I$(OMROOT)/LangCpp/tom
INST_LIBS=$(OMROOT)/LangCpp/lib/linuxomtrace$(LIB_EXT)
$(OMROOT)/LangCpp/lib/linuxaomtrace$(LIB_EXT) OXF_LIBS=
$(OMROOT)/LangCpp/lib/linuxoxfinst$(LIB_EXT)
$(OMROOT)/LangCpp/lib/linuxomcomappl$(LIB_EXT) SOCK_LIB= else ifeq
$(INSTRUMENTATION),None) INST_FLAGS= INST_INCLUDES= INST_LIBS=
OXF_LIBS=$(OMROOT)/LangCpp/lib/linuxoxf$(LIB_EXT) SOCK_LIB= else @echo An invalid
Instrumentation $(INSTRUMENTATION) is specified. exit endif endif endif .SUFFIXES: $(CPP_EXT)
#####
##### Context dependencies and commands #####
$OMContextDependencies $OMFilePath : $OMMainImplementationFile $(OBJS) @$(CC)
```



```
$(ConfigurationCPPCompileSwitches) -o $OMFileObjPath $OMMainImplementationFile
##### #
Predefined linking instructions # # INST_LIBS is included twice to solve bi-directional dependency
between libraries #
##### #
$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS) $OMFileObjPath
$OMMakefileName $OMModelLibs @echo Linking $(TARGET_NAME)$(EXE_EXT) @$(LINK_CMD)
$OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \ $(OXF_LIBS) \
$(INST_LIBS) \ $(SOCK_LIB) \ $(LINK_FLAGS) -o $(TARGET_NAME)$(EXE_EXT)
$(TARGET_NAME)$(LIB_EXT) : $(OBJS) $(ADDITIONAL_OBJS) $OMMakefileName @echo Building
library $@ @$(LIB_CMD) $(LIB_FLAGS) $(TARGET_NAME)$(LIB_EXT) $(OBJS)
$(ADDITIONAL_OBJS) clean: @echo Cleanup $OMCleanOBJS $(RM) $OMFileObjPath
$(ADDITIONAL_OBJS) $(RM) $(TARGET_NAME)$(LIB_EXT) $(RM) $(TARGET_NAME)$(EXE_EXT)
$(CLEAN_OBJ_DIR)
```

## NoneIncludeDirectories

The NoneIncludeDirectories property is used to specify the directories that must be referenced in the makefile for projects where Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with INST\_INCLUDES.

*Default = Blank*

## NoneInstLibs

The NoneInstLibs property is used to specify the static libraries that are required when Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with INST\_LIBS.

*Default = Blank*

## NoneOxfLibs

The NoneOxfLibs property is used to specify the framework libraries that are required when Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with OXF\_LIBS.

*Default = \$(OMROOT)/LangCpp/lib/linuxoxf\$(LIB\_EXT)*

## NonePreprocessor

The NonePreprocessor property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with INST\_FLAGS.

*Default = Blank*

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## **ObjCleanCommand**

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default = \$(RM) \$OMFileObjPath*

## **ObjectName**

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = Empty string*

## **ObjectsDirectory**

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = Empty string*

## **ObjExtension**

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*Default = .o*

## **OSFileSystemCaseSensitive**

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Checked*

## **ParseErrorMessage**

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file

name and line number of errors reported by the compiler.

*Default = ([^:]+)([:])([0-9]+)([:])*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation.

*Default = /*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Checked*

## RemoteHost

*The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running the product, whereas the target is the machine running the application. To run remotely, the UseRemoteHost property must be set to True. If UseRemoteHost is True and RemoteHost is blank, the current host name is used for the remote host. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine. Default = Empty string*

## ReusableStatechartSwitches

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts. See the upgrade history on the support site for information on this change.

*Default = -DOM\_REUSABLE\_STATECHART\_IMPLEMENTATION*

## SockLib

The SockLib property represents the name of the socket library.

For the Cygwin environment, the value entered for this property is used in the generated makefile.

When using the integration with Visual Studio, the value of this property is included in the "additional dependencies" for the Visual Studio project.

*Default = Blank*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*Default = .h*

## TraceIncludeDirectories

The TraceIncludeDirectories property is used to specify the directories that must be referenced in the makefile for projects where Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with INST\_INCLUDES.

*Default = \$(INCLUDE\_QUALIFIER)\$(OMROOT)/LangCpp/aom  
\$(INCLUDE\_QUALIFIER)\$(OMROOT)/LangCpp/tom*

## TraceInstLibs

The TraceInstLibs property is used to specify the static libraries that are required when Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with INST\_LIBS.

*Default = \$(OMROOT)/LangCpp/lib/linuxomtrace\$(LIB\_EXT)  
\$(OMROOT)/LangCpp/lib/linuxaomtrace\$(LIB\_EXT)*

## TraceOxfLibs

The TraceOxfLibs property is used to specify the framework libraries that are required when Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with OXF\_LIBS.

*Default = \$(OMROOT)/LangCpp/lib/linuxoxfirst\$(LIB\_EXT)  
\$(OMROOT)/LangCpp/lib/linuxomcomappl\$(LIB\_EXT)*

## TracePreprocessor

The TracePreprocessor property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with INST\_FLAGS.

*Default = \$(DEFINE\_QUALIFIER)OMTRACER*

## UnixLineTerminationStyle

The UnixLineTerminationStyle property specifies whether generated files use the UNIX end-of-line style. If this property is set to Cleared, the end-of-line style depends on the host type (for example, DOS style on Windows machines, and UNIX style on Solaris machines).

*Default = Checked*

## UnixPathNameForOMROOT

The UnixPathNameForOMROOT property specifies whether the makefile must include UNIXstyle path names.

The pRISM compilers do not tolerate DOS-style path conventions. If you do not set this property correctly, there might be many compilation problems.

*Default = Checked*

## UseRemoteHost

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Checked*

## UseTemplateTypename

In the declarations for members of a template class, some compilers require that you use precede the member type with the keyword "typename" for types with dependent names. Since support for this keyword varies between compilers, the UseTemplateTypename property is used to specify whether the "typename" keyword should be included in the generated code.

*Default = Checked*

## Microsoft

The Microsoft metaclass contains environment properties (Compiler, framework libraries, and so on) for Microsoft Win32 compiler.

## AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path. Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files. To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)/LangCpp/osconfig/WIN32*

## **AdditionalReservedWords**

The AdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody. The property value is checked at run time when you name or rename an element, based on the active configuration environment setting. Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default =*

`__asm __finally naked __based __inline __single_inheritance __cdecl __int8 __stdcall __declspec __int16  
dllexport __int32 __try dllimport __int64 __virtual_inheritance __except __leave __fastcall  
__multiple_inheritance`

## **BuildArgumentsInIDE**

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the  
CPP\_CG::[environment]::BuildCommandInIDE property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## **BuildCommandInIDE**

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## **BuildCommandSet**

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration.

To change this property, use the Configuration window in the browser - do not change it by using the Properties tab in the Features window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link.

The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*Default = Debug*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\"\\etc\\msmake.bat msbuild.mak build  
\\\"USE\_STL=FALSE\" \"USE\_PDB=FALSE\" "*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## COM

The COM property is a boolean property that controls the value of the SUBSYSTEM linker option in the generated makefile.

*Default = False*

## CompilerFlags

The CompilerFlags property is used to specify additional compilation options. The value of this property is included in the value of the CPPCompileSwitches property.

When using the integration with Visual Studio, the options specified with this property are included in the settings for the Visual Studio project.

*Default = /GX*

## CompileSwitches

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property.

*Default =*

*/I . /I \$OMDefaultSpecificationDirectory /I \$(OMROOT)\LangCpp /I \$(OMROOT)\LangCpp\oxf /nologo /W3 /GX \$OMCPPCompileCommandSet /D "\_AFXDLL" /D "WIN32" /D "\_CONSOLE" /D "\_MBCS" /D "\_WINDOWS" \$(INST\_FLAGS) \$(INCLUDE\_PATH) \$(INST\_INCLUDES) /c*

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default =*

*\$(CREATE\_OBJ\_DIR) \$(CPP) \$OMFileCPPCompileSwitches /Fo"\$OMFileObjPath" "\$OMFileImpPath"*

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*Default = /Zi /Od /D "\_DEBUG" /MDd /Fd"\$(TARGET\_NAME)"*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*Default = /Ox /D"NDEBUG" /MD /Fd"\$(TARGET\_NAME)"*

## DefinedSymbols

The DefinedSymbols property is used to specify preprocessor directives. The value of this property is included in the value of the CPPCompileSwitches property.



When using the integration with Visual Studio, the preprocessor directives specified with this property are included in the preprocessor-related settings for the Visual Studio project.

*Default = \$(DEFINE\_QUALIFIER)\_CRT\_SECURE\_NO\_DEPRECATED*

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

*Default = \$OMFileObjPath : \$OMFileImpPath \$OMFileSpecPath \$OMFileDependencies*

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: `$OMFileObjPath : $OMFileImpPath "*.bmp" "*.ico" "*.rc2"`

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction with an IDE such as Eclipse, the EnableDebugIntegrationWithIDE property can be used to specify whether or not the IDE debugger should be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to True, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's exit() function is called automatically. The system function calls the destructors of framework instances in the order it determines. The EndApplicationCode property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an endApplication() function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to endApplication(), it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The EntryPoint property specifies the name of the main program for a given environment.

*Default = main*

See also the definition of the EntryPointDeclarationModifier property for more information.

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens has three parameters, each with an integer value:

- TotalNumberOfTokens - The number of tokens in the regular expression
- FileTokenPosition - The position of the file name token in the expression
- LineTokenPosition - The position of the line number token in the expression

*Default = TotalNumberOfTokens=3,FileTokenPosition=1,LineTokenPosition=2*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements. The file inclusions are generated in the makefile.

*Default = \$OMSpecIncludeInElements \$OMImpIncludeInElements*

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile. Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

The default is as follows:

```
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)WebComponents$(LIB_POSTFIX)$(LIB_EXT),  
$(OMROOT)\lib\$(LIB_PREFIX)WebServices$(LIB_POSTFIX)$(LIB_EXT), ws2_32$(LIB_EXT)
```

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment.

*Default = .cpp*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*Default = !INCLUDE*

## **IncludeDirectories**

For environments such as Cygwin, the IncludeDirectories property is used to specify additional directories to search for include files. The value of this property is included in the value of the CPPCompileSwitches property and is then included in the generated makefile.

When using the integration with Visual Studio, the directories specified with this property are added to the "additional include directories" specified for the Visual Studio project.

*Default = \$(INCLUDE\_QUALIFIER). \$(INCLUDE\_QUALIFIER)\$(OMROOT)  
\$(INCLUDE\_QUALIFIER)\$(OMROOT)\LangCpp \$(INCLUDE\_QUALIFIER)\$(OMROOT)\LangCpp\oxf*

## **InvokeAnimatedExecutable**

The AnimationPortRange value in the rhapsody.ini file makes it possible to use animation with multiple instances of Rhapsody running simultaneously.

Ordinarily, when you run an application with animation, the executable is launched using the command specified by the InvokeExecutable property. However, if you have specified a value greater than zero for AnimationPortRange in your rhapsody.ini file, the executable is launched using the command specified by the InvokeAnimatedExecutable property. By default, the value of InvokeAnimatedExecutable includes a reference to the port to use for animation.

*Default = \$executable -port \$port*

## **InvokeExecutable**

The InvokeExecutable property specifies the command used to run an executable file.

*Default = "\$executable"*

## **InvokeMake**

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

*"rhapsody\_dir>\share/etc/Executer.exe""dir; dir | more"*

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

*"\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\vx6make.bat\" \$makefile \$maketarget 6.2 \$BSP gnu"*

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\msmake.bat\" \$makefile \$maketarget"*

## **IsFileNameShort**

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format.

If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## **LibExtension**

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .lib*

## **LibPrefix**

The LibPrefix property is used to specify a prefix to add to generic names of runtime libraries. The value of this property is used in references to libraries in the generated makefile.

*Default = MS*

## **LinkDebug**

The LinkDebug property specifies the special link switches used to link in debug mode.

*Default = Empty string (blank)*

## **LinkRelease**

The LinkRelease property specifies the special link switches used to link in release mode.

Default value = Empty string

## **LinkSwitches**

The LinkSwitches property specifies the standard link switches used to link in any mode.

*Default = \$OMLinkCommandSet /NOLOGO*

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration.

```
Default = ##### Target type (Debug/Release) #####
#####
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease BuildSet=$OMBuildSet
SUBSYSTEM=$OMSubSystem COM=$OMCOM RPFFrameWorkDll=$OMRPFFrameWorkDll
SimulinkLibName=$SimulinkLibName
ConfigurationCPPCompileSwitches=$OMReusableStatechartSwitches
$OMConfigurationCPPCompileSwitches !IF "$(RPFFrameWorkDll)" == "True"
ConfigurationCPPCompileSwitches=$(ConfigurationCPPCompileSwitches) /D "FRAMEWORK_DLL"
!ENDIF !IF "$(COM)" == "True" SUBSYSTEM=/SUBSYSTEM:windows !ENDIF
##### Compilation flags #####
##### INCLUDE_QUALIFIER=/I
LIB_PREFIX=MS ##### Commands definition #####
##### RMDIR = rmdir
LIB_CMD=link.exe -lib LINK_CMD=link.exe LIB_FLAGS=$OMConfigurationLinkSwitches
LINK_FLAGS=$OMConfigurationLinkSwitches $(SUBSYSTEM) /MACHINE:I386 #####
Generated macros #####
$OMContextMacros OBJ_DIR=$OMObjectsDir !IF "$(OBJ_DIR)" != "" CREATE_OBJ_DIR=if not exist
$(OBJ_DIR) mkdir $(OBJ_DIR) CLEAN_OBJ_DIR= if exist $(OBJ_DIR) $(RMDIR) $(OBJ_DIR) !ELSE
CREATE_OBJ_DIR= CLEAN_OBJ_DIR= !ENDIF ##### Predefined macros
#####
##### $(OBJS) :
$(INST_LIBS) $(OXF_LIBS) LIB_POSTFIX= !IF "$(BuildSet)" == "Release" LIB_POSTFIX=R !ENDIF
!IF "$(TARGET_TYPE)" == "Executable" LinkDebug=$(LinkDebug) /DEBUG
LinkRelease=$(LinkRelease) /OPT:NOREF !ELSEIF "$(TARGET_TYPE)" == "Library"
LinkDebug=$(LinkDebug) /DEBUGTYPE:CV !ENDIF !IF "$(INSTRUMENTATION)" == "Animation"
INST_FLAGS=/D "OMANIMATOR" INST_INCLUDES=/I $(OMROOT)\LangCpp\lom /I
$(OMROOT)\LangCpp\lom !IF "$(RPFFrameWorkDll)" == "True" INST_LIBS=
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfanimdll$(LIB_POSTFIX)$(LIB_EXT) !ELSE
INST_LIBS= $(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomanim$(LIB_POSTFIX)$(LIB_EXT)
```

```

OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfinst$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX)$(LIB_EXT)
$(SimulinkLibName) !ENDIF SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)" ==
"Tracing" INST_FLAGS=/D "OMTRACER" INST_INCLUDES=/I $(OMROOT)\LangCpp\iom /I
$(OMROOT)\LangCpp\iom !IF "$(RPFrameWorkDll)" == "True" INST_LIBS= OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxftracedll$(LIB_POSTFIX)$(LIB_EXT) !ELSE
INST_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)iomtrace$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)iomtrace$(LIB_POSTFIX)$(LIB_EXT) OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfinst$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX)$(LIB_EXT)
$(SimulinkLibName) !ENDIF SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)" == "None"
INST_FLAGS= INST_INCLUDES= INST_LIBS= !IF "$(RPFrameWorkDll)" == "True"
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfdll$(LIB_POSTFIX)$(LIB_EXT) !ELSE
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$(LIB_POSTFIX)$(LIB_EXT)
$(SimulinkLibName) !ENDIF SOCK_LIB= !ELSE !ERROR An invalid Instrumentation
$(INSTRUMENTATION) is specified. !ENDIF ##### Generated dependencies
#####
#####
$OMContextDependencies $OMFileObjPath : $OMMainImplementationFile $(OBJS) $(CPP)
$(ConfigurationCPPCompileSwitches) /Fo"$OMFileObjPath" $OMMainImplementationFile
##### Linking instructions #####
#####
$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS) $OMFileObjPath
$OMMakefileName $OMModelLibs @echo Linking $(TARGET_NAME)$(EXE_EXT) $(LINK_CMD)
$OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \ $(OXF_LIBS) \
$(SOCK_LIB) \ $(LINK_FLAGS) /out:$(TARGET_NAME)$(EXE_EXT) $(TARGET_NAME)$(LIB_EXT) :
$(OBJS) $(ADDITIONAL_OBJS) $OMMakefileName @echo Building library $@ $(LIB_CMD)
$(LIB_FLAGS) /out:$(TARGET_NAME)$(LIB_EXT) $(OBJS) $(ADDITIONAL_OBJS) clean: @echo
Cleanup $OMCleanOBJS if exist $OMFileObjPath erase $OMFileObjPath if exist *$(OBJ_EXT) erase
*$(OBJ_EXT) if exist $(TARGET_NAME).pdb erase $(TARGET_NAME).pdb if exist
$(TARGET_NAME)$(LIB_EXT) erase $(TARGET_NAME)$(LIB_EXT) if exist $(TARGET_NAME).ilk
erase $(TARGET_NAME).ilk if exist $(TARGET_NAME)$(EXE_EXT) erase
$(TARGET_NAME)$(EXE_EXT) $(CLEAN_OBJ_DIR)

```

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## ObjCleanCommand

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default = if exist \$OMFileObjPath erase \$OMFileObjPath*

## ObjectName

The ObjectName property specifies an alternative name for the compiled object file in the generated

makefile.

*Default = Empty string*

## ObjectsDirectory

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = Empty string*

## ObjExtension

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*Default = .obj*

## OSFileSystemCaseSensitive

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Cleared*

## ParseErrorDescript

The ParseErrorDescript property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = ([^()]+)[([0-9]+)] [:] (error/warning/fatal error) (.\*)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = ([^()]+)[([0-9]+)] [:] (error/warning/fatal error)*

## ParseErrorMoreInfo

The ParseErrorMoreInfo property is used to define a regular expression that represents the format of compiler error messages that follow the first line which contains the term ‘error’ or ‘warning’.

This property is used to parse the compiler output and extract the additional information from the message so that this information can be displayed to the user in the build output window.

*Default = ^[( )]*

## **ParseMakeError**

The ParseMakeError property is used to define a regular expression that represents the format of make process or linker error messages. This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = (NMAKE|LINK)(.\*)(fatal error)*

## **ParseSeverityError**

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = [:] (error|fatal error)*

## **ParseSeverityWarning**

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = [:] (warning)*

## **PathDelimiter**

The PathDelimiter property specifies an alternative path separator for code generation.

*Default = \*

## **QuoteOMROOT**

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Checked*



## RCCompileCommand

The RCCompileCommand property is a string that specifies the compilation command for the resource file. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = \$(RC) /Fo"\$(TARGET\_MAIN).res" \$(TARGET\_MAIN)\$OMRCEExtension*

## RCEExtension

The RCEExtension property is a string that specifies the extension for resource files. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = .rc*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely.

In such configurations, the host is the machine running Rational Rhapsody, whereas the target is the machine running the application.

To run remotely, the UseRemoteHost property must be set to Checked. If UseRemoteHost is Checked and RemoteHost is blank, the current host name is used for the remote host. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = Empty string*

## ReusableStatechartSwitches

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts. See the upgrade history on the support site for information on this change.

*Default = /D "OM\_REUSABLE\_STATECHART\_IMPLEMENTATION"*

## RPFrameWorkDll

The RPFrameWorkDll property can be used to specify that an application should use the dll generated for the OXF framework.

When the property is set to True, the makefile that is generated for the application includes linkage to the relevant OXF framework dll, based on the instrumentation mode specified for the configuration: oxfanimdll.dll if animation is used, oxfracedll.dll if tracing is used, and oxfdll.dll if the configuration does not use animation or tracing.

*Default = False*

## **SockLib**

The SockLib property represents the name of the socket library.

For the Cygwin environment, the value entered for this property is used in the generated makefile.

When using the integration with Visual Studio, the value of this property is included in the "additional dependencies" for the Visual Studio project.

*Default = wsock32.lib*

## **SpecExtension**

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*Default = .h*

## **SubSystem**

The SubSystem property is a string that defines the type of the program for the Microsoft linker. The possible values are as follows:

- **CONSOLE** - Used for a Win32 character-mode application
- **WINDOWS** - Used for an application that does not require a console
- **NATIVE** - Applies device drivers for Windows NT
- **POSIX** - Creates an application that runs with the POSIX subsystem in Windows NT

*Default = /SUBSYSTEM:console*

## **UpdateBuildSettingsInIDE**

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## **UseNewBuildOutputWindow**

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log

tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the `CG::General::ShowLogViewAfterBuild` property to `True`.

*Default = Checked*

## UseNonZeroStdInputHandle

The `UseNonZeroStdInputHandle` property is a Boolean value that specifies whether to use a non-zero standard input handle.

*Default = Checked*

## UseRemoteHost

The `UseRemoteHost` property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Cleared*

## UseTemplateName

In the declarations for members of a template class, some compilers require that you use precede the member type with the keyword "typename" for types with dependent names. Since support for this keyword varies between compilers, the `UseTemplateName` property is used to specify whether the "typename" keyword should be included in the generated code.

*Default = Cleared*

## MicrosoftDLL

The `MicrosoftDLL` metaclass contains environment properties (Compiler, framework libraries, and so on) for Microsoft Win32 compiler that creates DLLs instead of static libraries.

## AdaptorSearchPath

The `AdaptorSearchPath` property specifies the path to the operating system configuration file. This path is added to the generated makefile search path. Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files. To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)/LangCpp/osconfig/WIN32*

## AdditionalReservedWords

The AdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody. The property value is checked at run time when you name or rename an element, based on the active configuration environment setting. Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default =*

`__asm __finally naked __based __inline __single_inheritance __cdecl __int8 __stdcall __declspec __int16  
dllexport __int32 __try dllimport __int64 __virtual_inheritance __except __leave __fastcall  
__multiple_inheritance`

## BuildArgumentsInIDE

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the  
CPP\_CG::[environment]::BuildCommandInIDE property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandInIDE

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration.

To change this property, use the Configuration window in the browser - do not change it by using the Properties tab in the Features window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link.

The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*Default = Debug*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\"\\etc\\msmake.bat msbuild.mak build  
\\\"USE\_STL=FALSE\" \"\\\"USE\_PDB=FALSE\" \" \"*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## COM

The COM property is a boolean property that controls the value of the SUBSYSTEM linker option in the generated makefile.

*Default = False*

## CompileSwitches

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property.

*Default =*

```
/I . /I $OMDefaultSpecificationDirectory /I $(OMROOT)\LangCpp /I $(OMROOT)\LangCpp\oxf /nologo  
/W3 /GX $OMCPPCompileCommandSet /D "_AFXDLL" /D "WIN32" /D "_CONSOLE" /D "_MBCS"  
/D "_WINDOWS" $(INST_FLAGS) $(INCLUDE_PATH) $(INST_INCLUDES) /c
```

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default =*

```
$(CREATE_OBJ_DIR) $(CPP) $OMFileCPPCompileSwitches /Fo"$OMFileObjPath"  
"$OMFileImpPath"
```

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*Default = /Zi /Od /D "\_DEBUG" /MDd /Fd"\$(TARGET\_NAME)"*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*Default = /Ox /D"NDEBUG" /MD /Fd"\$(TARGET\_NAME)"*

## DEFExtension

The DEFExtension property is a string that specifies the extension for DLL definition files. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = .def*

## DependencyRule

The `DependencyRule` property specifies how file dependencies for a configuration are generated in the makefile.

*Default = \$OMFileObjPath : \$OMFileImpPath \$OMFileSpecPath \$OMFileDependencies*

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: `$OMFileObjPath : $OMFileImpPath "*.bmp" "*.ico" "*.rc2"`

## DllExtension

The `DllExtension` property is a string that specifies the extension for DLL files. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = .dll*

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction with an IDE such as Eclipse, the `EnableDebugIntegrationWithIDE` property can be used to specify whether or not the IDE debugger should be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to `True`, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's `exit()` function is called automatically. The system function calls the destructors of framework instances in the order it determines. The `EndApplicationCode` property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an `endApplication()` function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to `endApplication()`, it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The `EntryPoint` property specifies the name of the main program for a given environment.

*Default = main*

See also the definition of the EntryPointDeclarationModifier property for more information.

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens has three parameters, each with an integer value:

- TotalNumberOfTokens - The number of tokens in the regular expression
- FileTokenPosition - The position of the file name token in the expression
- LineTokenPosition - The position of the line number token in the expression

*Default = TotalNumberOfTokens=3,FileTokenPosition=1,LineTokenPosition=2*

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the CPP\_CG::<Environment>::ExeName property plus the value of this property.

*(Default = .exe)*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the CPP\_CG::<Environment>::ExeExtension property.

*(Default = Blank)*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements. The file inclusions are generated in the makefile.

*Default = \$OMSpecIncludeInElements \$OMImpIncludeInElements*



## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment.

*Default = .cpp*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*Default = !INCLUDE*

## InvokeAnimatedExecutable

The AnimationPortRange value in the rhapsody.ini file makes it possible to use animation with multiple instances of Rhapsody running simultaneously.

Ordinarily, when you run an application with animation, the executable is launched using the command specified by the InvokeExecutable property. However, if you have specified a value greater than zero for AnimationPortRange in your rhapsody.ini file, the executable is launched using the command specified by the InvokeAnimatedExecutable property. By default, the value of InvokeAnimatedExecutable includes a reference to the port to use for animation.

*Default = \$executable -port \$port*

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*Default = "\$executable"*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\msmake.bat\" \$makefile \$maketarget"*

## **IsFileNameShort**

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format.

If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## **LibExtension**

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .dll*

## **LinkDebug**

The LinkDebug property specifies the special link switches used to link in debug mode.

*Default = Empty string (blank)*

## **LinkRelease**

The LinkRelease property specifies the special link switches used to link in release mode.

Default value = Empty string

## **LinkSwitches**

The LinkSwitches property specifies the standard link switches used to link in any mode.

*Default = \$OMLinkCommandSet /NOLOGO*

## **MakeExtension**

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the

CPP\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

(Default = .mak)

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration.

```
Default = ##### Target type (Debug/Release) #####
#####
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease BuildSet=$OMBuildSet
SUBSYSTEM=$OMSubSystem COM=$OMCOM RPFFrameWorkDll=$OMRPFFrameWorkDll
DEF_EXT=$OMDEFExtension DLL_EXT=$OMDllExtension
ConfigurationCPPCompileSwitches=$OMReusableStatechartSwitches
$OMConfigurationCPPCompileSwitches !IF "$(RPFFrameWorkDll)" == "True"
ConfigurationCPPCompileSwitches=$(ConfigurationCPPCompileSwitches) /D "FRAMEWORK_DLL"
!ENDIF !IF "$(COM)" == "True" SUBSYSTEM=/SUBSYSTEM:windows !ENDIF
##### Compilation flags #####
##### INCLUDE_QUALIFIER=/I
LIB_PREFIX=MS ##### Commands definition #####
##### RMDIR = rmdir
DLL_CMD=link.exe -dll LINK_CMD=link.exe DLL_FLAGS=$OMConfigurationLinkSwitches
LINK_FLAGS=$OMConfigurationLinkSwitches $(SUBSYSTEM) /MACHINE:I386 #####
Generated macros #####
$OMContextMacros OBJ_DIR=$OMObjectsDir !IF "$(OBJ_DIR)" != "" CREATE_OBJ_DIR=if not exist
$(OBJ_DIR) mkdir $(OBJ_DIR) CLEAN_OBJ_DIR= if exist $(OBJ_DIR) $(RMDIR) $(OBJ_DIR) !ELSE
CREATE_OBJ_DIR= CLEAN_OBJ_DIR= !ENDIF ##### Predefined macros
#####
##### !IF "$(OBS)"
!= "" $(OBS) : $(FLAGSFILE) $(RULESFILE) $(INST_LIBS) $(OXF_LIBS) !ENDIF LIB_EXT=.lib
LIB_POSTFIX= !IF "$(BuildSet)"=="Release" LIB_POSTFIX=R !ENDIF !IF "$(TARGET_TYPE)" ==
"Executable" LinkDebug=$(LinkDebug) /DEBUG LinkRelease=$(LinkRelease) /OPT:NOREF !ELSEIF
```

```

"$(TARGET_TYPE)" == "Library" LinkDebug=$(LinkDebug) /DEBUG /DEBUGTYPE:CV
LinkRelease=$(LinkRelease) /OPT:NOREF !ENDIF !IF "$(TIME_MODEL)" == "Simulated"
TIM_EXT= !ELSEIF "$(TIME_MODEL)" == "RealTime" TIM_EXT= !ELSE !ERROR An invalid Time
Model "$(TIME_MODEL)" is specified. !ENDIF !IF "$(INSTRUMENTATION)" == "Animation"
INST_FLAGS=/D "OMANIMATOR" INST_INCLUDES=/I $(OMROOT)\LangCpp\om /I
$(OMROOT)\LangCpp\om !IF "$(RPFrameWorkDll)" == "True" INST_LIBS=
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfanimdll$(LIB_POSTFIX)$(LIB_EXT) !ELSE
INST_LIBS= $(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomanim$(LIB_POSTFIX)$(LIB_EXT)
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$(TIM_EXT)inst$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX)$(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)" == "Tracing" INST_FLAGS=/D
"OMTRACER" INST_INCLUDES=/I $(OMROOT)\LangCpp\om /I $(OMROOT)\LangCpp\om !IF
"$(RPFrameWorkDll)" == "True" INST_LIBS= OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxftracedll$(LIB_POSTFIX)$(LIB_EXT) !ELSE
INST_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)tomtrace$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomtrace$(LIB_POSTFIX)$(LIB_EXT) OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$(TIM_EXT)inst$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX)$(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)" == "None" INST_FLAGS=
INST_INCLUDES= INST_LIBS= !IF "$(RPFrameWorkDll)" == "True"
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfdll$(LIB_POSTFIX)$(LIB_EXT) !ELSE
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$(TIM_EXT)$(LIB_POSTFIX)$(LIB_EXT)
!ENDIF SOCK_LIB= !ELSE !ERROR An invalid Instrumentation $(INSTRUMENTATION) is specified.
!ENDIF !IF "$(COM)" == "True" COM_LIB=kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib
advapi32.lib shell32.lib ole32.lib oleaut32.lib uuid.lib odbcc32.lib odbccp32.lib
COM_OBJS=$OMFileObjPath DEF_NAME=$(TARGET_MAIN)$(DEF_EXT)
LINK_DEF=/def:$(DEF_NAME) !ELSE COM_LIB= COM_OBJS= DEF_NAME= LINK_DEF= !ENDIF
##### Generated dependencies #####
#####
$OMContextDependencies !IF "$(TARGET_MAIN)" != "" CLEAN_MAIN_OBJ=if exist $OMFileObjPath
erase $OMFileObjPath $OMFileObjPath : $OMMainImplementationFile $(OBJS) $(FLAGSFILE)
$(RULESFILE) $(CPP) $(ConfigurationCPPCompileSwitches) /Fo"$OMFileObjPath"
$OMMainImplementationFile !ELSE CLEAN_MAIN_OBJ= !ENDIF #####
Linking instructions #####
##### !IF
"$(TARGET_NAME)" != "" $(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS)
$OMFileObjPath $OMMakefileName $OMModelLibs @echo Linking $(TARGET_NAME)$(EXE_EXT)
$(LINK_CMD) $OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \
$(OXF_LIBS) \ $(SOCK_LIB) \ $(LINK_FLAGS) /out:$(TARGET_NAME)$(EXE_EXT)
$(TARGET_NAME)$(DLL_EXT) : $(OBJS) $(ADDITIONAL_OBJS) $(COM_OBJS) $(DEF_NAME)
$OMMakefileName @echo Building library @$ $(DLL_CMD) $(DLL_FLAGS) $(COM_LIB) $(OBJS)
$(COM_OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \ $(OXF_LIBS) \ $(SOCK_LIB) \
$(LINK_DEF) \ /out:$(TARGET_NAME)$(DLL_EXT) !ENDIF clean: @echo Cleanup $OMCleanOBJS
$(CLEAN_MAIN_OBJ) if exist *$(OBJ_EXT) erase *$(OBJ_EXT) if exist $(TARGET_NAME).pdb erase
$(TARGET_NAME).pdb if exist $(TARGET_NAME)$(LIB_EXT) erase $(TARGET_NAME)$(LIB_EXT) if
exist $(TARGET_NAME).ilk erase $(TARGET_NAME).ilk if exist $(TARGET_NAME)$(EXE_EXT) erase
$(TARGET_NAME)$(EXE_EXT) $(CLEAN_OBJ_DIR)

```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the CPP\_CG::<Environment>::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## **NullValue**

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## **ObjCleanCommand**

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default = if exist \$OMFileObjPath erase \$OMFileObjPath*

## **ObjectName**

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = Empty string*

## **ObjectsDirectory**

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = Empty string*

## **ObjExtension**

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*Default = .obj*

## **OSFileSystemCaseSensitive**

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Cleared*

## ParseErrorDescript

The ParseErrorDescript property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = ([^()]+)[([0-9]+)] [:] (error/warning/fatal error) (.\*)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = ([^()]+)[([0-9]+)] [:] (error/warning/fatal error)*

## ParseMakeError

The ParseMakeError property is used to define a regular expression that represents the format of make process or linker error messages. This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = (NMAKE)(.\*)(fatal error)*

## ParseSeverityError

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = [:] (error/fatal error)*

## ParseSeverityWarning

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = [:] (warning)*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation.

*Default = \*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Checked*

## RCCompileCommand

The RCCompileCommand property is a string that specifies the compilation command for the resource file. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = \$(RC) /Fo"\$(TARGET\_MAIN).res" \$(TARGET\_MAIN)\$OMRCEExtension*

## RCExtension

The RCExtension property is a string that specifies the extension for resource files. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = .rc*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely.

In such configurations, the host is the machine running Rational Rhapsody, whereas the target is the machine running the application.

To run remotely, the UseRemoteHost property must be set to Checked. If UseRemoteHost is Checked and RemoteHost is blank, the current host name is used for the remote host. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = Empty string*

## ReusableStatechartSwitches

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts. See the upgrade history on the support site for information on this change.

*Default = /D "OM\_REUSABLE\_STATECHART\_IMPLEMENTATION"*

## RPFrameWorkDll

The RPFrameWorkDll property can be used to specify that an application should use the dll generated for the OXF framework.

When the property is set to True, the makefile that is generated for the application includes linkage to the relevant OXF framework dll, based on the instrumentation mode specified for the configuration: oxfanimdll.dll if animation is used, oxfracedll.dll if tracing is used, and oxfdll.dll if the configuration does not use animation or tracing.

*Default = False*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*Default = .h*

## SubSystem

The SubSystem property is a string that defines the type of the program for the Microsoft linker. The possible values are as follows:

- **CONSOLE** - Used for a Win32 character-mode application
- **WINDOWS** - Used for an application that does not require a console
- **NATIVE** - Applies device drivers for Windows NT
- **POSIX** - Creates an application that runs with the POSIX subsystem in Windows NT

*Default = /SUBSYSTEM:console*

## UpdateBuildSettingsInIDE

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## UseNewBuildOutputWindow

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.



This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the `CG::General::ShowLogViewAfterBuild` property to `True`.

*Default = Checked*

## UseNonZeroStdInputHandle

The `UseNonZeroStdInputHandle` property is a Boolean value that specifies whether to use a non-zero standard input handle.

*Default = Checked*

## UseRemoteHost

The `UseRemoteHost` property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Cleared*

## UseTemplateName

In the declarations for members of a template class, some compilers require that you use precede the member type with the keyword "typename" for types with dependent names. Since support for this keyword varies between compilers, the `UseTemplateName` property is used to specify whether the "typename" keyword should be included in the generated code.

*Default = Cleared*

# MicrosoftWinCE600

The `MicrosoftWinCE600` metaclass contains environment properties (Compiler, framework libraries, and so on) for `MicrosoftWinCE600` compiler.

## AdaptorSearchPath

The `AdaptorSearchPath` property specifies the path to the operating system configuration file. This path is added to the generated makefile search path. Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files. To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.

- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)/LangCpp/osconfig/WIN32*

## BuildArgumentsInIDE

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the CPP\_CG::[environment]::BuildCommandInIDE property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandInIDE

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration.

To change this property, use the Configuration window in the browser - do not change it by using the Properties tab in the Features window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link. The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*Default = Debug*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\"\\etc\\mscemake.bat mscebuild.mak build \$CPU \\\"BUILD\_SET=\$BuildCommandSet\\\" "*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## CompileSwitches

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property.

*Default =*

*/I . /I \$(OMROOT)\LangCpp /I \$(OMROOT)\LangCpp\oxf /nologo /W3 /GX- /D  
\_WIN32\_WCE=\$(CEVersion) /D "\$(CEConfigName)" \$(MACHINE\_CPP\_FLAGS) /D  
"\_OM\_NO\_IOSTREAM" /D UNDER\_CE=\$(CEVersion) /D "UNICODE" /D  
"\_OM\_UNICODE\_ONLY" \$OMCPPCompileCommandSet /D "\_AFXDLL" /D "WIN32"  
\$(INST\_FLAGS) \$(INCLUDE\_PATH) \$(INST\_INCLUDES) /D "\_X86\_" /c*

## CPPAdditionalReservedWords

The CPPAdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use.

In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody. The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

*Default =*

`__asm __finally naked __based __inline __single_inheritance __cdecl __int8 __stdcall __declspec __int16  
dllexport __int32 __try dllimport __int64 __virtual_inheritance __except __leave __fastcall  
__multiple_inheritance`

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default =*

`$(CPP) $OMFileCPPCompileSwitches /Fo"$OMFileObjPath" "$OMFileImpPath"`

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*Default =* `/Zi /Od /D "_DEBUG" /M$(CECrtMTDebug) /Fd"$(TARGET_NAME)"`

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*Default =* `/Ox /D"NDEBUG" /M$(CECrtMT) /Fd"$(TARGET_NAME)`

## CPU

The CPU property is a string that specifies the CPU type.

*Default =* `x86`

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

*Default =* `$OMFileObjPath : $OMFileImpPath $OMFileSpecPath $OMFileDependencies`

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: `$OMFileObjPath : $OMFileImpPath "*.bmp" "*.ico" "*.rc2"`

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction with an IDE such as Eclipse, the `EnableDebugIntegrationWithIDE` property can be used to specify whether or not the IDE debugger should be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to `True`, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's `exit()` function is called automatically. The system function calls the destructors of framework instances in the order it determines. The `EndApplicationCode` property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an `endApplication()` function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to `endApplication()`, it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The `EntryPoint` property specifies the name of the main program for a given environment.

*Default = cemain*

See also the definition of the `EntryPointDeclarationModifier` property for more information.

## ErrorMessageTokensFormat

The `ErrorMessageTokensFormat`, working with the `ParseErrorMessage` property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. `ErrorMessageTokensFormat` defines the number and location of tokens within the regular expression defined by the `ParseErrorMessage` property. `ErrorMessageTokens` has three parameters, each with an integer value:

- `TotalNumberOfTokens` - The number of tokens in the regular expression
- `FileTokenPosition` - The position of the file name token in the expression
- `LineTokenPosition` - The position of the line number token in the expression

*Default = ToTalNumberOfTokens=3,FileTokenPosition=1,LineTokenPosition=2*

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the CPP\_CG::<Environment>::ExeName property plus the value of this property.

*(Default = .exe)*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the CPP\_CG::<Environment>::ExeExtension property.

*(Default = Blank)*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements. The file inclusions are generated in the makefile.

*Default = \$OMSpecIncludeInElements \$OMImpIncludeInElements*

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile. Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

*Default =*

*\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)WebComponents\$(LIB\_POSTFIX)\$(LIB\_EXT),  
\$(OMROOT)\lib\\$(LIB\_PREFIX)WebServices\$(LIB\_POSTFIX)\$(LIB\_EXT), winsock.lib*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment.

*Default = .cpp*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*Default = !INCLUDE*

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*Default = Empty string*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\msceNETmake.bat\" \$makefile \$maketarget x86"*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format.

If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .lib*

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

*Default = Empty string (blank)*

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

Default value = Empty string

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

*Default = \$OMLinkCommandSet /NOLOGO*

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the CPP\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*(Default = .mak)*

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags



- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration.

```

Default = USE_MFC_APP_WINDOW=FALSE ##### Target type (Debug/Release)
#####
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease BuildSet=$OMBuildSet
ConfigurationCPPCompileSwitches=$OMReusableStatechartSwitches
$OMConfigurationCPPCompileSwitches !IF "$(OSVERSION)" == "WCE400"
CESubsystem=windowsce,4.00 CEVersion=400 CEConfigName=OPPS !ELSEIF "$(OSVERSION)" ==
"WCE420" CESubsystem=windowsce,4.20 CEVersion=420 CEConfigName=OPPS !ELSE !MESSAGE
An invalid OSVERSION "$(OSVERSION)" is specified. !MESSAGE Please specify OSVERSION=
WCE400 or WCE420 !ERROR Exiting !ENDIF CECrtMT=T CECrtMTDebug=Td
CENoDefaultLib=libc.lib /nodefaultlib:libcd.lib /nodefaultlib:libcmtd.lib /nodefaultlib:libcmtd.lib
/nodefaultlib:msvcrt.lib /nodefaultlib:msvcrt.lib /nodefaultlib:OldNames.lib CECorelibc=corelibc.lib !IF
"$(MACHINE)" == "SH3" CPP=shcl.exe MACHINE_CPP_FLAGS=/D "SHx" /D "SH3" /D "_SH3_"
MACHINE_EXT=SH !ELSEIF "$(MACHINE)" == "SH4" CPP=shcl.exe
MACHINE_CPP_FLAGS=/Qsh4 /D "SHx" /D "SH4" /D "_SH4_" MACHINE_EXT=SH !ELSEIF
"$(MACHINE)" == "MIPS" CPP=clmips.exe MACHINE_CPP_FLAGS=/D "MIPS" /D "_MIPS_"
MACHINE_EXT=MIPS !ELSEIF "$(MACHINE)" == "ARM" CPP=clarm.exe
MACHINE_CPP_FLAGS=/D "ARM" /D "_ARM_" MACHINE_EXT=PPC !ELSEIF "$(MACHINE)" ==
"IX86" CPP=cl.exe MACHINE_CPP_FLAGS=/D "x86" /D "_i386_" /D "_x86_" /D "i_386_"
MACHINE_EXT=IX86 !ELSE !MESSAGE An invalid MACHINE "$(MACHINE)" is specified.
!MESSAGE Please specify MACHINE= SH3 SH4 MIPS ARM or IX86 !ERROR Exiting !ENDIF
##### Compilation flags #####
##### INCLUDE_QUALIFIER=/I
LIB_PREFIX=Ce$(CEVersion)$(TARGETCPU) ##### Commands definition
#####
##### LIB_CMD=link.exe -lib
LINK_CMD=link.exe LIB_FLAGS=$OMConfigurationLinkSwitches
LINK_FLAGS=$OMConfigurationLinkSwitches $(CECorelibc) commctrl.lib coredll.lib
/SUBSYSTEM:$(CESubsystem) /MACHINE:$(MACHINE) /nodefaultlib:$(CENoDefaultLib)
##### Generated macros #####
##### $OMContextMacros
##### Predefined macros #####
##### $(OBS) :
$(FLAGSFILE) $(RULESFILE) $(INST_LIBS) $(OXF_LIBS) LIB_POSTFIX= !IF
"$(BuildSet)"=="Release" LIB_POSTFIX=R !ENDIF !IF "$(TARGET_TYPE)" == "Executable"
LinkDebug=$(LinkDebug) /DEBUG LinkRelease=$(LinkRelease) /OPT:NOREF !ELSEIF
"$(TARGET_TYPE)" == "Library" LinkDebug=$(LinkDebug) /DEBUGTYPE:CV !ENDIF !IF
"$(INSTRUMENTATION)" == "Animation" INST_FLAGS=/D "OMANIMATOR" INST_INCLUDES=/I
$(OMROOT)\LangCpp\iom /I $(OMROOT)\LangCpp\tom INST_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)ioanim$(LIB_POSTFIX)$(LIB_EXT)
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfinst$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComApp$(LIB_POSTFIX)$(LIB_EXT)

```

```

SOCK_LIB=winsock.lib !ELSEIF "$(INSTRUMENTATION)" == "Tracing" INST_FLAGS=/D
"OMTRACER" INST_INCLUDES=/I $(OMROOT)\LangCpp\iom /I $(OMROOT)\LangCpp\tom
INST_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)tomtrace$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)iomtrace$(LIB_POSTFIX)$(LIB_EXT) OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfirst$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)iomComAppl$(LIB_POSTFIX)$(LIB_EXT)
SOCK_LIB=winsock.lib !ELSEIF "$(INSTRUMENTATION)" == "None" INST_FLAGS=
INST_INCLUDES= INST_LIBS=
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$(TIM_EXT)$(LIB_POSTFIX)$(LIB_EXT)
SOCK_LIB= !ELSE !ERROR An invalid Instrumentation $(INSTRUMENTATION) is specified. !ENDIF
##### Generated dependencies #####
#####
$OMContextDependencies $(TARGET_MAIN)$(OBJ_EXT) : $(TARGET_MAIN)$(CPP_EXT) $(OBJS)
$(FLAGSFILE) $(RULESFILE) $(CPP) $(ConfigurationCPPCompileSwitches) /Fo"$OMFileObjPath"
$(TARGET_MAIN)$(CPP_EXT) !IF "$(USE_MFC_APP_WINDOW)"=="TRUE" CE_APP_FLAGS=/D
USE_MFC_APP_WINDOW MAIN_ENTRY_NAME=wWinMainCRTStartup !ELSE
MAIN_ENTRY_NAME=wWinMain CE_APP_FLAGS= !ENDIF MsCeApp$(CPP_EXT) : @echo Copying
MsCeApp$(CPP_EXT) @copy $(OMROOT)\MakeTmp\MsCeApp$(CPP_EXT) MsCeApp$(CPP_EXT)
MsCeApp$(OBJ_EXT) : MsCeApp$(CPP_EXT) $(CPP) $(CE_APP_FLAGS)
$(ConfigurationCPPCompileSwitches) MsCeApp$(CPP_EXT) ##### Linking
instructions #####
#####
$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS) $(TARGET_MAIN)$(OBJ_EXT)
MsCeApp$(OBJ_EXT) $OMMakefileName $OMModelLibs @echo Linking
$(TARGET_NAME)$(EXE_EXT) $(LINK_CMD) $(TARGET_MAIN)$(OBJ_EXT) MsCeApp$(OBJ_EXT)
/entry:"$(MAIN_ENTRY_NAME)" /base:"0x00010000" $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \
$(INST_LIBS) \ $(OXF_LIBS) \ $(SOCK_LIB) \ $(LINK_FLAGS) /out:$(TARGET_NAME)$(EXE_EXT)
$(TARGET_NAME)$(LIB_EXT) : $(OBJS) $(ADDITIONAL_OBJS) $OMMakefileName @echo Building
library @@ $(LIB_CMD) $(LIB_FLAGS) /out:$(TARGET_NAME)$(LIB_EXT) $(OBJS)
$(ADDITIONAL_OBJS) clean: @echo Cleanup $OMCleanOBJS if exist *$(OBJ_EXT) erase
*$(OBJ_EXT) if exist $(TARGET_NAME).pdb erase $(TARGET_NAME).pdb if exist
$(TARGET_NAME)$(LIB_EXT) erase $(TARGET_NAME)$(LIB_EXT) if exist $(TARGET_NAME).ilk
erase $(TARGET_NAME).ilk if exist $(TARGET_NAME)$(EXE_EXT) erase
$(TARGET_NAME)$(EXE_EXT)

```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the CPP\_CG::<Environment>::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

(Default = Blank)

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## **ObjCleanCommand**

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default = if exist \$OMFileObjPath erase \$OMFileObjPath*

## **ObjectName**

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = Empty string*

## **ObjectsDirectory**

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = Empty string*

## **ObjExtension**

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*Default = .obj*

## **OSFileSystemCaseSensitive**

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Cleared*

## **ParseErrorMessage**

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = ([^()]+)([()([0-9]+)]\*)[:] (error/warning/fatal error)*

## **QuoteOMROOT**

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Checked*

## **RCCompileCommand**

The RCCompileCommand property is a string that specifies the compilation command for the resource file. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = \$(RC) /Fo"\$(TARGET\_MAIN).res" \$(TARGET\_MAIN)\$OMRCEExtension*

## **RCEExtension**

The RCEExtension property is a string that specifies the extension for resource files. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = .rc*

## **RemoteHost**

The RemoteHost property specifies the name of the host machine when you run an application remotely.

In such configurations, the host is the machine running Rational Rhapsody, whereas the target is the machine running the application.

To run remotely, the UseRemoteHost property must be set to Checked. If UseRemoteHost is Checked and RemoteHost is blank, the current host name is used for the remote host. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = Empty string*

## **ReusableStatechartSwitches**

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts. See the upgrade history on the support site for information on this change.

*Default = /D "OM\_REUSABLE\_STATECHART\_IMPLEMENTATION"*

## RPFrameWorkDll

The RPFrameWorkDll property determines whether the configuration uses the DLL flavor of the framework libraries. To use OXF DLLs for the creation of COM ATL components, you must set this property to Checked before you generate code. Rational Rhapsody COM ATL components use a DLL version of the OXF. This version of the OXF allows the use of multiple Rational Rhapsody-generated DLL/executable components confined to a single process. There are three versions of the OXF DLL:

DLL Version	Animation Enabled	Trace Enabled	oxfdll.dll	No	No	oxanimdll.dll	Yes	No	oxtracedll.dll
No	Yes								

OXF DLLs are not a part of a typical Rational Rhapsody installation, and must be built from the OXF C++ sources. To obtain these sources, you can either perform a custom installation, or update the installation to add the sources to your existing installation. To rebuild the framework DLLs, run the following in your \$OMROOT\LangCpp folder: `nmake -f msoxfanimtracedll.mak CFG=oxfdll` `nmake -f msoxfanimtracedll.mak CFG=oxanimdll` `nmake -f msoxfanimtracedll.mak CFG=oxtracedll`

To use OXF DLLS for the creation of COM ATL components, set the following component configuration properties to True before you generate code:

- `CPP_CG::Microsoft::RPFrameWorkDll`
- `CPP_CG::MicrosoftDLL::RPFrameWorkDll` this is True by default)

In addition, make sure that the following are included in the system environment path:

- OXF DLL path (\$OMROOT\LangCpp\lib)
- The full path to `regsrv32.exe`

Without these settings, COM ATL components are not registered and cannot run. Limitations:

- Rational Rhapsody components with different instrumentation settings (both Animation and Tracing) are not supported within a single process. However, you can mix instrumented and noninstrumented DLLs, as long as you use the instrumented DLL.
- Mixing Rational Rhapsody components that link to the DLL and the library version of OXF is not supported within a single process.

*Default = Cleared*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*Default = .h*

## SubSystem

The SubSystem property is a string that defines the type of the program for the Microsoft linker. The possible values are as follows:

- **CONSOLE** - Used for a Win32 character-mode application
- **WINDOWS** - Used for an application that does not require a console
- **NATIVE** - Applies device drivers for Windows NT
- **POSIX** - Creates an application that runs with the POSIX subsystem in Windows NT

*Default = /SUBSYSTEM:console*

## UpdateBuildSettingsInIDE

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## UseNonZeroStdInputHandle

The UseNonZeroStdInputHandle property is a Boolean value that specifies whether to use a non-zero standard input handle.

*Default = Checked*

## UseRemoteHost

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Cleared*

## UseTemplateName

In the declarations for members of a template class, some compilers require that you use precede the member type with the keyword "typename" for types with dependent names. Since support for this keyword varies between compilers, the UseTemplateName property is used to specify whether the "typename" keyword should be included in the generated code.

*Default = Cleared*

## ModelElement

The metaclass ModelElement contains properties that can be used to customize code generation by changing the way that Rational Rhapsody handles specific elements when it transforms a model into a

simplified model before generating code.

In general, the properties in this metaclass relate to model elements that can be found under other types of model elements, for example, descriptions and annotations. These properties are therefore visible at different project levels - for example, package, class, and attribute.

## CallOperationGenerationPattern

The CallOperationGenerationPattern property stores the value entered in the "Code pattern" field on the General tab of the Features window for Call Operation elements.

*Default = \$target\$goArr\$operation*

## ForLoopInitialization

If you have a "for" loop in a flowchart, the ForLoopInitialization property is used to store the loop initialization code that you entered in the "Loop initialization" field on the General tab of the Features window for the relevant action or decision node.

For detailed instructions on using "for" loops in flowcharts, see the KC topic called ""While" loops and "for" loops in flowcharts".

*Default = Blank*

## ForLoopStep

If you have a "for" loop in a flowchart, the ForLoopStep property is used to store the loop increment code that you entered in the "Loop step" field on the General tab of the Features window for the relevant action or decision node.

For detailed instructions on using "for" loops in flowcharts, see the KC topic called ""While" loops and "for" loops in flowcharts".

*Default = Blank*

## RefactorRenameRegularExpression

When you use the Refactor:Rename feature, Rhapsody searches the user-defined code in the model for all occurrences of the name of the specified element, including places where the element name appears as part of the name of elements whose name is derived from the element that is being renamed.

To locate these elements, Rhapsody uses a set of regular expressions.

The regular expressions used for this search are defined by the property RefactorRenameRegularExpression.

Different values are used for this property, depending on what type of element you are renaming. For



example, you will see that this property has a different value under Class than it does under Attribute. For some types of elements, Rhapsody uses a more generic expression, taken from the value of the property under ModelElement.

In general, the value of this property is a regular expression that consists of a number of "Or" components. For the renaming to work properly, each such component must contain three groupings, with the middle grouping using the variable \$keyword, which represents the name of the element that is being renamed. In some cases, the expression takes this information from the value of another property rather than using the \$keyword variable.

*Default = ([^A-Za-z0-9\_]/^)(*\$keyword*)([^A-Za-z0-9\_]/\$)*

## **SimplifyAnnotations**

If you are using the Rational Rhapsody customizable code generation mechanism, the SimplifyAnnotations property can be used to change the way annotations are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - Annotations are ignored.
- Copy - Annotations are copied from the original to the simplified model. They are not modified in any way.
- Default - Uses the standard simplification for Annotations, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for Annotations has been applied.
- CodeUpdateAnnotations - Used in the CodeCentric settings in order to minimize the Rational Rhapsody annotations generated in code, limiting them to special cases such as animation. Note that if you try to use this value when not using the CodeCentric settings, roundtripping may not work correctly.

*Default = "Default"*

## **SimplifyDescription**

If you are using the Rational Rhapsody customizable code generation mechanism, the SimplifyDescription property can be used to change the way Descriptions are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - Descriptions are ignored.
- Copy - Descriptionare copied from the original to the simplified model. They are not modified in any way.
- Default - Uses the standard simplification for Descriptions, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user. (For details, see the section on



User-Provided Simplification in the Rational Rhapsody help.)

- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for Descriptions has been applied.

*Default = "Default"*

## SimplifyExternal

If you are using the Rational Rhapsody customizable code generation mechanism, the SimplifyExternal property can be used to change the way that code is generated for external elements that are not actually part of the model, for example, base classes for classes in the model, by changing the way that such elements are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - The elements are ignored.
- Copy - The element are copied from the original to the simplified model. They are not modified in any way.
- Default - Uses the standard simplification for these elements, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for these element has been applied.

*Default = "Default"*

## SimplifyInstrumentation

If you are using the Rational Rhapsody customizable code generation mechanism, the SimplifyInstrumentation property can be used to customize the generation of instrumentation code (such as animation) by changing the way instrumentation is handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - Instrumentation is ignored.
- Copy - Instrumentation is to be copied from the original to the simplified model. It is not modified in any way.
- Default - Uses the standard simplification for instrumentation, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for instrumentation has been applied.

*Default = "Default"*

## SimplifyProperties

If you are using the Rational Rhapsody customizable code generation mechanism, the `SimplifyProperties` property can be used to customize the way that overridden properties affect code generation by changing the way that Rational Rhapsody handles these properties when it transforms the model into a simplified model.

The property can take any of the following values:

- `None` - Overridden properties are ignored.
- `Copy` - Overridden properties are copied from the original to the simplified model. Their effect are not modified in any way.
- `Default` - Uses the standard simplification for overridden properties, as defined in Rational Rhapsody.
- `ByUser` - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- `ByUserPostDefault` - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for overridden properties has been applied.

*Default = "Default"*

## **SimplifyStandardOperations**

If you are using the Rational Rhapsody customizable code generation mechanism, the `SimplifyStandardOperations` property can be used to customize the way that code is generated for operations defined by using the "StandardOperation" properties by changing the way that Rational Rhapsody handles such operations when it transforms the model into a simplified model.

The property can take any of the following values:

- `None` - Standard Operations are ignored.
- `Copy` - Standard Operationare copied from the original to the simplified model. They are not modified in any way.
- `Default` - Uses the standard simplification for Standard Operations, as defined in Rational Rhapsody.
- `ByUser` - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- `ByUserPostDefault` - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for Standard Operations has been applied.

*Default = "Default"*

## **SimplifyWebify**

If you are using the Rational Rhapsody customizable code generation mechanism, the `SimplifyWebify` property can be used to customize the generation of Webify code by changing the way Webify is handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- `None` - Webify is ignored.
- `Copy` - Webify is to be copied from the original to the simplified model. It is not modified in any way.

- **Default** - Uses the standard simplification for Webify, as defined in Rational Rhapsody.
- **ByUser** - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- **ByUserPostDefault** - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for Webify has been applied.

*Default = "Default"*

## TagsToConsider

By default, tags in your model are not copied to the simplified model for the purpose of code generation.

This can be problematic in cases where you have used the conditional property feature to include the value of specific tags in code-generation properties.

In such cases, you can use the property TagsToConsider to specify the names of tags that should be copied to the simplified model.

The value of the property should be a comma-separated list of tag names.

*Default = Blank*

## MSSStandardLibrary

The MSSStandardLibrary metaclass contains environment properties (Compiler, framework libraries, and so on) for Microsoft Win32 compiler and uses the standard iostreams (relevant only for VC++ 6.0 users).

## AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path. Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files. To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)/LangCpp/osconfig/WIN32*

## AdditionalReservedWords

The AdditionalReservedWords property is a string that specifies additional reserved keywords that

Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody. The property value is checked at run time when you name or rename an element, based on the active configuration environment setting. Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default =*

`__asm __finally naked __based __inline __single_inheritance __cdecl __int8 __stdcall __declspec __int16  
thread dllexport __int32 __try dllimport __int64 __virtual_inheritance __except __leave __fastcall  
__multiple_inheritance`

## BuildArgumentsInIDE

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the  
CPP\_CG::[environment]::BuildCommandInIDE property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandInIDE

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration.

To change this property, use the Configuration window in the browser - do not change it by using the Properties tab in the Features window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link. The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag

(:cx\_option=exceptions).

- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*Default = Debug*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\$OMROOT/etc/Executer.exe" "\" \"\$OMROOT\" \"etc\msmake.bat msbuild.mak build \*  
*\"USE\_STL=TRUE\" \"USE\_PDB=FALSE\" \"\"*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## CompileSwitches

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property.

*Default =*

*/I . /I \$OMDefaultSpecificationDirectory /I \$(OMROOT)\LangCpp /I \$(OMROOT)\LangCpp\oxf /nologo /W3 /GX \$OMCPPCompileCommandSet /D "\_AFXDLL" /D "WIN32" /D "\_CONSOLE" /D "\_MBCS" /D "\_WINDOWS" /D "OM\_USE\_STL" \$(INST\_FLAGS) \$(INCLUDE\_PATH) \$(INST\_INCLUDES) /c*

## CPPAdditionalReservedWords

The CPPAdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use.

In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody. The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

*Default =*

`__asm __finally naked __based __inline __single_inheritance __cdecl __int8 __stdcall __declspec __int16  
dllexport __int32 __try __declspec __declspec __int64 __virtual_inheritance __except __leave __fastcall  
__multiple_inheritance`

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default =*

`$(CREATE_OBJ_DIR) $(CPP) $OMFileCPPCompileSwitches /Fo"$OMFileObjPath"  
"$OMFileImpPath"`

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*Default =* `/Zi /Od /D "_DEBUG" /MDd /Fd"$(TARGET_NAME)"`

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*Default =* `/Ox /D"NDEBUG" /MD /Fd"$(TARGET_NAME)"`

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

*Default =* `$OMFileObjPath : $OMFileImpPath $OMFileSpecPath $OMFileDependencies`

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: `$OMFileObjPath : $OMFileImpPath "*.bmp" "*.ico" "*.rc2"`

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction with an IDE such as Eclipse, the EnableDebugIntegrationWithIDE property can be used to specify whether or not the IDE debugger should

be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to True, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's exit() function is called automatically. The system function calls the destructors of framework instances in the order it determines. The EndApplicationCode property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an endApplication() function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to endApplication(), it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The EntryPoint property specifies the name of the main program for a given environment.

*Default = main*

See also the definition of the EntryPointDeclarationModifier property for more information.

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens has three parameters, each with an integer value:

- TotalNumberOfTokens - The number of tokens in the regular expression
- FileTokenPosition - The position of the file name token in the expression
- LineTokenPosition - The position of the line number token in the expression

*Default = ToTalNumberOfTokens=3,FileTokenPosition=1,LineTokenPosition=2*

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the `CPP_CG::<Environment>::ExeName` property plus the value of this property.

*(Default = .exe)*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the `ExeName` property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the `CPP_CG::<Environment>::ExeExtension` property.

*(Default = Blank)*

## FileDependencies

The `FileDependencies` property specifies which framework specification files and implementation files should be included in model elements. The file inclusions are generated in the makefile.

*Default = \$OMSpecIncludeInElements \$OMImpIncludeInElements*

## GetConnectedRuntimeLibraries

The `GetConnectedRuntimeLibraries` property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile. Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

*Default =*

`$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)WebComponents$(LIB_POSTFIX)$(LIB_EXT),  
$(OMROOT)\lib\MSWebServices$(LIB_POSTFIX)$(LIB_EXT), ws2_32$(LIB_EXT)`

## ImpExtension

The `ImpExtension` property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment.

*Default = .cpp*



## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*Default = !INCLUDE*

## InvokeAnimatedExecutable

The AnimationPortRange value in the rhapsody.ini file makes it possible to use animation with multiple instances of Rhapsody running simultaneously.

Ordinarily, when you run an application with animation, the executable is launched using the command specified by the InvokeExecutable property. However, if you have specified a value greater than zero for AnimationPortRange in your rhapsody.ini file, the executable is launched using the command specified by the InvokeAnimatedExecutable property. By default, the value of InvokeAnimatedExecutable includes a reference to the port to use for animation.

*Default = \$executable -port \$port*

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*Default = "\$executable"*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\msmake.bat\" \$makefile \$maketarget"*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format.

If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .lib*

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

*Default = Empty string (blank)*

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

Default value = Empty string

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

*Default = \$OMLinkCommandSet /NOLOGO*

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the CPP\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*(Default = .mak)*

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration.

```
Default = ##### Target type (Debug/Release) #####
#####
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease BuildSet=$OMBuildSet
ConfigurationCPPCompileSwitches=$OMReusableStatechartSwitches
$OMConfigurationCPPCompileSwitches ##### Compilation flags
#####
LIB_PREFIX=MSStl INCLUDE_QUALIFIER=/I ##### Commands definition
#####
##### RMDIR = rmdir
LIB_CMD=link.exe -lib LINK_CMD=link.exe LIB_FLAGS=$OMConfigurationLinkSwitches
LINK_FLAGS=$OMConfigurationLinkSwitches /SUBSYSTEM:console /MACHINE:I386
/nodefaultlib:"libc.lib" ##### Generated macros #####
##### $OMContextMacros
OBJ_DIR=$OMObjectsDir !IF "$(OBJ_DIR)"!=" " CREATE_OBJ_DIR=if not exist $(OBJ_DIR) mkdir
$(OBJ_DIR) CLEAN_OBJ_DIR= if exist $(OBJ_DIR) $(RMDIR) $(OBJ_DIR) !ELSE
CREATE_OBJ_DIR= CLEAN_OBJ_DIR= !ENDIF ##### Predefined macros
#####
##### $(OBS) :
$(INST_LIBS) $(OXF_LIBS) LIB_POSTFIX= !IF "$(BuildSet)"=="Release" LIB_POSTFIX=R !ENDIF
!IF "$(TARGET_TYPE)"=="Executable" LinkDebug=$(LinkDebug) /DEBUG
LinkRelease=$(LinkRelease) /OPT:NOREF !ELSEIF "$(TARGET_TYPE)"=="Library"
LinkDebug=$(LinkDebug) /DEBUGTYPE:CV !ENDIF !IF "$(INSTRUMENTATION)"=="Animation"
INST_FLAGS=/D "OMANIMATOR" INST_INCLUDES=/I $(OMROOT)\LangCpp\com /I
$(OMROOT)\LangCpp\com INST_LIBS=
$(OMROOT)\LangCpp\lib$(LIB_PREFIX)aomanim$(LIB_POSTFIX)$(LIB_EXT)
OXF_LIBS=$(OMROOT)\LangCpp\lib$(LIB_PREFIX)oxfinst$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib$(LIB_PREFIX)omComAppl$(LIB_POSTFIX)$(LIB_EXT)
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)"=="Tracing" INST_FLAGS=/D
"OMTRACER" INST_INCLUDES=/I $(OMROOT)\LangCpp\com /I $(OMROOT)\LangCpp\com
INST_LIBS=$(OMROOT)\LangCpp\lib$(LIB_PREFIX)tomtrace$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib$(LIB_PREFIX)aomtrace$(LIB_POSTFIX)$(LIB_EXT) OXF_LIBS=
```

```

$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfirst$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX)$(LIB_EXT)
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)" == "None" INST_FLAGS=
INST_INCLUDES= INST_LIBS=
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$(LIB_POSTFIX)$(LIB_EXT) SOCK_LIB=
!ELSE !ERROR An invalid Instrumentation $(INSTRUMENTATION) is specified. !ENDIF
##### Generated dependencies #####
#####
$OMContextDependencies $OMFileObjPath : $OMMainImplementationFile $(OBJS) $(CPP)
$(ConfigurationCPPCompileSwitches) /Fo"$OMFileObjPath" $OMMainImplementationFile
##### Linking instructions #####
#####
$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS) $OMFileObjPath
$OMMakefileName $OMModelLibs @echo Linking $(TARGET_NAME)$(EXE_EXT) $(LINK_CMD)
$OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \ $(OXF_LIBS) \
$(SOCK_LIB) \ $(LINK_FLAGS) /out:$(TARGET_NAME)$(EXE_EXT) $(TARGET_NAME)$(LIB_EXT) :
$(OBJS) $(ADDITIONAL_OBJS) $OMMakefileName @echo Building library $@ $(LIB_CMD)
$(LIB_FLAGS) /out:$(TARGET_NAME)$(LIB_EXT) $(OBJS) $(ADDITIONAL_OBJS) clean: @echo
Cleanup $OMCleanOBJS if exist $OMFileObjPath erase $OMFileObjPath if exist *$(OBJ_EXT) erase
*$(OBJ_EXT) if exist $(TARGET_NAME).pdb erase $(TARGET_NAME).pdb if exist
$(TARGET_NAME)$(LIB_EXT) erase $(TARGET_NAME)$(LIB_EXT) if exist $(TARGET_NAME).ilk
erase $(TARGET_NAME).ilk if exist $(TARGET_NAME)$(EXE_EXT) erase
$(TARGET_NAME)$(EXE_EXT) $(CLEAN_OBJ_DIR)

```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the CPP\_CG::<Environment>::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## ObjCleanCommand

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default = if exist \$OMFileObjPath erase \$OMFileObjPath*

## ObjectName

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = Empty string*

## ObjectsDirectory

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = Empty string*

## ObjExtension

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*Default = .obj*

## OSFileSystemCaseSensitive

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Cleared*

## ParseErrorDescript

The ParseErrorDescript property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = ([^()]+)(([0-9]+)) [:] (error/warning/fatal error) (.\*)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = ([^()]+)(([0-9]+)) [:] (error/warning/fatal error)*

## ParseMakeError

The ParseMakeError property is used to define a regular expression that represents the format of make process or linker error messages. This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = (NMAKE)(.\*)(fatal error)*

## ParseSeverityError

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^()]+)([()([0-9]+)] [:] (error|fatal error)*

## ParseSeverityWarning

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^()]+)([()([0-9]+)] [:] (warning)*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation.

*Default = \*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Checked*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely.

In such configurations, the host is the machine running Rational Rhapsody, whereas the target is the machine running the application.

To run remotely, the UseRemoteHost property must be set to Checked. If UseRemoteHost is Checked and

RemoteHost is blank, the current host name is used for the remote host. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = Empty string*

## ReusableStatechartSwitches

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts. See the upgrade history on the support site for information on this change.

*Default = /D "OM\_REUSABLE\_STATECHART\_IMPLEMENTATION"*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*Default = .h*

## UpdateBuildSettingsInIDE

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## UseNewBuildOutputWindow

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the CG::General::ShowLogViewAfterBuild property to True.

*Default = Checked*

## UseNonZeroStdInputHandle

The UseNonZeroStdInputHandle property is a Boolean value that specifies whether to use a non-zero standard input handle.

*Default = Checked*

## UseRemoteHost

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Cleared*

## UseTemplateName

In the declarations for members of a template class, some compilers require that you use precede the member type with the keyword "typename" for types with dependent names. Since support for this keyword varies between compilers, the UseTemplateName property is used to specify whether the "typename" keyword should be included in the generated code.

*Default = Cleared*

# MSVC

The MSVC metaclass contains the Environment settings (compiler, framework libraries, and so on) for the MSVC environment.

## AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)/LangCpp/osconfig/WIN32*

## AdditionalReservedWords

The AdditionalReservedWords property is a string that specifies additional reserved keywords that



Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = \_\_asm \_\_finally naked \_\_based \_\_inline \_\_single\_inheritance \_\_cdecl \_\_int8 \_\_stdcall  
\_\_declspec \_\_int16 dllexport \_\_int32 \_\_try dllimport \_\_int64 \_\_virtual\_inheritance \_\_except \_\_leave  
\_\_fastcall \_\_multiple\_inheritance*

## **AnimIncludeDirectories**

The AnimIncludeDirectories property is used to specify the directories that must be referenced in the makefile for projects where Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with INST\_INCLUDES.

*Default = \$(INCLUDE\_QUALIFIER)\$(OMROOT)\LangCpp\acom  
\$(INCLUDE\_QUALIFIER)\$(OMROOT)\LangCpp\com*

## **AnimInstLibs**

When using one of Rational Rhapsody's IDE integrations, the AnimInstLibs property is used to specify the instrumentation framework libraries that are required when Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with INST\_LIBS.

*Default = \$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)aomanim\$(LIB\_POSTFIX)\$(LIB\_EXT)*

## **AnimOxfLibs**

When using one of Rational Rhapsody's IDE integrations, the AnimOxfLibs property is used to specify the framework libraries that are required when Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with OXF\_LIBS.

*Default = \$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)oxfinst\$(LIB\_POSTFIX)\$(LIB\_EXT)  
\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)omComAppl\$(LIB\_POSTFIX)\$(LIB\_EXT) winmm.lib*

## **AnimPreprocessor**

The AnimPreprocessor property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with INST\_FLAGS.

*Default = \$(DEFINE\_QUALIFIER)OMANIMATOR*

## AutoAttachToIDEDebugger

This property is not relevant for the MSVC9 environment.

## BuildArgumentsInIDE

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the [lang]\_CG::[environment]::BuildCommandInIDE property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandInIDE

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration. To change this property, use the Configuration window in the browser - do not change it by using the Properties window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link.

*Default = Debug*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

```
Default = "$OMROOT/etc/Executer.exe" "\"$OMROOT\"\\etc\\msvcmake.bat msbuild.mak build $CPU  
$IDEVersion \"LIB_PREFIX=$(LibPrefix)\" \"USE_STL=FALSE\" \"USE_PDB=FALSE\"  
\"BUILD_SET=$BuildCommandSet\" \"USE_LIBCMT=$UseLIBCMT\" \"\"
```

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = False*

## COM

The COM property is a boolean property that controls the value of the SUBSYSTEM linker option in the generated makefile.

*Default = False*

## CompilerFlags

The CompilerFlags property is used to specify additional compilation options. The value of this property is included in the value of the CPPCompileSwitches property.

When using the integration with Visual Studio, the options specified with this property are included in the settings for the Visual Studio project.

*Default = /EHa*

## CompileSwitches

The CompileSwitches property specifies the compiler switches.

```
Default = /I . /I $OMDefaultSpecificationDirectory /I $(OMROOT)\LangCpp /I  
$(OMROOT)\LangCpp\oxf /nologo /W3 $(ENABLE_EH) $(CRT_FLAGS) $OMCPPCompileCommandSet  
/D "_AFXDLL" /D "WIN32" /D "_CONSOLE" /D "_MBCS" /D "_WINDOWS" $(INST_FLAGS)  
$(INCLUDE_PATH) $(INST_INCLUDES) /c
```

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = \$(CREATE\_OBJ\_DIR) \$(CPP) \$OMFileCPPCompileSwitches /Fo"\$OMFileObjPath"  
"\$OMFileImpPath"*

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*Default = /Zi /Od /D "\_DEBUG" \$(LIBCRT\_FLAG) /Fd"\$\$(TARGET\_NAME)"*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*Default = /Ox /D "NDEBUG" \$(LIBCRT\_FLAG) /Fd"\$\$(TARGET\_NAME)"*

## CPU

The CPU property is used to specify the CPU of the target environment.

When building applications or framework libraries for MSVC9 environments, Rhapsody uses a variable called \$CPU which represents the value provided for the CPU property for these environments. This variable is provided as a parameter to the msvc9make.bat batch file, and is also used in the makefiles that are generated.

When building 32-bit applications, use the value x86 for this property.

When building 64-bit applications, use the value x64 for this property.

*Default = x86*

## DefinedSymbols

The DefinedSymbols property is used to specify preprocessor directives. The value of this property is included in the value of the CPPCompileSwitches property.

When using the integration with Visual Studio, the preprocessor directives specified with this property are included in the preprocessor-related settings for the Visual Studio project.

*Default = \$(DEFINE\_QUALIFIER)\_CRT\_SECURE\_NO\_DEPRECATED*

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: `$OMFileObjPath : $OMFileImpPath "*.bmp" "*.ico" "*.rc2"`

*Default = \$OMFileObjPath : \$OMFileImpPath \$OMFileSpecPath \$OMFileDependencies*

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction an IDE such as Eclipse, you can use the `EnableDebugIntegrationWithIDE` property to specify whether or not the IDE debugger is to be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to `Checked`, the IDE debugger is used.

*Default = False*

## EndApplicationCode

When the main function ends, the system's `exit()` function is called automatically. The system function calls the destructors of framework instances in the order it determines. The `EndApplicationCode` property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an `endApplication()` function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to `endApplication()`, it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The `EntryPoint` property specifies the name of the main program for a given environment.

*Default = main*

## ErrorMessageTokensFormat

The `ErrorMessageTokensFormat`, working with the `ParseErrorMessage` property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. The `ErrorMessageTokensFormat` property defines the number and location of tokens within the regular expression defined by the `ParseErrorMessage` property. The `ErrorMessageTokens` property has three parameters, each with an integer value:

- `TotalNumberOfTokens` - The number of tokens in the regular expression
- `FileTokenPosition` - The position of the file name token in the expression
- `LineTokenPosition` - The position of the line number token in the expression

*Default = ToTalNumberOfTokens=3,FileTokenPosition=1,LineTokenPosition=2*

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the C\_CG::<Environment>::ExeName property plus the value of the ExeExtension property.

*Default = .exe*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the [lang]\_CG::[environment]::ExeExtension property.

*Default = Blank*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements. The file inclusions are generated in the makefile.

*Default = \$OMSpecIncludeInElements \$OMImpIncludeInElements*

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile.

Note: If you select Release Build Set, these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

*Default = \$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)WebComponents\$(LIB\_POSTFIX)\$(LIB\_EXT),  
\$(OMROOT)\lib\\$(LIB\_PREFIX)WebServices\$(LIB\_POSTFIX)\$(LIB\_EXT), ws2\_32\$(LIB\_EXT)*

## IDEVersion

When you select one of the provided Microsoft environments for a configuration in your model, you use the IDEVersion property to tell Rational Rhapsody which version of MS Visual Studio you are using, 2008 or 2010.

The value of this property is passed as a parameter to the Build Framework command, and to the command used to build your application. The value of the property is also used in properties that affect the content of the generated makefile.

The possible values are:

- VC9 - to specify that you are using Visual Studio 2008
- VC10 - to specify that you are using Visual Studio 2010
- VC11 - to specify that you are using Visual Studio 2012

*Default = VC10*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment.

*Default = .cpp*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*Default = !INCLUDE*

## IncludeDirectories

For environments such as Cygwin, the IncludeDirectories property is used to specify additional directories to search for include files. The value of this property is included in the value of the CPPCompileSwitches property and is then included in the generated makefile.

When using the integration with Visual Studio, the directories specified with this property are added to the "additional include directories" specified for the Visual Studio project.

*Default = \$(INCLUDE\_QUALIFIER). \$(INCLUDE\_QUALIFIER)\$(OMROOT)  
\$(INCLUDE\_QUALIFIER)\$(OMROOT)\LangCpp \$(INCLUDE\_QUALIFIER)\$(OMROOT)\LangCpp\oxf*

## InvokeAnimatedExecutable

The AnimationPortRange value in the rhapsody.ini file makes it possible to use animation with multiple instances of Rhapsody running simultaneously.

Ordinarily, when you run an application with animation, the executable is launched using the command specified by the InvokeExecutable property. However, if you have specified a value greater than zero for AnimationPortRange in your rhapsody.ini file, the executable is launched using the command specified by the InvokeAnimatedExecutable property. By default, the value of InvokeAnimatedExecutable includes a reference to the port to use for animation.

*Default = \$executable -port \$port*

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*Default = "\$executable"*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, include the name of the property preceded by \$. As shown in the following example (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\msvc9make.bat \$makefile \$maketarget \$CPU\" "*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = False*

## LibExtension



The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .lib*

## LibPrefix

The LibPrefix property is used to specify a prefix to add to generic names of runtime libraries. The value of this property is used in references to libraries in the generated makefile.

*Default = MS\$(IDEVersion)\$(CPU)\$(MT\_PREFIX)*

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

*Default = Blank*

## LinkerFlags

The LinkerFlags property allows you to define linker flags. The value of the property is inserted into the value of the LinkSwitches property. In the generated makefile, you can see the value of this property in the line that begins with LINK\_FLAGS=.

*Default = Blank*

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

*Default = Blank*

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

*Default = \$OMLinkCommandSet \$LinkerFlags /NOLOGO*

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the [lang]\_CG::[environment]::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*Default = .mak*

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration. For example, the default content of the target type section of a C++ makefile for the Microsoft environment is as follows: ##### Target type (Debug/Release)

```
#####  
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease  
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease BuildSet=$OMBuildSet  
SUBSYSTEM=$OMSubSystem COM=$OMCOM RPFrameWorkDll=$OMRPFrameWorkDll  
ConfigurationCPPCompileSwitches= $OMReusableStatechartSwitches  
$OMConfigurationCPPCompileSwitches !IF "$(RPFrameWorkDll)" == "True"  
ConfigurationCPPCompileSwitches= $(ConfigurationCPPCompileSwitches) /D "FRAMEWORK_DLL"  
!ENDIF !IF "$(COM)" == "True" SUBSYSTEM=/SUBSYSTEM:windows !ENDIF
```

**Compilation Flags** The compilation flags section of the makefile contains the default compilation flags stored in the CompileSwitches property. For example, the default content of the compilation flags section of a C++ makefile for the Microsoft environment is as follows: ##### Compilation flags

```
#####  
INCLUDE_QUALIFIER=/I LIB_PREFIX=MS
```

**Commands Definitions** The commands definition section of the makefile specifies programs to execute from the makefile. For example, the default commands definition section of a C++ makefile for the Microsoft environment is as follows: ##### Commands definition #####

```
##### RMDIR = rmdir LIB_CMD=link.exe -lib  
LINK_CMD=link.exe LIB_FLAGS=$OMConfigurationLinkSwitches  
LINK_FLAGS=$OMConfigurationLinkSwitches $(SUBSYSTEM) / MACHINE:I386
```

**Generated Macros** The generated macros section of the makefile contains a variable that expands to the Rational Rhapsody-generated macros in the makefile. For example: ##### Generated macros

```
##### $OMContextMacros  
OBJ_DIR=$OMObjectsDir !IF "$(OBJ_DIR)" != "" CREATE_OBJ_DIR=if not exist $(OBJ_DIR) mkdir  
$(OBJ_DIR) CLEAN_OBJ_DIR= if exist $(OBJ_DIR) $(RMDIR) $(OBJ_DIR) !ELSE
```

```
CREATE_OBJ_DIR= CLEAN_OBJ_DIR= !ENDIF
```

The `$OMContextMacros` keyword expands several macros in the makefile. Each makefile macro has its own keyword. You can use these keywords separately to customize the makefile. The `$OMContextMacros` variable modifies target-specific variables. Replace the `$OMContextMacros` line in the `MakeFileContent` property with the following: `FLAGSFILE=$OMFlagsFile`  
`RULESFILE=$OMRulesFile OMROOT=$OMROOT C_EXT=$OMImplExt H_EXT=$OMSpecExt`  
`OBJ_EXT=$OMObjExt EXE_EXT=$OMExeExt LIB_EXT=$OMLibExt`  
`INSTRUMENTATION=$OMInstrumentation TIME_MODEL=$OMTimeModel`  
`TARGET_TYPE=$OMTargetType TARGET_NAME=$OMTargetName $OMAllDependencyRule`  
`TARGET_MAIN=$OMTargetMain LIBS=$OMLibs INCLUDE_PATH=$OMIncludePath`  
`ADDITIONAL_OBJS=$OMAdditionalObjs OBJS= $OMObjs`

**Predefined Macros** The predefined macros section of the makefile contains other macros than the Rational Rhapsody-generated macros specified in the generated macros section. For example, the default predefined macros section of a C++ makefile for the Microsoft environment is as follows:

```
##### Predefined macros #####
##### $(OBJS) : $(INST_LIBS)
$(OXF_LIBS) LIB_POSTFIX= !IF "$$(BuildSet)"=="Release" LIB_POSTFIX=R !ENDIF !IF
"$(TARGET_TYPE)"=="Executable" LinkDebug=$(LinkDebug) /DEBUG
LinkRelease=$(LinkRelease) /OPT:NOREF !ELSEIF "$(TARGET_TYPE)"=="Library"
LinkDebug=$(LinkDebug) /DEBUGTYPE:CV !ENDIF !IF "$(INSTRUMENTATION)"=="Animation"
INST_FLAGS=/D "OMANIMATOR" INST_INCLUDES=/I $(OMROOT)\LangCpp\om /I
$(OMROOT)\LangCpp\tom !IF "$(RPFrameWorkDll)"=="True" INST_LIBS=
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxanimdll$(LIB_POSTFIX)$(LIB_EXT)
!ELSE INST_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomanim$(LIB_POSTFIX)$(LIB_EXT)
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfinst$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX)$(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)"=="Tracing" INST_FLAGS=/D
"OMTRACER" INST_INCLUDES=/I $(OMROOT)\LangCpp\om /I $(OMROOT)\LangCpp\tom !IF
"$(RPFrameWorkDll)"=="True" INST_LIBS= OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfracedll$(LIB_POSTFIX)$(LIB_EXT) !ELSE
INST_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)tomtrace$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomtrace$(LIB_POSTFIX)$(LIB_EXT) OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfinst$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX)$(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)"=="None" INST_FLAGS=
INST_INCLUDES= INST_LIBS= !IF "$(RPFrameWorkDll)"=="True"
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfdll$(LIB_POSTFIX)$(LIB_EXT) !ELSE
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$(LIB_POSTFIX)$(LIB_EXT) !ENDIF
SOCK_LIB= !ELSE !ERROR An invalid Instrumentation $(INSTRUMENTATION) is specified.
!ENDIF
```

**Generated Dependencies** The generated dependencies section of the makefile contains a variable that expands to Rational Rhapsody-generated dependencies and compilation instructions. For example, the generated dependencies section of a C++ makefile for the Microsoft environment is as follows:

```
##### Generated dependencies #####
##### $OMContextDependencies
$OMFileObjPath : $OMMainImplementationFile $(OBJS) $(CPP) $(ConfigurationCPPCompileSwitches)
/Fo"$OMFileObjPath" $OMMainImplementationFile
```

**Linking Instructions** The linking instructions section of the makefile contains the predefined linking instructions. For example, the default linking instructions section of a C++ makefile for the Microsoft environment is as follows: ##### Linking instructions #####

```
#####
$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS) $OMFileObjPath
$OMMakefileName $OMModelLibs @echo Linking $(TARGET_NAME)$(EXE_EXT) $(LINK_CMD)
$OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \ $(OXF_LIBS) \
$(SOCK_LIB) \ $(LINK_FLAGS) /out:$(TARGET_NAME)$(EXE_EXT)
$(TARGET_NAME)$(LIB_EXT) : $(OBJS) $(ADDITIONAL_OBJS) $OMMakefileName @echo
Building library $@ $(LIB_CMD) $(LIB_FLAGS) /out:$(TARGET_NAME)$(LIB_EXT) $(OBJS)
$(ADDITIONAL_OBJS) clean: @echo Cleanup $OMCleanOBJS if exist $OMFileObjPath erase
$OMFileObjPath if exist *$(OBJ_EXT) erase *$(OBJ_EXT) if exist $(TARGET_NAME).pdb erase
$(TARGET_NAME).pdb if exist $(TARGET_NAME)$(LIB_EXT) erase
$(TARGET_NAME)$(LIB_EXT) if exist $(TARGET_NAME).ilk erase $(TARGET_NAME).ilk if exist
$(TARGET_NAME)$(EXE_EXT) erase $(TARGET_NAME)$(EXE_EXT) $(CLEAN_OBJ_DIR)
```

```
Default = ##### Target type (Debug/Release) #####
#####
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease BuildSet=$OMBuildSet
SUBSYSTEM=$OMSubSystem RPFrameworkDll=$OMRPFrameworkDll
SimulinkLibName=$SimulinkLibName
ConfigurationCPPCompileSwitches=$OMReusableStatechartSwitches
$OMConfigurationCPPCompileSwitches $OMSDLCompileSwitches $OMDDSFileCPPCompileSwitches
!IF "$(RPFrameworkDll)" == "True"
ConfigurationCPPCompileSwitches=$(ConfigurationCPPCompileSwitches) /D "FRAMEWORK_DLL"
!ENDIF ##### Compilation flags #####
##### INCLUDE_QUALIFIER=/I
DEFINE_QUALIFIER=/D !IF "$(UseLIBCMT)" == "True" MT_PREFIX=MT LIBCRT_FLAG=/MT
!ELSE MT_PREFIX= LIBCRT_FLAG=/MD !ENDIF LIB_PREFIX=$LibPrefix
CRT_FLAGS=$DefinedSymbols ENABLE_EH=/EHa WINMM_LIB=winmm.lib #####
Commands definition #####
##### RMDIR = rmdir
LIB_CMD=link.exe -lib LINK_CMD=link.exe LIB_FLAGS=$OMConfigurationLinkSwitches
LINK_FLAGS=$OMConfigurationLinkSwitches $(SUBSYSTEM) /MACHINE:$CPU #####
Generated macros #####
$OMContextMacros OBJ_DIR=$OMObjectsDir !IF "$(OBJ_DIR)" != "" CREATE_OBJ_DIR=if not exist
$(OBJ_DIR) mkdir $(OBJ_DIR) CLEAN_OBJ_DIR= if exist $(OBJ_DIR) $(RMDIR) $(OBJ_DIR) !ELSE
CREATE_OBJ_DIR= CLEAN_OBJ_DIR= !ENDIF ##### Predefined macros
##### !IF "$(OBJS)"
!= "" $(OBJS) : $(INST_LIBS) $(OXF_LIBS) !ENDIF LIB_POSTFIX= !IF "$(BuildSet)" == "Release"
LIB_POSTFIX=R !ENDIF !IF "$(TARGET_TYPE)" == "Executable" LinkDebug=$(LinkDebug)
/DEBUG LinkRelease=$(LinkRelease) /OPT:NOREF !ELSEIF "$(TARGET_TYPE)" == "Library"
LinkDebug=$(LinkDebug) !ENDIF !IF "$(INSTRUMENTATION)" == "Animation" INST_FLAGS=/D
"OMANIMATOR" INST_INCLUDES=/I $(OMROOT)\LangCpp\aoom /I $(OMROOT)\LangCpp\tom !IF
"$(RPFrameworkDll)" == "True" INST_LIBS=
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfanimdll$(LIB_POSTFIX)$(LIB_EXT) !ELSE
INST_LIBS= $(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomanim$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxsiminst$(LIB_POSTFIX)$(LIB_EXT)
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfinst$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX)$(LIB_EXT)
$(SimulinkLibName) !ENDIF SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)" ==
"Tracing" INST_FLAGS=/D "OMTRACER" INST_INCLUDES=/I $(OMROOT)\LangCpp\aoom /I
$(OMROOT)\LangCpp\tom !IF "$(RPFrameworkDll)" == "True" INST_LIBS= OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfracedll$(LIB_POSTFIX)$(LIB_EXT) !ELSE
INST_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)tomtrace$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomtrace$(LIB_POSTFIX)$(LIB_EXT)
```

```

$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxsiminst$(LIB_POSTFIX)$(LIB_EXT) OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfst$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX)$(LIB_EXT)
$(SimulinkLibName) !ENDIF SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)" == "None"
INST_FLAGS= INST_INCLUDES= INST_LIBS= !IF "$(RPFrameWorkDll)" == "True"
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfdll$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxsim$(LIB_POSTFIX)$(LIB_EXT) !ELSE
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$(LIB_POSTFIX)$(LIB_EXT)
$(SimulinkLibName) $(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxsim$(LIB_POSTFIX)$(LIB_EXT)
!ENDIF SOCK_LIB= !ELSE !ERROR An invalid Instrumentation $(INSTRUMENTATION) is specified.
!ENDIF ##### Generated dependencies #####
#####
$OMContextDependencies $OMFileObjPath : $OMMainImplementationFile $(OBJS) $(CPP)
$(ConfigurationCPPCompileSwitches) /Fo"$OMFileObjPath" $OMMainImplementationFile
##### Linking instructions #####
#####
$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS) $OMFileObjPath
$OMMakefileName $OMModelLibs @echo Linking $(TARGET_NAME)$(EXE_EXT) $(LINK_CMD)
$OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \ $(OXF_LIBS) \
$(SOCK_LIB) \ $(WINMM_LIB) \ $(LINK_FLAGS) /out:$(TARGET_NAME)$(EXE_EXT) if exist
$(TARGET_NAME)$(EXE_EXT).manifest mt.exe -manifest $@.manifest
-outputresource:$(TARGET_NAME)$(EXE_EXT);1 $(TARGET_NAME)$(LIB_EXT) : $(OBJS)
$(ADDITIONAL_OBJS) $OMMakefileName @echo Building library $@ $(LIB_CMD) $(LIB_FLAGS)
/out:$(TARGET_NAME)$(LIB_EXT) $(OBJS) $(ADDITIONAL_OBJS) $(LIBS) clean: @echo Cleanup
$OMCleanOBJS if exist $OMFileObjPath erase $OMFileObjPath if exist *$(OBJ_EXT) erase
*$(OBJ_EXT) if exist $(TARGET_NAME).pdb erase $(TARGET_NAME).pdb if exist
$(TARGET_NAME)$(LIB_EXT) erase $(TARGET_NAME)$(LIB_EXT) if exist $(TARGET_NAME).ilk
erase $(TARGET_NAME).ilk if exist $(TARGET_NAME)$(EXE_EXT) erase
$(TARGET_NAME)$(EXE_EXT) if exist $(TARGET_NAME)$(EXE_EXT).manifest erase
$(TARGET_NAME)$(EXE_EXT).manifest $(CLEAN_OBJ_DIR) $OMCleanDdsIdlFiles

```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the [lang]\_CG::[environment]::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*Default = Blank*

## NoneIncludeDirectories

The NoneIncludeDirectories property is used to specify the directories that must be referenced in the makefile for projects where Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with INST\_INCLUDES.

*Default = Blank*



## NoneInstLibs

When using one of Rational Rhapsody's IDE integrations, the NoneInstLibs property is used to specify the instrumentation framework libraries that are required when Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with INST\_LIBS.

*Default = Blank*

## NoneOxfLibs

When using one of Rational Rhapsody's IDE integrations, the NoneOxfLibs property is used to specify the framework libraries that are required when Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with OXF\_LIBS.

*Default = \$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)oxf\$(LIB\_POSTFIX)\$(LIB\_EXT) winmm.lib*

## NonePreprocessor

The NonePreprocessor property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with INST\_FLAGS.

*Default = Blank*

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## ObjCleanCommand

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default = if exist \$OMFileObjPath erase \$OMFileObjPath*

## ObjectName

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = Blank*

## ObjectsDirectory

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = Blank*

## ObjExtension

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*Default = .obj*

## OSFileSystemCaseSensitive

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = False*

## ParseErrorDescript

The ParseErrorDescript property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = ([^()]+)[([0-9]+)] [:] (error/warning/fatal error) (.\*)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning)

Along with the ErrorMessageTokensFormat property, the ParseErrorMessage property specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = ([^()]+)[([0-9]+)] [:] (error/warning/fatal error)*

## ParseErrorMoreInfo

The ParseErrorMoreInfo property is used to define a regular expression that represents the format of compiler error messages that follow the first line which contains the term 'error' or 'warning'.

This property is used to parse the compiler output and extract the additional information from the message so that this information can be displayed to the user in the build output window.

*Default = ^[( )]*

## **ParseMakeError**

The ParseMakeError property is used to define a regular expression that represents the format of make process or linker error messages. This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = (NMAKE/LINK)(.\*) (fatal error)*

## **ParseSeverityError**

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = [:] (error/fatal error)*

## **ParseSeverityWarning**

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = [:] (warning)*

## **PathDelimiter**

The PathDelimiter property specifies an alternative path separator for code generation.

*Default = \*

## **QuoteOMROOT**

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = True*

## **RemoteHost**

The RemoteHost property specifies the name of the host machine when you run an application remotely.



In such configurations, the host is the machine running the product, whereas the target is the machine running the application. To run remotely, the UseRemoteHost property must be set to Checked.

If the UseRemoteHost property is set to Checked and the RemoteHost property is blank, the current host name is used for the remote host. You can use this as a workaround if you have problems running animated applications on Windows 95.

You can leave the RemoteHost property blank if both the application and Rational Rhapsody are running on the same machine.

*Default = Blank*

## **ReusableStatechartSwitches**

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts.

*Default = /D "OM\_REUSABLE\_STATECHART\_IMPLEMENTATION"*

## **RPFrameWorkDll**

The RPFrameWorkDll property can be used to specify that an application should use the dll generated for the OXF framework.

When the property is set to True, the makefile that is generated for the application includes linkage to the relevant OXF framework dll, based on the instrumentation mode specified for the configuration: oxfanimdll.dll if animation is used, oxfracedll.dll if tracing is used, and oxfdll.dll if the configuration does not use animation or tracing.

*Default = False*

## **SockLib**

The SockLib property represents the name of the socket library.

For the Cygwin environment, the value entered for this property is used in the generated makefile.

When using the integration with Visual Studio, the value of this property is included in the "additional dependencies" for the Visual Studio project.

*Default = wsock32.lib*

## **SpecExtension**

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*Default = .h*

## SubSystem

The SubSystem property is a string that defines the type of the program for the Microsoft linker. The possible values are as follows:

- **CONSOLE** - Used for a Win32 character-mode application
- **WINDOWS** - Used for an application that does not require a console
- **NATIVE** - Applies device drivers for Windows NT
- **POSIX** - Creates an application that runs with the POSIX subsystem in Windows NT

*Default = /SUBSYSTEM:console*

## TraceIncludeDirectories

The TraceIncludeDirectories property is used to specify the directories that must be referenced in the makefile for projects where Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with INST\_INCLUDES.

*Default = \$(INCLUDE\_QUALIFIER)\$(OMROOT)\LangCpp\iom  
\$(INCLUDE\_QUALIFIER)\$(OMROOT)\LangCpp\iom*

## TraceInstLibs

When using one of Rational Rhapsody's IDE integrations, the TraceInstLibs property is used to specify the instrumentation framework libraries that are required when Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with INST\_LIBS.

*Default = \$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)iomtrace\$(LIB\_POSTFIX)\$(LIB\_EXT)  
\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)iomtrace\$(LIB\_POSTFIX)\$(LIB\_EXT)*

## TraceOxfLibs

When using one of Rational Rhapsody's IDE integrations, the TraceOxfLibs property is used to specify the framework libraries that are required when Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with OXF\_LIBS.

*Default = \$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)oxfinst\$(LIB\_POSTFIX)\$(LIB\_EXT)  
\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)omComAppl\$(LIB\_POSTFIX)\$(LIB\_EXT) winmm.lib*

## TracePreprocessor

The TracePreprocessor property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with INST\_FLAGS.

*Default = \$(DEFINE\_QUALIFIER)OMTRACER*

## **UpdateBuildSettingsInIDE**

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = True*

## **UseLIBCMT**

If you want your application to use the statically-linked library libcmtd.lib, rather than the dynamically-linked library msvcrt.lib, set the value of the property UseLIBCMT to True.

If the property is set to True, the makefile will include the option /MT. Otherwise, the makefile will include the option /MD.

*Default = False*

## **UseNewBuildOutputWindow**

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the CG::General::ShowLogViewAfterBuild property to True.

*Default = True*

## **UseNonZeroStdInputHandle**

The UseNonZeroStdInputHandle property is a Boolean value that specifies whether to use a non-zero standard input handle.

*Default = True*

## **UseRemoteHost**

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = False*

## UseTemplateName

In the declarations for members of a template class, some compilers require that you use precede the member type with the keyword "typename" for types with dependent names. Since support for this keyword varies between compilers, the UseTemplateName property is used to specify whether the "typename" keyword should be included in the generated code.

*Default = True*

## MSVCDLL

The MSVCDLL metaclass contains the Environment settings (compiler, framework libraries, and so on) for the MSVCDLL environment.

## AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)/LangCpp/osconfig/WIN32*

## AdditionalReservedWords

The AdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = \_\_asm \_\_finally naked \_\_based \_\_inline \_\_single\_inheritance \_\_cdecl \_\_int8 \_\_stdcall  
\_\_declspec \_\_int16 dllexport \_\_int32 \_\_try dllimport \_\_int64 \_\_virtual\_inheritance \_\_except \_\_leave  
\_\_fastcall \_\_multiple\_inheritance*

## BuildArgumentsInIDE

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the [lang]\_CG::[environment]::BuildCommandInIDE property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandInIDE

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration. To change this property, use the Configuration window in the browser - do not change it by using the Properties window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link.

*Default = Debug*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\$OMROOT/etc/Executer.exe" "%\""\$OMROOT\""%etc\msvcmake.bat msbuild.mak build \$CPU  
\$IDEVersion \"LIB\_PREFIX=\$(LibPrefix)\" \"%USE\_STL=FALSE\" \"%USE\_PDB=FALSE\"*

```
\ "BUILD_SET=$BuildCommandSet\ " \ "USE_LIBCMT=$UseLIBCMT\ " \ " "
```

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = False*

## COM

The COM property is a boolean property that controls the value of the SUBSYSTEM linker option in the generated makefile.

*Default = False*

## CompileSwitches

The CompileSwitches property specifies the compiler switches.

*Default = /I . /I \$OMDefaultSpecificationDirectory /I \$(OMROOT)\LangCpp /I \$(OMROOT)\LangCpp\oxf\nologo /W3 \$(ENABLE\_EH) \$(CRT\_FLAGS) \$OMCPPCompileCommandSet /D "\_AFXDLL" /D "WIN32" /D "\_CONSOLE" /D "\_MBCS" /D "\_WINDOWS" \$(INST\_FLAGS) \$(INCLUDE\_PATH) \$(INST\_INCLUDES) /c*

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = \$(CREATE\_OBJ\_DIR) \$(CPP) \$OMFileCPPCompileSwitches /Fo"\$OMFileObjPath" "\$OMFileImpPath"*

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*Default = /Zi /Od /D "\_DEBUG" \$(LIBCRT\_FLAG)d /Fd"\$(TARGET\_NAME)"*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*Default = /Ox /D"NDEBUG" \$(LIBCRT\_FLAG) /Fd"\$(TARGET\_NAME)"*

## CPU

The CPU property is used to specify the CPU of the target environment.

When building applications or framework libraries for MSVC9 environments, Rhapsody uses a variable called \$CPU which represents the value provided for the CPU property for these environments. This variable is provided as a parameter to the msvc9make.bat batch file, and is also used in the makefiles that are generated.

When building 32-bit applications, use the value x86 for this property.

When building 64-bit applications, use the value x64 for this property.

*Default = x86*

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: \$OMFileObjPath : \$OMFileImpPath "\*.bmp" "\*.ico" "\*.rc2"

*Default = \$OMFileObjPath : \$OMFileImpPath \$OMFileSpecPath \$OMFileDependencies*

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction an IDE such as Eclipse, you can use the EnableDebugIntegrationWithIDE property to specify whether or not the IDE debugger is to be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to Checked, the IDE debugger is used.

*Default = False*

## EndApplicationCode

When the main function ends, the system's exit() function is called automatically. The system function

calls the destructors of framework instances in the order it determines. The EndApplicationCode property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an endApplication() function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to endApplication(), it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The EntryPoint property specifies the name of the main program for a given environment.

*Default = main*

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. The ErrorMessageTokensFormat property defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. The ErrorMessageTokens property has three parameters, each with an integer value:

- TotalNumberOfTokens - The number of tokens in the regular expression
- FileTokenPosition - The position of the file name token in the expression
- LineTokenPosition - The position of the line number token in the expression

*Default = TotalNumberOfTokens=3,FileTokenPosition=1,LineTokenPosition=2*

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the C\_CG::<Environment>::ExeName property plus the value of the ExeExtension property.

*Default = .exe*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.



If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the [lang]\_CG::[environment]::ExeExtension property.

*Default = Blank*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements. The file inclusions are generated in the makefile.

*Default = \$OMSpecIncludeInElements \$OMImpIncludeInElements*

## IDEVersion

When you select one of the provided Microsoft environments for a configuration in your model, you use the IDEVersion property to tell Rational Rhapsody which version of MS Visual Studio you are using, 2008 or 2010.

The value of this property is passed as a parameter to the Build Framework command, and to the command used to build your application. The value of the property is also used in properties that affect the content of the generated makefile.

The possible values are:

- VC9 - to specify that you are using Visual Studio 2008
- VC10 - to specify that you are using Visual Studio 2010
- VC11 - to specify that you are using Visual Studio 2012

*Default = VC10*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment.

*Default = .cpp*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*Default = !INCLUDE*

## InvokeAnimatedExecutable

The AnimationPortRange value in the rhapsody.ini file makes it possible to use animation with multiple instances of Rhapsody running simultaneously.

Ordinarily, when you run an application with animation, the executable is launched using the command specified by the InvokeExecutable property. However, if you have specified a value greater than zero for AnimationPortRange in your rhapsody.ini file, the executable is launched using the command specified by the InvokeAnimatedExecutable property. By default, the value of InvokeAnimatedExecutable includes a reference to the port to use for animation.

*Default = \$executable -port \$port*

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*Default = "\$executable"*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, include the name of the property preceded by \$. As shown in the following example (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\msvc9make.bat \$makefile \$maketarget \$CPU\" "*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = False*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .dll*

## LibPrefix

The LibPrefix property is used to specify a prefix to add to generic names of runtime libraries. The value of this property is used in references to libraries in the generated makefile.

*Default = MS\$(IDEVersion)\$(CPU)\$(MT\_PREFIX)*

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

*Default = Blank*

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

*Default = Blank*

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

*Default = \$OMLinkCommandSet /NOLOGO*

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the [lang]\_CG::[environment]::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*Default = .mak*

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration. For example, the default content of the target type section of a C++ makefile for the Microsoft environment is as follows: ##### Target type (Debug/Release)

```
#####  
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease  
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease BuildSet=$OMBuildSet  
SUBSYSTEM=$OMSubSystem COM=$OMCOM RPFrameWorkDll=$OMRPFrameWorkDll  
ConfigurationCPPCompileSwitches= $OMReusableStatechartSwitches  
$OMConfigurationCPPCompileSwitches !IF "$ (RPFrameWorkDll)" == "True"  
ConfigurationCPPCompileSwitches= $(ConfigurationCPPCompileSwitches) /D "FRAMEWORK_DLL"  
!ENDIF !IF "$ (COM)" == "True" SUBSYSTEM=/SUBSYSTEM:windows !ENDIF
```

**Compilation Flags** The compilation flags section of the makefile contains the default compilation flags stored in the CompileSwitches property. For example, the default content of the compilation flags section of a C++ makefile for the Microsoft environment is as follows: ##### Compilation flags

```
#####  
INCLUDE_QUALIFIER=/I LIB_PREFIX=MS
```

**Commands Definitions** The commands definition section of the makefile specifies programs to execute from the makefile. For example, the default commands definition section of a C++ makefile for the Microsoft environment is as follows: ##### Commands definition

```
##### RMDIR = rmdir LIB_CMD=link.exe -lib  
LINK_CMD=link.exe LIB_FLAGS=$OMConfigurationLinkSwitches  
LINK_FLAGS=$OMConfigurationLinkSwitches $(SUBSYSTEM) / MACHINE:I386
```

**Generated Macros** The generated macros section of the makefile contains a variable that expands to the Rational Rhapsody-generated macros in the makefile. For example: ##### Generated macros

```
##### $OMContextMacros  
OBJ_DIR=$OMObjectsDir !IF "$ (OBJ_DIR)" != "" CREATE_OBJ_DIR=if not exist $(OBJ_DIR) mkdir  
$(OBJ_DIR) CLEAN_OBJ_DIR= if exist $(OBJ_DIR) $(RMDIR) $(OBJ_DIR) !ELSE  
CREATE_OBJ_DIR= CLEAN_OBJ_DIR= !ENDIF
```

The \$OMContextMacros keyword expands several macros in the makefile. Each makefile macro has its own keyword. You can use these keywords separately to customize the makefile. The

\$OMContextMacros variable modifies target-specific variables. Replace the \$OMContextMacros line in the MakeFileContent property with the following: `FLAGSFILE=$OMFlagsFile  
RULESFILE=$OMRulesFile OMROOT=$OMROOT C_EXT=$OMImplExt H_EXT=$OMSpecExt  
OBJ_EXT=$OMObjExt EXE_EXT=$OMExeExt LIB_EXT=$OMLibExt  
INSTRUMENTATION=$OMInstrumentation TIME_MODEL=$OMTimeModel  
TARGET_TYPE=$OMTargetType TARGET_NAME=$OMTargetName $OMAllDependencyRule  
TARGET_MAIN=$OMTargetMain LIBS=$OMLibs INCLUDE_PATH=$OMIncludePath  
ADDITIONAL_OBJS=$OMAdditionalObjs OBJS= $OMObjs`

**Predefined Macros** The predefined macros section of the makefile contains other macros than the Rational Rhapsody-generated macros specified in the generated macros section. For example, the default predefined macros section of a C++ makefile for the Microsoft environment is as follows:

```
##### Predefined macros #####
##### $(OBJS) : $(INST_LIBS)
$(OXF_LIBS) LIB_POSTFIX= !IF "$$(BuildSet)"=="Release" LIB_POSTFIX=R !ENDIF !IF
"$(TARGET_TYPE)" == "Executable" LinkDebug=$(LinkDebug) /DEBUG
LinkRelease=$(LinkRelease) /OPT:NOREF !ELSEIF "$(TARGET_TYPE)" == "Library"
LinkDebug=$(LinkDebug) /DEBUGTYPE:CV !ENDIF !IF "$(INSTRUMENTATION)" == "Animation"
INST_FLAGS=/D "OMANIMATOR" INST_INCLUDES=/I $(OMROOT)\LangCpp\om /I
$(OMROOT)\LangCpp\tom !IF "$(RPFrameWorkDll)" == "True" INST_LIBS=
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfanimdll$(LIB_POSTFIX) $(LIB_EXT)
!ELSE INST_LIBS= $(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomanim$(LIB_POSTFIX) $(LIB_EXT)
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfinst$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)" == "Tracing" INST_FLAGS=/D
"OMTRACER" INST_INCLUDES=/I $(OMROOT)\LangCpp\om /I $(OMROOT)\LangCpp\tom !IF
"$(RPFrameWorkDll)" == "True" INST_LIBS= OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxftracedll$(LIB_POSTFIX) $(LIB_EXT) !ELSE
INST_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)tomtrace$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomtrace$(LIB_POSTFIX) $(LIB_EXT) OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfinst$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)" == "None" INST_FLAGS=
INST_INCLUDES= INST_LIBS= !IF "$(RPFrameWorkDll)" == "True"
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfdll$(LIB_POSTFIX) $(LIB_EXT) !ELSE
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB= !ELSE !ERROR An invalid Instrumentation $(INSTRUMENTATION) is specified.
!ENDIF
```

**Generated Dependencies** The generated dependencies section of the makefile contains a variable that expands to Rational Rhapsody-generated dependencies and compilation instructions. For example, the generated dependencies section of a C++ makefile for the Microsoft environment is as follows:

```
##### Generated dependencies #####
##### $OMContextDependencies
$OMFileObjPath : $OMMainImplementationFile $(OBJS) $(CPP) $(ConfigurationCPPCompileSwitches)
/Fo"$OMFileObjPath" $OMMainImplementationFile
```

**Linking Instructions** The linking instructions section of the makefile contains the predefined linking instructions. For example, the default linking instructions section of a C++ makefile for the Microsoft environment is as follows:

```
##### Linking instructions #####
#####
$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS) $OMFileObjPath
$OMMakefileName $OMModelLibs @echo Linking $(TARGET_NAME)$(EXE_EXT) $(LINK_CMD)
$OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \ $(OXF_LIBS) \
```

```
$(SOCK_LIB) \ $(LINK_FLAGS) /out:$$(TARGET_NAME)$$(EXE_EXT)
$(TARGET_NAME)$$(LIB_EXT) : $(OBJS) $(ADDITIONAL_OBJS) $OMMakefileName @echo
Building library $@ $(LIB_CMD) $(LIB_FLAGS) /out:$$(TARGET_NAME)$$(LIB_EXT) $(OBJS)
$(ADDITIONAL_OBJS) clean: @echo Cleanup $OMCleanOBJS if exist $OMFileObjPath erase
$OMFileObjPath if exist *$(OBJ_EXT) erase *$(OBJ_EXT) if exist $(TARGET_NAME).pdb erase
$(TARGET_NAME).pdb if exist $(TARGET_NAME)$$(LIB_EXT) erase
$(TARGET_NAME)$$(LIB_EXT) if exist $(TARGET_NAME).ilk erase $(TARGET_NAME).ilk if exist
$(TARGET_NAME)$$(EXE_EXT) erase $(TARGET_NAME)$$(EXE_EXT) $(CLEAN_OBJ_DIR)
```

```
Default = ##### Target type (Debug/Release) #####
#####
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease BuildSet=$OMBuildSet
SUBSYSTEM=$OMSubSystem RPFramWorkDll=$OMRPFramWorkDll
DEF_EXT=$OMDEFExtension DLL_EXT=$OMDllExtension
ConfigurationCPPCompileSwitches=$OMReusableStatechartSwitches
$OMConfigurationCPPCompileSwitches !IF "$$(RPFramWorkDll)" == "True"
ConfigurationCPPCompileSwitches=$(ConfigurationCPPCompileSwitches) /D "FRAMEWORK_DLL"
!ENDIF ##### Compilation flags #####
##### INCLUDE_QUALIFIER=/I
LIB_PREFIX=$LibPrefix !IF "$$(UseLIBCMT)" == "True" MT_PREFIX=MT LIBCRT_FLAG=/MT
!ELSE MT_PREFIX= LIBCRT_FLAG=/MD !ENDIF CRT_FLAGS=/D
"_CRT_SECURE_NO_DEPRECATED" /D "_CRT_SECURE_NO_WARNINGS" ENABLE_EH=/EHa
WINMM_LIB=winmm.lib ##### Commands definition #####
##### RMDIR = rmdir
DLL_CMD=link.exe -dll LINK_CMD=link.exe DLL_FLAGS=$OMConfigurationLinkSwitches
LINK_FLAGS=$OMConfigurationLinkSwitches $(SUBSYSTEM) /MACHINE:$CPU #####
Generated macros #####
$OMContextMacros OBJ_DIR=$OMObjectsDir !IF "$$(OBJ_DIR)" != "" CREATE_OBJ_DIR=if not exist
$(OBJ_DIR) mkdir $(OBJ_DIR) CLEAN_OBJ_DIR= if exist $(OBJ_DIR) $(RMDIR) $(OBJ_DIR) !ELSE
CREATE_OBJ_DIR= CLEAN_OBJ_DIR= !ENDIF ##### Predefined macros
#####
##### !IF "$$(OBJS)"
!= "" $(OBJS) : $(FLAGSFILE) $(RULESFILE) $(INST_LIBS) $(OXF_LIBS) !ENDIF LIB_EXT=.lib
LIB_POSTFIX= !IF "$$(BuildSet)" == "Release" LIB_POSTFIX=R !ENDIF !IF "$$(TARGET_TYPE)" ==
"Executable" LinkDebug=$(LinkDebug) /DEBUG LinkRelease=$(LinkRelease) /OPT:NOREF !ELSEIF
"$$(TARGET_TYPE)" == "Library" LinkDebug=$(LinkDebug) /DEBUG LinkRelease=$(LinkRelease)
/OPT:NOREF !ENDIF !IF "$$(TIME_MODEL)" == "Simulated" TIM_EXT= !ELSEIF
"$$(TIME_MODEL)" == "RealTime" TIM_EXT= !ELSE !ERROR An invalid Time Model
"$$(TIME_MODEL)" is specified. !ENDIF !IF "$$(INSTRUMENTATION)" == "Animation"
INST_FLAGS=/D "OMANIMATOR" INST_INCLUDES=/I $(OMROOT)\LangCpp\iom /I
$(OMROOT)\LangCpp\iom !IF "$$(RPFramWorkDll)" == "True" INST_LIBS=
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfanimdll$(LIB_POSTFIX)$$(LIB_EXT) !ELSE
INST_LIBS= $(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomanim$(LIB_POSTFIX)$$(LIB_EXT)
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$(TIM_EXT)inst$(LIB_POSTFIX)$$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX)$$(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$$(INSTRUMENTATION)" == "Tracing" INST_FLAGS=/D
"OMTRACER" INST_INCLUDES=/I $(OMROOT)\LangCpp\iom /I $(OMROOT)\LangCpp\iom !IF
"$$(RPFramWorkDll)" == "True" INST_LIBS= OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfracedll$(LIB_POSTFIX)$$(LIB_EXT) !ELSE
INST_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)tomtrace$(LIB_POSTFIX)$$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomtrace$(LIB_POSTFIX)$$(LIB_EXT) OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$(TIM_EXT)inst$(LIB_POSTFIX)$$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX)$$(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$$(INSTRUMENTATION)" == "None" INST_FLAGS=
```



```

INST_INCLUDES= INST_LIBS= !IF "$(RPFrameWorkDll)" == "True"
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfdll$(LIB_POSTFIX)$(LIB_EXT) !ELSE
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$(TIM_EXT)$(LIB_POSTFIX)$(LIB_EXT)
!ENDIF SOCK_LIB= !ELSE !ERROR An invalid Instrumentation $(INSTRUMENTATION) is specified.
!ENDIF ##### Generated dependencies #####
#####
$OMContextDependencies !IF "$(TARGET_MAIN)" != "" CLEAN_MAIN_OBJ=if exist $OMFileObjPath
erase $OMFileObjPath $OMFileObjPath : $OMMainImplementationFile $(OBJS) $(FLAGSFILE)
$(RULESFILE) $(CPP) $(ConfigurationCPPCompileSwitches) /Fo"$OMFileObjPath"
$OMMainImplementationFile !ELSE CLEAN_MAIN_OBJ= !ENDIF #####
Linking instructions #####
##### !IF
"$(TARGET_NAME)" != "" $(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS)
$OMFileObjPath $OMMakefileName $OMModelLibs @echo Linking $(TARGET_NAME)$(EXE_EXT)
$(LINK_CMD) $OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \
$(OXF_LIBS) \ $(SOCK_LIB) \ $(WINMM_LIB) \ $(LINK_FLAGS) /out:$(TARGET_NAME)$(EXE_EXT)
if exist $(TARGET_NAME)$(EXE_EXT).manifest mt.exe -manifest
$(TARGET_NAME)$(EXE_EXT).manifest -outputresource:$(TARGET_NAME)$(EXE_EXT);1
$(TARGET_NAME)$(DLL_EXT) : $(OBJS) $(ADDITIONAL_OBJS) $OMMakefileName @echo Building
library @@ $(DLL_CMD) $(DLL_FLAGS) $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \
$(OXF_LIBS) \ $(SOCK_LIB) \ $(WINMM_LIB) \ /out:$(TARGET_NAME)$(DLL_EXT) if exist
$(TARGET_NAME)$(DLL_EXT).manifest mt.exe -manifest $(TARGET_NAME)$(DLL_EXT).manifest
-outputresource:$(TARGET_NAME)$(DLL_EXT);2 !ENDIF clean: @echo Cleanup $OMCleanOBJS
$(CLEAN_MAIN_OBJ) if exist *$(OBJ_EXT) erase *$(OBJ_EXT) if exist $(TARGET_NAME).pdb erase
$(TARGET_NAME).pdb if exist $(TARGET_NAME)$(LIB_EXT) erase $(TARGET_NAME)$(LIB_EXT) if
exist $(TARGET_NAME).ilk erase $(TARGET_NAME).ilk if exist $(TARGET_NAME)$(EXE_EXT) erase
$(TARGET_NAME)$(EXE_EXT) if exist $(TARGET_NAME)$(EXE_EXT).manifest erase
$(TARGET_NAME)$(DLL_EXT).manifest if exist $(TARGET_NAME)$(DLL_EXT).manifest erase
$(TARGET_NAME)$(DLL_EXT).manifest $(CLEAN_OBJ_DIR)

```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the [lang]\_CG::[environment]::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*Default = Blank*

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## ObjCleanCommand

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default = if exist \$OMFileObjPath erase \$OMFileObjPath*

## ObjectName

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = Blank*

## ObjectsDirectory

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = Blank*

## ObjExtension

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*Default = .obj*

## OSFileSystemCaseSensitive

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = False*

## ParseErrorDescript

The ParseErrorDescript property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = ([^()]+)([()([0-9]+)]+)[ :] (error/warning/fatal error) (.\*)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and



(Error|Fatal|Warning)

Along with the ErrorMessageTokensFormat property, the ParseErrorMessage property specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = ([^()]+)([()([0-9]+)] [[:] (error|warning|fatal error)*

## ParseMakeError

The ParseMakeError property is used to define a regular expression that represents the format of make process or linker error messages. This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = (NMAKE)(.\*)(fatal error)*

## ParseSeverityError

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = [[:] (error|fatal error)*

## ParseSeverityWarning

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = [[:] (warning)*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation.

*Default = \*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = True*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running the product, whereas the target is the machine running the application. To run remotely, the UseRemoteHost property must be set to Checked.

If the UseRemoteHost property is set to Checked and the RemoteHost property is blank, the current host name is used for the remote host. You can use this as a workaround if you have problems running animated applications on Windows 95.

You can leave the RemoteHost property blank if both the application and Rational Rhapsody are running on the same machine.

*Default = Blank*

## ReusableStatechartSwitches

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts.

*Default = /D "OM\_REUSABLE\_STATECHART\_IMPLEMENTATION"*

## RPFrameworkDll

The RPFrameworkDll property can be used to specify that an application should use the dll generated for the OXF framework.

When the property is set to True, the makefile that is generated for the application includes linkage to the relevant OXF framework dll, based on the instrumentation mode specified for the configuration: oxfanimdll.dll if animation is used, oxfracedll.dll if tracing is used, and oxfdll.dll if the configuration does not use animation or tracing.

*Default = False*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*Default = .h*

## SubSystem

The SubSystem property is a string that defines the type of the program for the Microsoft linker. The possible values are as follows:

- **CONSOLE** - Used for a Win32 character-mode application
- **WINDOWS** - Used for an application that does not require a console
- **NATIVE** - Applies device drivers for Windows NT
- **POSIX** - Creates an application that runs with the POSIX subsystem in Windows NT

*Default = /SUBSYSTEM:console*

## **UpdateBuildSettingsInIDE**

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = True*

## **UseLIBCMT**

If you want your application to use the statically-linked library libcmt.lib, rather than the dynamically-linked library msvcrt.lib, set the value of the property UseLIBCMT to True.

If the property is set to True, the makefile will include the option /MT. Otherwise, the makefile will include the option /MD.

*Default = False*

## **UseNewBuildOutputWindow**

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the CG::General::ShowLogViewAfterBuild property to True.

*Default = True*

## **UseNonZeroStdInputHandle**

The UseNonZeroStdInputHandle property is a Boolean value that specifies whether to use a non-zero standard input handle.

*Default = True*

## UseRemoteHost

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = False*

## UseTemplateName

In the declarations for members of a template class, some compilers require that you use precede the member type with the keyword "typename" for types with dependent names. Since support for this keyword varies between compilers, the UseTemplateName property is used to specify whether the "typename" keyword should be included in the generated code.

*Default = True*

## MSVCStandardLibrary

The MSVCStandardLibrary metaclass contains the Environment settings (compiler, framework libraries, and so on) for the MSVCStandardLibrary environment.

## AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)/LangCpp/osconfig/WIN32*

## AdditionalReservedWords

The AdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = \_\_asm \_\_finally naked \_\_based \_\_inline \_\_single\_inheritance \_\_cdecl \_\_int8 \_\_stdcall \_\_declspec \_\_int16 thread dllexport \_\_int32 \_\_try dllimport \_\_int64 \_\_virtual\_inheritance \_\_except \_\_leave \_\_fastcall \_\_multiple\_inheritance*

## BuildArgumentsInIDE

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the [lang]\_CG::[environment]::BuildCommandInIDE property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandInIDE

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration. To change this property, use the Configuration window in the browser - do not change it by using the Properties window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link.

*Default = Debug*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

```
Default = "$OMROOT/etc/Executer.exe" "\""$OMROOT\""\etc\msvcmake.bat msbuild.mak build $CPU  
$IDEVersion \"LIB_PREFIX=$(LibPrefix)\" \"USE_STL=TRUE\" \"USE_PDB=FALSE\"  
\"BUILD_SET=$BuildCommandSet\" \"USE_LIBCMT=$UseLIBCMT\" \" \"
```

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

```
Default = False
```

## CompileSwitches

The CompileSwitches property specifies the compiler switches.

```
Default = /I . /I $OMDefaultSpecificationDirectory /I $(OMROOT)\LangCpp /I  
$(OMROOT)\LangCpp\oxf\nologo /W3 $(ENABLE_EH) $(CRT_FLAGS) $OMCPPCompileCommandSet  
/D "_AFXDLL" /D "WIN32" /D "_CONSOLE" /D "_MBCS" /D "_WINDOWS" /D "OM_USE_STL"  
$(INST_FLAGS) $(INCLUDE_PATH) $(INST_INCLUDES) /c
```

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

```
Default = $(CREATE_OBJ_DIR) $(CPP) $OMFileCPPCompileSwitches /Fo"$OMFileObjPath"  
"$OMFileImpPath"
```

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

```
Default = /Zi /Od /D "_DEBUG" $(LIBCRT_FLAG)d /Fd"$(TARGET_NAME)"
```

## CPPCompileRelease

The `CPPCompileRelease` property specifies additional compilation flags for a configuration set to Release mode.

*Default = /Ox /D"NDEBUG" \$(LIBCRT\_FLAG) /Fd"\$(TARGET\_NAME)"*

## CPU

The `CPU` property is used to specify the CPU of the target environment.

When building applications or framework libraries for MSVC9 environments, Rhapsody uses a variable called `$CPU` which represents the value provided for the `CPU` property for these environments. This variable is provided as a parameter to the `msvc9make.bat` batch file, and is also used in the makefiles that are generated.

When building 32-bit applications, use the value `x86` for this property.

When building 64-bit applications, use the value `x64` for this property.

*Default = x86*

## DependencyRule

The `DependencyRule` property specifies how file dependencies for a configuration are generated in the makefile.

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: `$OMFileObjPath : $OMFileImpPath "*.bmp" "*.ico" "*.rc2"`

*Default = \$OMFileObjPath : \$OMFileImpPath \$OMFileSpecPath \$OMFileDependencies*

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction an IDE such as Eclipse, you can use the `EnableDebugIntegrationWithIDE` property to specify whether or not the IDE debugger is to be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to `Checked`, the IDE debugger is used.

*Default = False*

## EndApplicationCode

When the main function ends, the system's `exit()` function is called automatically. The system function calls the destructors of framework instances in the order it determines. The `EndApplicationCode` property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an `endApplication()` function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to `endApplication()`, it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The `EntryPoint` property specifies the name of the main program for a given environment.

*Default = main*

## ErrorMessageTokensFormat

The `ErrorMessageTokensFormat`, working with the `ParseErrorMessage` property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. The `ErrorMessageTokensFormat` property defines the number and location of tokens within the regular expression defined by the `ParseErrorMessage` property. The `ErrorMessageTokens` property has three parameters, each with an integer value:

- `TotalNumberOfTokens` - The number of tokens in the regular expression
- `FileTokenPosition` - The position of the file name token in the expression
- `LineTokenPosition` - The position of the line number token in the expression

*Default = TotalNumberOfTokens=3,FileTokenPosition=1,LineTokenPosition=2*

## ExeExtension

The `ExeExtension` property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the `C_CG::<Environment>::ExeName` property plus the value of the `ExeExtension` property.

*Default = .exe*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the `ExeName` property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.



Note that the full name of the executable program is composed of the value of this property plus the value of the [lang]\_CG::[environment]::ExeExtension property.

*Default = Blank*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements. The file inclusions are generated in the makefile.

*Default = \$OMSpecIncludeInElements \$OMImpIncludeInElements*

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile.

Note: If you select Release Build Set, these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

*Default = \$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)WebComponents\$(LIB\_POSTFIX)\$(LIB\_EXT), \$(OMROOT)\lib\\$(LIB\_PREFIX)WebServices\$(LIB\_POSTFIX)\$(LIB\_EXT), ws2\_32\$(LIB\_EXT)*

## IDEVersion

When you select one of the provided Microsoft environments for a configuration in your model, you use the IDEVersion property to tell Rational Rhapsody which version of MS Visual Studio you are using, 2008 or 2010.

The value of this property is passed as a parameter to the Build Framework command, and to the command used to build your application. The value of the property is also used in properties that affect the content of the generated makefile.

The possible values are:

- VC9 - to specify that you are using Visual Studio 2008
- VC10 - to specify that you are using Visual Studio 2010
- VC11 - to specify that you are using Visual Studio 2012

*Default = VC10*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment.

*Default = .cpp*

## **Include**

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*Default = !INCLUDE*

## **InvokeAnimatedExecutable**

The AnimationPortRange value in the rhapsody.ini file makes it possible to use animation with multiple instances of Rhapsody running simultaneously.

Ordinarily, when you run an application with animation, the executable is launched using the command specified by the InvokeExecutable property. However, if you have specified a value greater than zero for AnimationPortRange in your rhapsody.ini file, the executable is launched using the command specified by the InvokeAnimatedExecutable property. By default, the value of InvokeAnimatedExecutable includes a reference to the port to use for animation.

*Default = \$executable -port \$port*

## **InvokeExecutable**

The InvokeExecutable property specifies the command used to run an executable file.

*Default = "\$executable"*

## **InvokeMake**

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, include the name of the property preceded by \$. As shown in the following example (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\msvc9make.bat \$makefile \$maketarget \$CPU\" "*

## **IsFileNameShort**

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = False*

## **LibExtension**

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .lib*

## **LibPrefix**

The LibPrefix property is used to specify a prefix to add to generic names of runtime libraries. The value of this property is used in references to libraries in the generated makefile.

*Default = MS\$(IDEVersion)\$(CPU)\$(MT\_PREFIX)Stl*

## **LinkDebug**

The LinkDebug property specifies the special link switches used to link in debug mode.

*Default = Blank*

## **LinkRelease**

The LinkRelease property specifies the special link switches used to link in release mode.

*Default = Blank*

## **LinkSwitches**

The LinkSwitches property specifies the standard link switches used to link in any mode.

*Default = \$OMLinkCommandSet /NOLOGO*

## **MakeExtension**

The MakeExtension property can be used to specify the file extension you would like to use for the

makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the [lang]\_CG::[environment]::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*Default = .mak*

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration. For example, the default content of the target type section of a C++ makefile for the Microsoft environment is as follows: ##### Target type (Debug/Release)

```
#####  
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease  
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease BuildSet=$OMBuildSet  
SUBSYSTEM=$OMSubSystem COM=$OMCOM RPFrameWorkDll=$OMRPFrameWorkDll  
ConfigurationCPPCompileSwitches= $OMReusableStatechartSwitches  
$OMConfigurationCPPCompileSwitches !IF "$ (RPFrameWorkDll)" == "True"  
ConfigurationCPPCompileSwitches= $(ConfigurationCPPCompileSwitches) /D "FRAMEWORK_DLL"  
!ENDIF !IF "$ (COM)" == "True" SUBSYSTEM=/SUBSYSTEM:windows !ENDIF
```

**Compilation Flags** The compilation flags section of the makefile contains the default compilation flags stored in the CompileSwitches property. For example, the default content of the compilation flags section of a C++ makefile for the Microsoft environment is as follows: ##### Compilation flags

```
#####  
INCLUDE_QUALIFIER=/I LIB_PREFIX=MS
```

**Commands Definitions** The commands definition section of the makefile specifies programs to execute from the makefile. For example, the default commands definition section of a C++ makefile for the Microsoft environment is as follows: ##### Commands definition #####

```
##### RMDIR = rmdir LIB_CMD=link.exe -lib  
LINK_CMD=link.exe LIB_FLAGS=$OMConfigurationLinkSwitches  
LINK_FLAGS=$OMConfigurationLinkSwitches $(SUBSYSTEM) / MACHINE:I386
```

**Generated Macros** The generated macros section of the makefile contains a variable that expands to the

Rational Rhapsody-generated macros in the makefile. For example: ##### Generated macros  
##### SOMContextMacros  
OBJ\_DIR=\$OMObjectsDir !IF "\$(OBJ\_DIR)"!=" CREATE\_OBJ\_DIR=if not exist \$(OBJ\_DIR) mkdir  
\$(OBJ\_DIR) CLEAN\_OBJ\_DIR= if exist \$(OBJ\_DIR) \$(RMDIR) \$(OBJ\_DIR) !ELSE  
CREATE\_OBJ\_DIR= CLEAN\_OBJ\_DIR= !ENDIF

The \$OMContextMacros keyword expands several macros in the makefile. Each makefile macro has its own keyword. You can use these keywords separately to customize the makefile. The \$OMContextMacros variable modifies target-specific variables. Replace the \$OMContextMacros line in the MakeFileContent property with the following: FLAGSFILE=\$OMFlagsFile  
RULESFILE=\$OMRulesFile OMROOT=\$OMROOT C\_EXT=\$OMImplExt H\_EXT=\$OMSpecExt  
OBJ\_EXT=\$OMObjExt EXE\_EXT=\$OMExeExt LIB\_EXT=\$OMLibExt  
INSTRUMENTATION=\$OMInstrumentation TIME\_MODEL=\$OMTimeModel  
TARGET\_TYPE=\$OMTargetType TARGET\_NAME=\$OMTargetName \$OMAllDependencyRule  
TARGET\_MAIN=\$OMTargetMain LIBS=\$OMLibs INCLUDE\_PATH=\$OMIncludePath  
ADDITIONAL\_OBJS=\$OMAdditionalObjs OBJS= \$OMObjs

**Predefined Macros** The predefined macros section of the makefile contains other macros than the Rational Rhapsody-generated macros specified in the generated macros section. For example, the default predefined macros section of a C++ makefile for the Microsoft environment is as follows:

```
##### Predefined macros #####
##### $(OBJS) : $(INST_LIBS)
$(OXF_LIBS) LIB_POSTFIX= !IF "$(BuildSet)"=="Release" LIB_POSTFIX=R !ENDIF !IF
"$(TARGET_TYPE)" == "Executable" LinkDebug=$(LinkDebug) /DEBUG
LinkRelease=$(LinkRelease) /OPT:NOREF !ELSEIF "$(TARGET_TYPE)" == "Library"
LinkDebug=$(LinkDebug) /DEBUGTYPE:CV !ENDIF !IF "$(INSTRUMENTATION)" == "Animation"
INST_FLAGS=/D "OMANIMATOR" INST_INCLUDES=/I $(OMROOT)\LangCpp\om /I
$(OMROOT)\LangCpp\om !IF "$(RPFrameWorkDll)" == "True" INST_LIBS=
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxanimdll$(LIB_POSTFIX) $(LIB_EXT)
!ELSE INST_LIBS= $(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomanim$(LIB_POSTFIX) $(LIB_EXT)
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfinst$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)" == "Tracing" INST_FLAGS=/D
"OMTRACER" INST_INCLUDES=/I $(OMROOT)\LangCpp\om /I $(OMROOT)\LangCpp\om !IF
"$(RPFrameWorkDll)" == "True" INST_LIBS= OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfracedll$(LIB_POSTFIX) $(LIB_EXT) !ELSE
INST_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)tomtrace$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomtrace$(LIB_POSTFIX) $(LIB_EXT) OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfinst$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)" == "None" INST_FLAGS=
INST_INCLUDES= INST_LIBS= !IF "$(RPFrameWorkDll)" == "True"
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfdll$(LIB_POSTFIX) $(LIB_EXT) !ELSE
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB= !ELSE !ERROR An invalid Instrumentation $(INSTRUMENTATION) is specified.
!ENDIF
```

**Generated Dependencies** The generated dependencies section of the makefile contains a variable that expands to Rational Rhapsody-generated dependencies and compilation instructions. For example, the generated dependencies section of a C++ makefile for the Microsoft environment is as follows:

```
##### Generated dependencies #####
##### SOMContextDependencies
$OMFileObjPath : $OMMainImplementationFile $(OBJS) $(CPP) $(ConfigurationCPPCompileSwitches)
/Fo"$OMFileObjPath" $OMMainImplementationFile
```

Linking Instructions The linking instructions section of the makefile contains the predefined linking instructions. For example, the default linking instructions section of a C++ makefile for the Microsoft environment is as follows: ##### Linking instructions #####  
#####  
\$(TARGET\_NAME)\$(EXE\_EXT): \$(OBJS) \$(ADDITIONAL\_OBJS) \$OMFileObjPath  
\$OMMakefileName \$OMModelLibs @echo Linking \$(TARGET\_NAME)\$(EXE\_EXT) \$(LINK\_CMD)  
\$OMFileObjPath \$(OBJS) \$(ADDITIONAL\_OBJS) \ \$(LIBS) \ \$(INST\_LIBS) \ \$(OXF\_LIBS) \  
\$(SOCK\_LIB) \ \$(LINK\_FLAGS) /out:\$(TARGET\_NAME)\$(EXE\_EXT)  
\$(TARGET\_NAME)\$(LIB\_EXT) : \$(OBJS) \$(ADDITIONAL\_OBJS) \$OMMakefileName @echo  
Building library \$@ \$(LIB\_CMD) \$(LIB\_FLAGS) /out:\$(TARGET\_NAME)\$(LIB\_EXT) \$(OBJS)  
\$(ADDITIONAL\_OBJS) clean: @echo Cleanup \$OMCleanOBJS if exist \$OMFileObjPath erase  
\$OMFileObjPath if exist \*\$(OBJ\_EXT) erase \*\$(OBJ\_EXT) if exist \$(TARGET\_NAME).pdb erase  
\$(TARGET\_NAME).pdb if exist \$(TARGET\_NAME)\$(LIB\_EXT) erase  
\$(TARGET\_NAME)\$(LIB\_EXT) if exist \$(TARGET\_NAME).ilk erase \$(TARGET\_NAME).ilk if exist  
\$(TARGET\_NAME)\$(EXE\_EXT) erase \$(TARGET\_NAME)\$(EXE\_EXT) \$(CLEAN\_OBJ\_DIR)

Default = ##### Target type (Debug/Release) #####  
#####  
CPPCompileDebug=\$OMCPPCompileDebug CPPCompileRelease=\$OMCPPCompileRelease  
LinkDebug=\$OMLinkDebug LinkRelease=\$OMLinkRelease BuildSet=\$OMBuildSet  
ConfigurationCPPCompileSwitches=\$OMReusableStatechartSwitches  
\$OMConfigurationCPPCompileSwitches ##### Compilation flags  
##### !IF  
"\$(UseLIBCMT)" == "True" MT\_PREFIX=MT LIBCRT\_FLAG=/MT !ELSE MT\_PREFIX=  
LIBCRT\_FLAG=/MD !ENDIF LIB\_PREFIX=\$(LibPrefix) INCLUDE\_QUALIFIER=/I CRT\_FLAGS=/D  
"\_CRT\_SECURE\_NO\_DEPRECATED" /D "\_CRT\_SECURE\_NO\_WARNINGS" ENABLE\_EH=/EHa  
WINMM\_LIB=winmm.lib ##### Commands definition #####  
##### RMDIR = rmdir  
LIB\_CMD=link.exe -lib LINK\_CMD=link.exe LIB\_FLAGS=\$OMConfigurationLinkSwitches  
LINK\_FLAGS=\$OMConfigurationLinkSwitches /SUBSYSTEM:console /MACHINE:\$CPU  
/nodefaultlib:"libc.lib" ##### Generated macros #####  
##### \$OMContextMacros  
OBJ\_DIR=\$OMObjectsDir !IF "\$(OBJ\_DIR)" != "" CREATE\_OBJ\_DIR=if not exist \$(OBJ\_DIR) mkdir  
\$(OBJ\_DIR) CLEAN\_OBJ\_DIR= if exist \$(OBJ\_DIR) \$(RMDIR) \$(OBJ\_DIR) !ELSE  
CREATE\_OBJ\_DIR= CLEAN\_OBJ\_DIR= !ENDIF ##### Predefined macros  
#####  
##### !IF "\$(OBJS)"  
!= "" \$(OBJS) : \$(INST\_LIBS) \$(OXF\_LIBS) !ENDIF LIB\_POSTFIX= !IF "\$(BuildSet)" == "Release"  
LIB\_POSTFIX=R !ENDIF !IF "\$(TARGET\_TYPE)" == "Executable" LinkDebug=\$(LinkDebug)  
/DEBUG LinkRelease=\$(LinkRelease) /OPT:NOREF !ELSEIF "\$(TARGET\_TYPE)" == "Library"  
LinkDebug=\$(LinkDebug) !ENDIF !IF "\$(INSTRUMENTATION)" == "Animation" INST\_FLAGS=/D  
"OMANIMATOR" INST\_INCLUDES=/I \$(OMROOT)\LangCpp\iom /I \$(OMROOT)\LangCpp\iom  
INST\_LIBS= \$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)iomanim\$(LIB\_POSTFIX)\$(LIB\_EXT)  
OXF\_LIBS= \$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)iofinst\$(LIB\_POSTFIX)\$(LIB\_EXT)  
\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)iomComAppl\$(LIB\_POSTFIX)\$(LIB\_EXT)  
SOCK\_LIB=wsock32.lib !ELSEIF "\$(INSTRUMENTATION)" == "Tracing" INST\_FLAGS=/D  
"OMTRACER" INST\_INCLUDES=/I \$(OMROOT)\LangCpp\iom /I \$(OMROOT)\LangCpp\iom  
INST\_LIBS= \$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)iomtrace\$(LIB\_POSTFIX)\$(LIB\_EXT)  
\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)iomtrace\$(LIB\_POSTFIX)\$(LIB\_EXT) OXF\_LIBS=  
\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)iofinst\$(LIB\_POSTFIX)\$(LIB\_EXT)  
\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)iomComAppl\$(LIB\_POSTFIX)\$(LIB\_EXT)  
SOCK\_LIB=wsock32.lib !ELSEIF "\$(INSTRUMENTATION)" == "None" INST\_FLAGS=  
INST\_INCLUDES= INST\_LIBS=  
OXF\_LIBS= \$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)iof\$(LIB\_POSTFIX)\$(LIB\_EXT) SOCK\_LIB=  
!ELSE !ERROR An invalid Instrumentation \$(INSTRUMENTATION) is specified. !ENDIF



```
##### Generated dependencies #####
#####
$OMContextDependencies $OMFileObjPath : $OMMainImplementationFile $(OBJS) $(CPP)
$(ConfigurationCPPCompileSwitches) /Fo"$OMFileObjPath" $OMMainImplementationFile
##### Linking instructions #####
#####
$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS) $OMFileObjPath
$OMMakefileName $OMModelLibs @echo Linking $(TARGET_NAME)$(EXE_EXT) $(LINK_CMD)
$OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \ $(OXF_LIBS) \
$(SOCK_LIB) \ $(WINMM_LIB) \ $(LINK_FLAGS) /out:$(TARGET_NAME)$(EXE_EXT) if exist
$(TARGET_NAME)$(EXE_EXT).manifest mt.exe -manifest $(TARGET_NAME)$(EXE_EXT).manifest
-outputresource:$(TARGET_NAME)$(EXE_EXT);1 $(TARGET_NAME)$(LIB_EXT) : $(OBJS)
$(ADDITIONAL_OBJS) $OMMakefileName @echo Building library @$ $(LIB_CMD) $(LIB_FLAGS)
/out:$(TARGET_NAME)$(LIB_EXT) $(OBJS) $(ADDITIONAL_OBJS) $(LIBS) clean: @echo Cleanup
$OMCleanOBJS if exist $OMFileObjPath erase $OMFileObjPath if exist *$(OBJ_EXT) erase
*$(OBJ_EXT) if exist $(TARGET_NAME).pdb erase $(TARGET_NAME).pdb if exist
$(TARGET_NAME)$(LIB_EXT) erase $(TARGET_NAME)$(LIB_EXT) if exist $(TARGET_NAME).ilk
erase $(TARGET_NAME).ilk if exist $(TARGET_NAME)$(EXE_EXT) erase
$(TARGET_NAME)$(EXE_EXT) if exist $(TARGET_NAME)$(EXE_EXT).manifest erase
$(TARGET_NAME)$(EXE_EXT).manifest $(CLEAN_OBJ_DIR)
```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the [lang]\_CG::[environment]::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*Default = Blank*

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## ObjCleanCommand

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default = if exist \$OMFileObjPath erase \$OMFileObjPath*

## ObjectName

The `ObjectName` property specifies an alternative name for the compiled object file in the generated makefile.

*Default = Blank*

## ObjectsDirectory

The `ObjectsDirectory` property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = Blank*

## ObjExtension

The `ObjExtension` property specifies the extension appended to compiled object files for a given environment.

*Default = .obj*

## OSFileSystemCaseSensitive

The `OSFileSystemCaseSensitive` property specifies whether the OS file system for a given environment is case sensitive.

*Default = False*

## ParseErrorDescript

The `ParseErrorDescript` property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = ([^()]+)[([0-9]+)] [:] (error/warning/fatal error) (.\*)*

## ParseErrorMessage

The `ParseErrorMessage` property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning)

Along with the `ErrorMessageTokensFormat` property, the `ParseErrorMessage` property specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = ([^()]+)[([0-9]+)] [:] (error/warning/fatal error)*



## ParseMakeError

The ParseMakeError property is used to define a regular expression that represents the format of make process or linker error messages. This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = (NMAKE)(.\*)(fatal error)*

## ParseSeverityError

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^()]+)([()([0-9]+)] [[:] (error|fatal error)*

## ParseSeverityWarning

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^()]+)([()([0-9]+)] [[:] (warning)*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation.

*Default = \*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = True*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running the product, whereas the target is the machine running the application. To run remotely, the UseRemoteHost property must be set to Checked.

If the UseRemoteHost property is set to Checked and the RemoteHost property is blank, the current host name is used for the remote host. You can use this as a workaround if you have problems running

animated applications on Windows 95.

You can leave the RemoteHost property blank if both the application and Rational Rhapsody are running on the same machine.

*Default = Blank*

## ReusableStatechartSwitches

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts.

*Default = /D "OM\_REUSABLE\_STATECHART\_IMPLEMENTATION"*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*Default = .h*

## UpdateBuildSettingsInIDE

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = True*

## UseLIBCMT

If you want your application to use the statically-linked library libcmt.lib, rather than the dynamically-linked library msvcrt.lib, set the value of the property UseLIBCMT to True.

If the property is set to True, the makefile will include the option /MT. Otherwise, the makefile will include the option /MD.

*Default = False*

## UseNewBuildOutputWindow

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the `CG::General::ShowLogViewAfterBuild` property to `True`.

*Default = True*

## UseNonZeroStdInputHandle

The `UseNonZeroStdInputHandle` property is a Boolean value that specifies whether to use a non-zero standard input handle.

*Default = True*

## UseRemoteHost

The `UseRemoteHost` property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = False*

## UseTemplateTypename

In the declarations for members of a template class, some compilers require that you use precede the member type with the keyword "typename" for types with dependent names. Since support for this keyword varies between compilers, the `UseTemplateTypename` property is used to specify whether the "typename" keyword should be included in the generated code.

*Default = True*

# Multi4Linux

Environment settings (Compiler, framework libraries, and so on) for Multi4Linux compiler.

## BLDAdditionalOptions

The `BLDAdditionalOptions` property specifies additional compilation switches. The default values are as follows:

Environment Default Value  
`INTEGRITY :optimizestrategy=space :driver_opts=--diag_suppress=14 :driver_opts=--diag_suppress=550 IntegrityESTL :optimizestrategy=space :driver_opts=--diag_suppress=14 :driver_opts=--diag_suppress=550 :cx_mode=extended_embedded :cx_lib=eecx :stdcxxincdirs=$(INTEGRITY_ROOT)\eecxx :stdcxxincdirs=$(INTEGRITY_ROOT)\ansi MultiWin32 :cx_template_option=noimplicit :add_output_ext=checked :cx_e_option=msgnumbers`

:cx\_option=exceptions :check=bounds :check=assignbound :check=nilderef :cx\_template=local  
:cx\_remark=14 :cx\_remark=161 :cx\_remark=837 :cx\_remark=817 :cx\_remark=815 :cx\_remark=47  
:cx\_remark=69 :cx\_remark=830 :cx\_remark=550 :prelink.args=-r :prelink.args=-X7

## BLDMainExecutableOptions

The BLDMainExecutableOptions property specifies the options generated in the main build file of the executable component of the model. The default value for Ada is as follows:

:target\_os=integrity :ada\_library=full :integrity\_option=dynamic :staticlink=true

## BLDMainLibraryOptions

The BLDMainLibraryOptions property specifies the options generated in the main build file of the library component of the model. The default values are as follows:

Environment Default Value INTEGRITY :target\_os=integrity IntegrityESTL MultiWin32 Empty  
MultiLine

## BLDTarget

The BLDTarget property specifies the target BSP. For example, ":target=Win32". This property also affects the names of the framework libraries used in the link. The default value for ADA INTEGRITY metaclass is "sim800."

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration.

To change this property, use the Configuration window in the browser - do not change it by using the Properties tab in the Features window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link. The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

The default value for MultiWin32 is DebugNoExp; for the other environments, the default value is Debug.

## CompileSwitches

The CompileSwitches property specifies the compiler switches. This property replaces the

CPPCompileSwitches property. The default values are as follows:

```
Environment Default Compile Switches Borland -I$OMDefaultSpecificationDirectory
-I$(BCROOT)\INCLUDE;.; "$(OMROOT)\LangCpp";
"$(OMROOT)\LangCpp\oxf"; "$(OMROOT)\LangCpp\omCom";
-D_RTLDLL;_AFXDLL;WIN32;_CONSOLE;_MBCS; WINDOWS;BORLAND;_BOOLEAN
$(INST_FLAGS) $(INCLUDE_PATH) $(INST_INCLUDES) $OMCPPCompileCommandSet -c Linux
-I. -I$OMDefaultSpecificationDirectory -I$(OMROOT) -I$(OMROOT)\LangCpp
-I$(OMROOT)\LangCpp\oxf $(INST_FLAGS) $(INCLUDE_PATH) $(INST_INCLUDES)
-DUSE_Iostream $OMCPPCompileCommandSet -c Microsoft MicrosoftDLL /I . /I
$OMDefaultSpecificationDirectory /I $(OMROOT)\LangCpp /I $(OMROOT)\LangCpp\oxf /nologo /W3
/GX $OMCPPCompileCommandSet /D "_AFXDLL" /D "WIN32" /D "_CONSOLE" /D "_MBCS" /D
"_WINDOWS" $(INST_FLAGS) $(INCLUDE_PATH) $(INST_INCLUDES) /c MicrosoftWinCE /I . /I
$OMDefaultSpecificationDirectory /I $(OMROOT)\LangCpp /I $(OMROOT)\LangCpp\oxf /nologo /W3
/GX- /D _WIN32_WCE=$(CEVersion) /D "$(CEConfigName)" $(MACHINE_CPP_FLAGS) /D
"_OM_NO_Iostream" /D UNDER_CE=$(CEVersion) /D "UNICODE" /D
"_OM_UNICODE_ONLY" $OMCPPCompileCommandSet /D "_AFXDLL" /D "WIN32"
$(INST_FLAGS) $(INCLUDE_PATH) $(INST_INCLUDES) /c MicrosoftWinCE600 /I . /I
$(OMROOT)\LangCpp /I $(OMROOT)\LangCpp\oxf /nologo /W3 /GX- /D
_WIN32_WCE=$(CEVersion) /D "$(CEConfigName)" $(MACHINE_CPP_FLAGS) /D
"_OM_NO_Iostream" /D UNDER_CE=$(CEVersion) /D "UNICODE" /D
"_OM_UNICODE_ONLY" $OMCPPCompileCommandSet /D "_AFXDLL" /D "WIN32"
$(INST_FLAGS) $(INCLUDE_PATH) $(INST_INCLUDES) /D "_X86_" /c MSStandardLibrary /I . /I
$OMDefaultSpecificationDirectory /I $(OMROOT)\LangCpp /I $(OMROOT)\LangCpp\oxf /nologo /W3
/GX $OMCPPCompileCommandSet /D "_AFXDLL" /D "WIN32" /D "_CONSOLE" /D "_MBCS" /D
"_WINDOWS" /D "OM_USE_STL" $(INST_FLAGS) $(INCLUDE_PATH) $(INST_INCLUDES) /c
MultiWin32 ${CPPCompileDebugNoExp} $CPPAdditionalCompileSwitches Nucleus PLUS-PPC -v -c
-DPLUS -DUSE_Iostream -D__DIAB -t$(CPU) -I. -I$OMDefaultSpecificationDirectory
-I$(OMROOT)\LangCpp -I$(OMROOT)\LangCpp\oxf -I$(ATI_DIR) -Xmismatch-warning
-Xno-common $OMCPPCompileCommandSet $(INST_FLAGS) $(INCLUDE_PATH)
$(INST_INCLUDES) OsePPCDiab OseSfk -I. -I$OMDefaultSpecificationDirectory
-I$(OMROOT)\LangCpp $(INST_FLAGS) $(INCLUDE_PATH) $(INST_INCLUDES) PsosPPC
PsosX86 $OMCPPCompileCommandSet QNXNeutrinoCW -I. -I$OMDefaultSpecificationDirectory
-I$(OMROOT) -I$(OMROOT)\LangCpp -I$(OMROOT)\LangCpp\oxf $(INST_FLAGS)
$(INCLUDE_PATH) $(INST_INCLUDES) -DUSE_Iostream $OMCPPCompileCommandSet -c
QNXNeutrinoGCC Solaris2 Solaris2GNU -I. -I$OMDefaultSpecificationDirectory -I$(OMROOT)
-I$(OMROOT)\LangCpp -I$(OMROOT)\LangCpp\oxf $(INST_FLAGS) $(INCLUDE_PATH)
$(INST_INCLUDES) -DUSE_Iostream $OMCPPCompileCommandSet -c VxWorks
-I$OMDefaultSpecificationDirectory -I$(OMROOT) -I$(OMROOT)\LangCpp
-I$(OMROOT)\LangCpp\oxf -DVxWorks $(INST_FLAGS) $(INCLUDE_PATH)
$OMCPPCompileCommandSet -c
```

## DebugLibSuffix

A suffix added to library names. OMLibSuffix keyword is replaced with this property or ReleaseLibSuffix according to the compilation to the build type: Release/Debug.

## DebugSwitches

The DebugSwitches property sets the debug level used in debug switches. The default values are as follows:

Environment Possible Values Default Value INTEGRITY Default, Multi, None, Plain, and Stack Default  
OBJECTADA -ga, -gc, -ga -gc -ga RAVEN\_PPC -ga, -gc, -ga -gc -ga SPARK Empty string

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens has three parameters, each with an integer value:

- TotalNumberOfTokens - The number of tokens in the regular expression
- FileTokenPosition - The position of the file name token in the expression
- LineTokenPosition - The position of the line number token in the expression

The default values are as follows: Environment Default Value Borland PsosX86  
ToTalNumberOfTokens=5,FileTokenPosition=4,LineTokenPosition=5 GNAT  
ToTalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2 JDK Linux QNXNeutrinoCW  
QNXNeutrinoGCC Solaris2GNU SPARK VxWorks Microsoft  
ToTalNumberOfTokens=3,FileTokenPosition=1,LineTokenPosition=2 MicrosoftDLL  
MicrosoftWinCE600 MSStandardLibrary NucleusPLUS-PPC OsePPCDiab OseSfk PsosPPC Solaris2  
INTEGRITY (Ada) ToTalNumberOfTokens=3,FileTokenPosition=1,LineTokenPosition=3 MultiWin32  
(Ada) OBJECTADA RAVEN\_PPC

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the CPP\_CG::<Environment>::ExeName property plus the value of this property.

*(Default = Blank)*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the CPP\_CG::<Environment>::ExeExtension property.

*(Default = Blank)*

## ExtraFilesToCopy

A list of file names (with full paths) separated with commas. The generator copies this list of files to the folder of the makefile only if the file does not already exist.

## HasIDEInterface

The HasIDEInterface property is a Boolean value that specifies whether IDE support is enabled.

If IDE support is enabled (Checked), the IDEInterfaceDLL property points to an IDE adapter that provides connection to the IDE. If the property is set to Cleared, IDE support is disabled and IDE services are not attempted. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody. The default value for QNXNeutrinoCW is False; for the other environments, the default value is True.

## IDEInterfaceDLL

The IDEInterfaceDLL property is a string that points to the IDE adapter DLL. You should not have any reason to modify this property. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody. The default values are as follows:

Environment Default Value QNXNeutrinoCW \$OMROOT/DLLs/CodeWarriorIDE.dll INTEGRITY  
Empty string IntegrityESTL VxWorks \$OMROOT/DLLs/TornadoIDE.dll

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment. The default values are as follows:

*( C++ Default = .cpp; for OsePPCDiab and OseSfk, the default is .cc)*

## IntegrityLinkFile

The name of the Integrity link file that should be added to the makefile template.

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file. The default values are as follows:

Environment Default Value Borland "\$executable" GNAT Microsoft MicrosoftDLL MSStandardLibrary  
MultiWin32 (Ada) NucleusPLUS-PPC OBJECTADA RAVEN\_PPC SPARK INTEGRITY Empty string  
IntegrityESTL MicrosoftWinCE600 JDK "\$OMROOT/etc/Executer.exe" "\"\$OMROOT/etc\jdkrun.bat\"  
\$makefile Main\$ComponentName" Linux \$executable MultiWin32 (C++) QNXNeutrinoCW  
QNXNeutrinoGCC MicrosoftWinCE "\$OMROOT/etc\msceRun.bat" \$executable IX86EM OsePPCDiab



"\$OMROOT/etc/osesfkRun.bat" \$executable OseSfk Solaris2 xterm -e \$executable Solaris2GNU

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

The InvokeMake default values are as follows:

```
Environment Default Value Borland $OMROOT/etc/Executer.exe "\"$OMROOT\etc\bc5make.bat\"  
$makefile $maketarget" GNAT "$makefile" $maketarget INTEGRITY ESTL  
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\IntegrityMake.bat\" $makefile $maketarget" Integrity  
JDK "$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\jdkmake.bat\" $makefile $maketarget" Linux  
$OMROOT/etc/linuxmake $makefile $maketarget Microsoft "$OMROOT/etc/Executer.exe"  
"$OMROOT\etc\msmake.bat\" $makefile $maketarget" MicrosoftDLL MSStandardLibrary  
MicrosoftWinCE "$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\mscemake.bat\" $makefile  
$maketarget IX86EM" MicrosoftWinCE600 "$OMROOT/etc/Executer.exe"  
"$OMROOT\etc\msceNETmake.bat\" $makefile $maketarget x86" MultiWin32 (Ada)  
"$OMROOT/etc/Executer.exe" "$OMROOT\etc\AdaMultiWin32Make.bat $makefile $maketarget"  
OBJECTADA "$OMROOT/etc/Executer.exe" "$OMROOT\etc\ObjectAdaMake.bat $makefile  
$maketarget" OsePPCDiab "$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\oseppcdiabmake.bat\"  
$makefile $maketarget" OseSfk "$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\osesfkmake.bat\"  
$makefile $maketarget" PsosPPC "$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\psppcmake.bat\"  
$makefile $maketarget" PsosX86 "$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\psx86make.bat\"  
$makefile $maketarget" QNXNeutrinoCW "$OMROOT/etc/Executer.exe"  
"$OMROOT\etc\qnxcwmake.bat\" $makefile $maketarget" QNXNeutrinoGCC Empty string  
RAVEN_PPC "$OMROOT/etc/Executer.exe" "$OMROOT\etc\ObjectAdaRavenPPCMake.bat $makefile  
$maketarget" Solaris2 $OMROOT/etc/sol2make $makefile $maketarget Solaris2GNU SPARK  
"$OMROOT/etc/Executer.exe" "$OMROOT\etc\SPARKMake.bat $makefile $maketarget" VxWorks  
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vxmake.bat\" $makefile $maketarget"
```

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format.

If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.



*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment. The default values are as follows:

Environment Default Value Borland .lib Microsoft MicrosoftWinCE600 MSStandardLibrary MultiWin32 NucleusPLUS-PPC OBJECTADA OseSfk PsosX86 GNAT .a INTEGRITY IntegrityESTL Linux OsePPCDiab PsosPPC QNXNeutrinoCW QNXNeutrinoGCC RAVEN\_PPC Solaris2 Solaris2GNU VxWorks JDK Empty string SPARK MicrosoftDLL .dll

## LibPrefix

Combines all the prefixes of library names.

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode. The default values are as follows:

Environment Default Value Borland \$OMLinkCommandSet Linux MultiWin32 (C++) NucleusPLUS-PPC OsePPCDiab PsosPPC PsosX86 QNXNeutrinoGCC Solaris2 Solaris2GNU VxWorks INTEGRITY --one\_instantiation\_per\_object \$OMLinkCommandSet -cpu=\$(TARGET\_CPU) -map IntegrityESTL Microsoft \$OMLinkCommandSet /NOLOGO MicrosoftDLL MicrosoftWinCE600 MSStandardLibrary OseSfk -nologo \$OMLinkCommandSet QNXNeutrinoCW -static

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the CPP\_CG::::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*(Default = .gpj)*

## MakeFileContentForExe1

The content of the makefiles, in case of executable component type.

## MakeFileContentForExe2

The content of the makefiles, in case of executable component type.

## **MakeFileContentForLib1**

The content of the makefiles, in case of library component type.

## **MakeFileContentForLib2**

The content of the makefiles, in case of library component type.

## **MakeFileName**

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the `CPP_CG::<Environment>::MakeExtension` property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## **MakeFileNameForExe1**

The name of the makefiles, in case of executable component type.

## **MakeFileNameForExe2**

The name of the makefiles, in case of executable component type.

## **MakeFileNameForLib1**

The name of the makefiles, in case of library component type.

## **MakeFileNameForLib2**

The name of the makefiles, in case of library component type.

## **MultipleAddressSpacesIntFileContent**

The content of the MultipleAddressSpacesIntFileName file.

## MultipleAddressSpacesIntFileName

a file with this name is created in case of multiple address space compilation.

## MultipleAddressSpacesLibraries

names of libraries to add in case of multiple address space usage.

## MultipleAddressSpacesPrefix

A prefix that is added to libraries in case of multiple address space compilation. OMMultipleAddressSpacesPrefix keyword adds this prefix when needed.

## MultipleAddressSpacesSwitches

A switch for multiple address space compilation. The makefile template can add it directly but it is preferred to use the keyword OMMultipleAddressSpacesSwitches – that checks whether this switch should be added.

## NetAndSocketLibs

A list of library names that is added to OMWebLibs keyword if web-enabling flag is on or to OMInstrumentationFlags keyword if the instrumentation is in animation mode.

## OSFileSystemCaseSensitive

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive. The default values are as follows:

Environment	Default	Value
Borland	Cleared	GNAT
Integrity	ESTL	JDK
Microsoft	MicrosoftDLL	MicrosoftWinCE600
MSStandardLibrary	MultiWin32	NucleusPLUS-PPC
OseSfk	OsePPCDiab	PsosPPC
PsosX86	RAVEN_PPC	SPARK
VxWorks	Linux	Checked
QNXNeutrinoCW	QNXNeutrinoGCC	Solaris2
Solaris2GNU		

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. The default values are as follows:

Environment	Default	Value
Borland	PsosX86	
ToTalNumberOfTokens=5,	FileTokenPosition=4,	LineTokenPosition=5
GNAT		

ToTalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2 JDK Linux QNXNeutrinoCW  
QNXNeutrinoGCC Solaris2GNU SPARK VxWorks IntegrityESTL  
ToTalNumberOfTokens=3,FileTokenPosition=1,LineTokenPosition=2 Microsoft MicrosoftDLL  
MicrosoftWinCE600 MSStandardLibrary NucleusPLUS-PPC OsePPCDiab OseSfk PsosPPC Solaris2  
INTEGRITY (Ada) ToTalNumberOfTokens=3,FileTokenPosition=1,LineTokenPosition=3 MultiWin32  
(Ada) OBJECTADA RAVEN\_PPC

## PosixSharedMemoryFiles

This list is copied only in case of multiple address space compilation.

## ReleaseLibSuffix

A suffix added to library names. OMLibSuffix keyword is replaced with this property or DebugLibSuffix according to the compilation to the build type: Release/Debug.

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely.

In such configurations, the host is the machine running Rational Rhapsody, whereas the target is the machine running the application.

To run remotely, the UseRemoteHost property must be set to Checked. If UseRemoteHost is Checked and RemoteHost is blank, the current host name is used for the remote host. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = Empty string*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment. The extension ".ads" is the default for Ada.

## UseNonZeroStdInputHandle

The UseNonZeroStdInputHandle property is a Boolean value that specifies whether to use a non-zero standard input handle.

For INTEGRITY, OBJECTADA, RAVEN\_PPC, and SPARK the default value is Cleared; for the other environments, the default value is Checked.

## UseRemoteHost

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

The default value for the following environments is Cleared:

Borland GNAT INTEGRITY IntegrityESTL Microsoft MicrosoftDLL MSStandardLibrary MultiWin32

The default value for the following environments is Checked:

Linux MicrosoftWinCE600 NucleusPLUS-PPC OsePPCDiab OseSfk PsosPPC PsosX86  
QNXNeutrinoCW QNXNeutrinoGCC Solaris2 Solaris2GNU VxWorks

## WebInstLibs

A list of library names that is added to OMWebLibs keyword if web-enabling flag is on.

## Multi4Win32

The Multi4Win32 metaclass contains environment properties (Compiler, framework libraries, and so on) for GHS MULTI 4.0.X Win32 compiler.

## AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path. Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files. To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMRoot)\LangCpp\osconfig\MultiWin32*

## AnimInstLibs

The AnimInstLibs property is used to specify the static libraries that are required when Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with INST\_LIBS.

*Default = -l\$(LibPrefix)OxfInst\$(BLDTarget)\$OMLibSuffix\$LibExtension  
-l\$(LibPrefix)AomAnim\$(BLDTarget)\$OMLibSuffix\$LibExtension  
-l\$(FrameworkLibPrefix)OmComAppl\$(BLDTarget)\$OMLibSuffix\$LibExtension*

## AnimPreprocessor

The AnimPreprocessor property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with INST\_FLAGS.

*Default = -D\_OMINSTRUMENT*

## BLDAdditionalDefines

The BLDAdditionalDefines property specifies additional compiler preprocessor flags.

*Default = Empty string*

## BLDAdditionalOptions

The BLDAdditionalOptions property specifies additional compilation switches.

*Default =*

`-threading=multiple -I. --exceptions --no_implicit_include --display_error_number --diag_remark 14,161,837,817,815,47,69,830,550 -tlocal -prelink.args=-r -prelink.args=-X7`

## BLDIncludeAdditionalBLD

The IncludeAdditional property specifies additional build options.

*Default = Empty MultiLine*

## BLDTarget

The BLDTarget property specifies the target BSP. For example, ":target=Win32". This property also affects the names of the framework libraries used in the link.

*Default = Empty String*

## BuildArgumentsInIDE

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that are used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the CPP\_CG::[environment]::BuildCommandInIDE property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandInIDE

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration.

To change this property, use the Configuration window in the browser - do not change it by using the Properties tab in the Features window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link. The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*Default = DebugNoExp*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT/etc\MultiWin32Make.bat\" MultiWin32Build.bat buildLibs "*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## COM

The COM property specifies whether the current component is a COM component. By default, this property is set to Checked for all COM components (stereotypes COM DLL, COM EXE, and COM TLB). If you set this property in the generated makefile for the component, the linker option /SUBSYSTEM is set to :windows.

*Default = Cleared*

## CompileDebug

The CompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*Default = Empty string*

## CompileSwitches

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property.

*Default =*

`-DUSE_Iostream -DHAS_NO_EXP`

## CPPAdditionalReservedWords

The CPPAdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use.

In general, this is an environment property that can be contained in any of the environment metaclasses supported by the software. The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

*Default =*

`__asm __finally naked __based __inline __single_inheritance __cdecl __int8 __stdcall __declspec __int16  
dllexport __int32 __try dllimport __int64 __virtual_inheritance __except __leave __fastcall`



`__multiple_inheritance`

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*Default = -D\_DEBUG -G*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*Default = Empty MultiLine*

## DebugLibSuffix

A suffix added to library names. OMLibSuffix keyword is replaced with this property or ReleaseLibSuffix according to the compilation to the build type: Release/Debug.

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction with an IDE such as Eclipse, the EnableDebugIntegrationWithIDE property can be used to specify whether or not the IDE debugger should be used in conjunction with the software animation feature.

If the value of the property is set to True, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's exit() function is called automatically. The system function calls the destructors of framework instances in the order it determines. The EndApplicationCode property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an endApplication() function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to endApplication(), it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The EntryPoint property specifies the name of the main program for a given environment.

*Default = main*

See also the definition of the EntryPointDeclarationModifier property for more information.

## EnvironmentVarName

The EnvironmentVarName property specifies the name of the global variable that you must define in order to use the Embedded C++ compiler. It is used by the MultiMakefileGenerator. The value replaces the \$EnvironmentVarName value keyword inside the value for the BLDAdditionalOptions property.

*Default = MULTI\_ROOT*

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens has three parameters, each with an integer value:

- TotalNumberOfTokens - The number of tokens in the regular expression
- FileTokenPosition - The position of the file name token in the expression
- LineTokenPosition - The position of the line number token in the expression

*Default = ToTalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2*

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the CPP\_CG::<Environment>::ExeName property plus the value of this property.

*(Default = .exe)*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the `CPP_CG::<Environment>::ExeExtension` property.

*(Default = Blank)*

## ExtraFilesToCopy

A list of file names (with full paths) separated with commas. The generator copies this list of files to the folder of the makefile only if the file does not already exist.

*Default = Empty string*

## FrameworkLibPrefix

The `FrameworkLibPrefix` property specifies the prefix of the Rational Rhapsody framework library linked with your application.

*Default = Multi4Win32*

## GetConnectedRuntimeLibraries

The `GetConnectedRuntimeLibraries` property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile. Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

*Default =*

`$(LibPrefix)WebComponents$(BLDTarget)$OMLibSuffix$LibExtension ,  
$OMRoot/lib/$(FrameworkLibPrefix)WebServices$(BLDTarget)$OMLibSuffix$LibExtension`

## ImpExtension

The `ImpExtension` property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment.

*Default = .cpp*

## InvokeExecutable

The `InvokeExecutable` property specifies the command used to run an executable file.

*Default = "\$executable"*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

```
Default = "$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\Multi4Win32Make.bat\" $makefile $maketarget"
```

## InvokeMakeGenerator

The InvokeMakeGenerator specifies the path to the executable program for an external makefile generator. This external generator is started each time you request a makefile generation. If the specified path is incorrect, Rational Rhapsody generates an error message. If you are using a full-featured external code generator, this property setting is ignored.

*Default = \$OMROOT/etc/MultiMakefileGenerator.exe*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format.

If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .lib*

## LibPrefix

Combines all the prefixes of library names.

Default = \$(FrameworkLibPrefix)

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

*Default = Empty string (blank)*

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

Default value = Empty string

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

*Default = Empty string*

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the CPP\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*(Default = .gpj)*

## MakeFileContentForExe1

The content of the makefiles, in case of executable component type.

*Default =*

```
#!/gbuild primaryTarget=$PrimaryTarget [Project] -object_dir=$ObjectsDirectory  
$MakeFileNameForExe2
```

## MakeFileContentForExe2

The content of the makefiles, in case of executable component type.

*Default =*

```
#!/gbuild [Program] -o $OMTargetName$ExeExtension -object_dir=$ObjectsDirectory
$BLDAdditionalOptions $BLDIncludeAdditionalBLD -I$(OMRoot)/LangCpp
-L$(OMRoot)/LangCpp/lib $OMUserIncludePath $LinkSwitches $OMCompilationFlag
$CompileSwitches $OMReusableFlag $OMInstrumentationFlags $OMInstrumentationLibs
$BLDAdditionalDefines $OMUserLibs $OMMainFiles$ImpExtension $OMSrcFiles
```

## MakeFileContentForLib1

The content of the makefiles, in case of library component type.

The default is as follows:

```
#!/gbuild primaryTarget=$PrimaryTarget [Project] -object_dir=$ObjectsDirectory
$MakeFileNameForLib2
```

## MakeFileContentForLib2

The content of the makefiles, in case of library component type.

The default is as follows:

```
#!/gbuild [Library] -o $OMTargetName$LibExtension -object_dir=$ObjectsDirectory
$BLDAdditionalOptions $BLDIncludeAdditionalBLD -I$(OMRoot)/LangCpp $OMUserIncludePath
$OMCompilationFlag $CompileSwitches $OMInstrumentationFlags $OMReusableFlag
$BLDAdditionalDefines $OMSrcFiles
```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the CPP\_CG::<Environment>::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## MakeFileNameForExe1

The name of the makefiles, in case of executable component type.

*Default = \$(OMTargetName)\$MakeExtension*

## **MakeFileNameForExe2**

The name of the makefiles, in case of executable component type.

*Default = \$(OMTargetName)\_program\$MakeExtension*

## **MakeFileNameForLib1**

The name of the makefiles, in case of library component type.

*Default = \$(OMTargetName)\$MakeExtension*

## **MakeFileNameForLib2**

The name of the makefiles, in case of library component type.

*Default = \$(OMTargetName)\_library\$MakeExtension*

## **NetAndSocketLibs**

A list of library names that is added to OMWebLibs keyword if web-enabling flag is on or to OMInstrumentationFlags keyword if the instrumentation is in animation mode.

*Default = -lwsock32.lib*

## **NoneInstLibs**

The NoneInstLibs property is used to specify the static libraries that are required when Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with INST\_LIBS.

*Default = -l\$(LibPrefix)Oxf\$(BLDTarget)\$OMLibSuffix\$LibExtension*

## **NonePreprocessor**

The NonePreprocessor property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with INST\_FLAGS.

*Default = Blank*

## **NullValue**

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## **ObjCleanCommand**

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default = Empty string*

## **ObjectName**

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = Empty string*

## **ObjectsDirectory**

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = work*

## **ObjExtension**

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*Default = .obj*

## **OSFileSystemCaseSensitive**

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Cleared*

## **ParseErrorDescript**



The ParseErrorDescrip property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = ([^"]+)", line ([0-9]+)[: ] (error/warning) (.\*)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = ([^"]+)", line ([0-9]+)*

## ParseMakeError

The ParseMakeError property is used to define a regular expression that represents the format of make process or linker error messages. This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = (Error)[: ] (build failed)*

## ParseSeverityError

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^"]+)", line ([0-9]+)[: ] (error)*

## ParseSeverityWarning

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^"]+)", line ([0-9]+)[: ] (warning)*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation.

*Default = /*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Checked*

## RCCompileCommand

The RCCompileCommand property is a string that specifies the compilation command for the resource file. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = Empty string*

## RCExtension

The RCExtension property is a string that specifies the extension for resource files. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = .rc*

## ReleaseLibSuffix

A suffix added to library names. OMLibSuffix keyword is replaced with this property or DebugLibSuffix according to the compilation to the build type: Release/Debug.

*Default = R*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely.

In such configurations, the host is the machine running Rational Rhapsody, whereas the target is the machine running the application.

To run remotely, the UseRemoteHost property must be set to Checked. If UseRemoteHost is Checked and RemoteHost is blank, the current host name is used for the remote host. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = Empty string*

## ReusableStatechartSwitches

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts. See the upgrade history on the support site for information on this change.

*Default = OM\_REUSABLE\_STATECHART\_IMPLEMENTATIO*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*Default = .h*

## SubSystem

The SubSystem property is a string that defines the type of the program for the Microsoft linker. The possible values are as follows:

- **CONSOLE** - Used for a Win32 character-mode application
- **WINDOWS** - Used for an application that does not require a console
- **NATIVE** - Applies device drivers for Windows NT
- **POSIX** - Creates an application that runs with the POSIX subsystem in Windows NT

*Default = /SUBSYSTEM:console*

## TraceInstLibs

The TraceInstLibs property is used to specify the static libraries that are required when Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with INST\_LIBS.

*Default = -l\$(LibPrefix)OxfInst\$(BLDTarget)\$OMLibSuffix\$LibExtension  
-l\$(LibPrefix)AomTrace\$(BLDTarget)\$OMLibSuffix\$LibExtension  
-l\$(LibPrefix)TomTrace\$(BLDTarget)\$OMLibSuffix\$LibExtension  
-l\$(FrameworkLibPrefix)OmComAppl\$(BLDTarget)\$OMLibSuffix\$LibExtension*

## TracePreprocessor

The TracePreprocessor property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with INST\_FLAGS.

*Default = -DOMTRACER*

## UpdateBuildSettingsInIDE

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build

settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## UseActorsCode

The UseActorsCode property specifies whether code is generated for actors. The value of the property should be synchronized with the configuration Generate Code For Actors check box (located in the configuration Initialization tab).

*Default = Cleared*

## UseNewBuildOutputWindow

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the CG::General::ShowLogViewAfterBuild property to True.

*Default = Checked*

## UseTemplateName

In the declarations for members of a template class, some compilers require that you use precede the member type with the keyword "typename" for types with dependent names. Since support for this keyword varies between compilers, the UseTemplateName property is used to specify whether the "typename" keyword should be included in the generated code.

*Default = Cleared*

# MultiWin32

The MultiWin32 metaclass contains the environment settings (Compiler, framework libraries, and so on) for GHS MULTI 3.5 Win32 compiler.

## AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path. Previously, the RTOS-specific code was removed from the

framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files. To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the `AdaptorSearchPath` property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)\LangCpp\osconfig\MultiWin32*

## **BLDAdditionalDefines**

The `BLDAdditionalDefines` property specifies additional compiler preprocessor flags.

*Default = Empty string*

## **BLDAdditionalOptions**

The `BLDAdditionalOptions` property specifies additional compilation switches.

*Default =*

`:cx_template_option=noimplicit :add_output_ext=true :cx_e_option=msgnumbers :cx_option=exceptions :check=bounds :check=assignbound :check=nilderef :cx_template=local :cx_remark=14 :cx_remark=161 :cx_remark=837 :cx_remark=817 :cx_remark=815 :cx_remark=47 :cx_remark=69 :cx_remark=830 :cx_remark=550 :prelink.args=-r :prelink.args=-X7 :defines=OM_STL`

## **BLDIncludeAdditionalBLD**

The `IncludeAdditional` property specifies additional build options.

*Default = Empty MultiLine*

## **BLDTarget**

The `BLDTarget` property specifies the target BSP. For example, `":target=Win32"`. This property also affects the names of the framework libraries used in the link.

*Default = Empty string*

## **BuildArgumentsInIDE**

The `BuildArgumentsInIDE` property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the `CPP_CG::[environment]::BuildCommandInIDE` property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the `BuildInIDE` property is set to `True`.

*Default = Blank*

## **BuildCommandInIDE**

The `BuildCommandInIDE` property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the `BuildInIDE` property is set to `True`.

*Default = Blank*

## **BuildCommandSet**

The `BuildCommandSet` property generates a set of commands in the makefile to build a debug or release version of the configuration.

To change this property, use the Configuration window in the browser - do not change it by using the Properties tab in the Features window or by modifying the `site.prp` file. Note that this property also affects the names of the framework libraries used in the link. The possible values are as follows:

- `Debug` - Generate the debug command that is set in the makefile.
- `DebugNoExp` - Generate the debug command that is set in the makefile without the exceptions flag (`:cx_option=exceptions`).
- `Release` - Generate the release command that is set in the makefile.
- `ReleaseNoExp` - Generate the release command that is set in the makefile without the exceptions flag (`:cx_option=exceptions`).

*Default = DebugNoExp*

## **buildFrameworkCommand**

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The `buildFrameworkCommand` property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT/etc\MultiWin32Make.bat\" MultiWin32Build.bat buildLibs bld "*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## COM

The COM property specifies whether the current component is a COM component. By default, this property is set to Checked for all COM components (stereotypes COM DLL, COM EXE, and COM TLB). If you set this property in the generated makefile for the component, the linker option /SUBSYSTEM is set to :windows.

*Default = Cleared*

## CPPAdditionalReservedWords

The CPPAdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use.

In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody. The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

*Default =*

`__asm __finally naked __based __inline __single_inheritance __cdecl __int8 __stdcall __declspec __int16  
dllexport __int32 __try dllimport __int64 __virtual_inheritance __except __leave __fastcall  
__multiple_inheritance`

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction with an IDE such as Eclipse, the EnableDebugIntegrationWithIDE property can be used to specify whether or not the IDE debugger should be used in conjunction with the software animation feature.

If the value of the property is set to True, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's `exit()` function is called automatically. The system function calls the destructors of framework instances in the order it determines. The `EndApplicationCode` property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an `endApplication()` function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to `endApplication()`, it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The `EntryPoint` property specifies the name of the main program for a given environment.

*Default = main*

See also the definition of the `EntryPointDeclarationModifier` property for more information.

## ErrorMessageTokensFormat

The `ErrorMessageTokensFormat`, working with the `ParseErrorMessage` property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. `ErrorMessageTokensFormat` defines the number and location of tokens within the regular expression defined by the `ParseErrorMessage` property. `ErrorMessageTokens` has three parameters, each with an integer value:

- `TotalNumberOfTokens` - The number of tokens in the regular expression
- `FileTokenPosition` - The position of the file name token in the expression
- `LineTokenPosition` - The position of the line number token in the expression

*Default = ToTalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2*

## ExeExtension

The `ExeExtension` property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the `CPP_CG::<Environment>::ExeName` property plus the value of this property.

*(Default = .exe)*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active



component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the CPP\_CG::<Environment>::ExeExtension property.

*(Default = Blank)*

## FrameworkLibPrefix

The FrameworkLibPrefix property specifies the prefix of the Rational Rhapsody framework library linked with your application.

*Default = MultiWin32*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment.

*Default = .cpp*

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*Default = \$executable*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\MultiWin32Make.bat\" \$makefile*

*\$maketarget"*

## InvokeMakeGenerator

The InvokeMakeGenerator specifies the path to the executable program for an external makefile generator. This external generator is started each time you request a makefile generation. If the specified path is incorrect, Rational Rhapsody generates an error message. If you are using a full-featured external code generator, this property setting is ignored.

*Default = \$OMROOT/etc/MultiMakefileGenerator*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format.

If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .lib*

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the CPP\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*(Default = .bld)*

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration.

*Default = Empty MultiLine*

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the CPP\_CG::<Environment>::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## ObjCleanCommand

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default = Empty string*

## ObjectName

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = Empty string*

## **ObjectsDirectory**

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = obj\_dir*

## **ObjExtension**

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*Default = .obj*

## **OSFileSystemCaseSensitive**

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Cleared*

## **ParseErrorDescript**

The ParseErrorDescript property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = ([^"]+)", line ([0-9]+)[: ] (error/warning) (.\*)*

## **ParseErrorMessage**

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning). Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. The default values are as follows:

*Default = ([^"]+)", line ([0-9]+)*

## **ParseMakeError**

The ParseMakeError property is used to define a regular expression that represents the format of make process or linker error messages. This property is used to determine the type of icon that should be

displayed alongside the message on the Build tab of the Output window.

*Default = (Error)[:] (build failed)*

## **ParseSeverityError**

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^"]+)", line ([0-9]+)[:] (error)*

## **ParseSeverityWarning**

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^"]+)", line ([0-9]+)[:] (warning)*

## **PathDelimiter**

The PathDelimiter property specifies an alternative path separator for code generation.

*Default = /*

## **QuoteOMROOT**

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Cleared*

## **RCCompileCommand**

The RCCompileCommand property is a string that specifies the compilation command for the resource file. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = Empty string*

## **RCExtension**

The RCExtension property is a string that specifies the extension for resource files. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational

Rhapsody.

*Default = .rc*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely.

In such configurations, the host is the machine running Rational Rhapsody, whereas the target is the machine running the application.

To run remotely, the UseRemoteHost property must be set to Checked. If UseRemoteHost is Checked and RemoteHost is blank, the current host name is used for the remote host. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = Empty string*

## ReusableStatechartSwitches

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts. See the upgrade history on the support site for information on this change.

*Default = OM\_REUSABLE\_STATECHART\_IMPLEMENTATION.*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*Default = .h*

## SubSystem

The SubSystem property is a string that defines the type of the program for the Microsoft linker. The possible values are as follows:

- CONSOLE - Used for a Win32 character-mode application
- WINDOWS - Used for an application that does not require a console
- NATIVE - Applies device drivers for Windows NT
- POSIX - Creates an application that runs with the POSIX subsystem in Windows NT

*Default = /SUBSYSTEM:console*

## UpdateBuildSettingsInIDE

The `UpdateBuildSettingsInIDE` property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to `True`, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## UseNewBuildOutputWindow

The `UseNewBuildOutputWindow` property determines which tab is brought to the front of the Output window after the completion of a build action. If set to `True`, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the `CG::General::ShowLogViewAfterBuild` property to `True`.

*Default = Checked*

## UseRemoteHost

The `UseRemoteHost` property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Cleared*

## UseTemplateTypename

In the declarations for members of a template class, some compilers require that you use precede the member type with the keyword `"typename"` for types with dependent names. Since support for this keyword varies between compilers, the property `UseTemplateTypename` is used to specify whether the `"typename"` keyword should be included in the generated code.

*Default = Cleared*

## NucleusPLUS-PPC

The `NucleusPLUS-PPC` metaclass contains environment settings (Compiler, framework libraries, and so on) for NucleusPLUS RTOS compiled for PPC, by using Diab compiler.

## AdaptorSearchPath

The `AdaptorSearchPath` property specifies the path to the operating system configuration file. This path is

added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the `AdaptorSearchPath` property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)/LangCpp/osconfig/Nucleus*

## BuildArgumentsInIDE

The `BuildArgumentsInIDE` property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the `CPP_CG::[environment]::BuildCommandInIDE` property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the `BuildInIDE` property is set to `True`.

*Default = Blank*

## BuildCommandInIDE

The `BuildCommandInIDE` property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the `BuildInIDE` property is set to `True`.

*Default = Blank*

## BuildCommandSet

The `BuildCommandSet` property generates a set of commands in the makefile to build a debug or release version of the configuration.

To change this property, use the Configuration window in the browser - do not change it by using the Properties tab in the Features window or by modifying the `site.prp` file. Note that this property also affects



the names of the framework libraries used in the link. The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*Default = Debug*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\$OMROOT/etc/Executer.exe" "\""\$OMROOT\etc\numake.bat" nubuild.mak buildLibs  
\"CPU=\$CPU\" \"BUILD\_SET=\$BuildCommandSet\" \" \"*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then the software calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## CompileSwitches

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property.

*Default =*

*-v -c -DPLUS -DUSE\_IOSTREAM -D\_\_DIAB -t\$(CPU) -I. -I\$OMDefaultSpecificationDirectory  
-I\$(OMROOT)\LangCpp -I\$(OMROOT)\LangCpp\oxf -I\$(ATI\_DIR) -Xmismatch-warning  
-Xno-common \$OMCPPCompileCommandSet \$(INST\_FLAGS) \$(INCLUDE\_PATH)  
\$(INST\_INCLUDES)*

## CPPAdditionalReservedWords

The CPPAdditionalReservedWords property is a string that specifies additional reserved keywords that

Rational Rhapsody does not allow you to use.

In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody. The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

*Default =*

`__asm __finally naked __based __inline __single_inheritance __cdecl __int8 __stdcall __declspec __int16  
dllexport __int32 __try dllimport __int64 __virtual_inheritance __except __leave __fastcall  
__multiple_inheritance`

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The default is as follows:

`$(CPP) $OMFileCPPCompileSwitches -o $OMFileObjPath $OMFileImpPath`

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*Default = -g -D\_DEBUG -DASSERT\_DEBUG*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*Default = -DNDEBUG*

## CPU

The CPU property represents the target CPU for projects using the NucleusPLUS-PPC environment. The value of this property is referred to in the value of the buildFrameworkCommand property, which defines the command that is run to build the framework for the environment when you select Code > Build Framework from the main Rational Rhapsody menu.

*Default = PPC860ES*

## DependencyRule

The `DependencyRule` property specifies how file dependencies for a configuration are generated in the makefile.

The default is as follows:

```
$OMFileObjPath : $OMFileImpPath $OMFileSpecPath $(CREATE_OBJ_DIR) $OMFileDependencies
```

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: `$OMFileObjPath : $OMFileImpPath "*.bmp" "*.ico" "*.rc2"`

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction with an IDE such as Eclipse, the `EnableDebugIntegrationWithIDE` property can be used to specify whether or not the IDE debugger should be used in conjunction with the software animation feature.

If the value of the property is set to `True`, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's `exit()` function is called automatically. The system function calls the destructors of framework instances in the order it determines. The `EndApplicationCode` property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an `endApplication()` function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to `endApplication()`, it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The `EntryPoint` property specifies the name of the main program for a given environment.

`Default = numain`

See also the definition of the `EntryPointDeclarationModifier` property for more information.

## ErrorMessageTokensFormat

The `ErrorMessageTokensFormat`, working with the `ParseErrorMessage` property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line

number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens has three parameters, each with an integer value:

- TotalNumberOfTokens - The number of tokens in the regular expression
- FileTokenPosition - The position of the file name token in the expression
- LineTokenPosition - The position of the line number token in the expression

*Default = TotalNumberOfTokens=3,FileTokenPosition=1,LineTokenPosition=2*

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the CPP\_CG::<Environment>::ExeName property plus the value of this property.

*(Default = .elf)*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the CPP\_CG::<Environment>::ExeExtension property.

*(Default = Blank)*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements. The file inclusions are generated in the makefile.

*Default = \$OMSpecIncludeInElements \$OMImpIncludeInElements*

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile. Note that if you select Release Build Set (in the

Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

The default is as follows:

```
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)WebComponents$(LIB_POSTFIX)$(LIB_EXT),  
$(OMROOT)\lib\$(LIB_PREFIX)WebServices$(LIB_POSTFIX)$(LIB_EXT)
```

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment. The default values are as follows:

*Default* = .cpp

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*Default* = .INCLUDE:

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*Default* = "\$executable"

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

*Default* = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\numake.bat\" \$makefile \$maketarget"

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## **LibExtension**

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .lib*

## **LinkDebug**

The LinkDebug property specifies the special link switches used to link in debug mode.

*Default = Empty string*

## **LinkRelease**

The LinkRelease property specifies the special link switches used to link in release mode.

*Default = Empty string*

## **LinkSwitches**

The LinkSwitches property specifies the standard link switches used to link in any mode.

*Default = \$OMLinkCommandSet*

## **MakeExtension**

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the CPP\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*(Default = .mak)*

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration.

```
Default = ##### Custom User Settings #####
#####.IMPORT .IGNORE :
ATI_DIR CPU=PPC860ES LIBS=$(ATI_DIR)\PLUS\O\PLUS.LIB -ld
DLDFILE=$(OMROOT)\MakeTpl\nuos.dld NU_SOCK_LIB=$(ATI_DIR)\lib\net.lib
$(ATI_DIR)\lib\pquicc.lib ##### Target type (Debug/Release) #####
#####
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease BuildSet=$OMBuildSet
ConfigurationCPPCompileSwitches=$OMReusableStatechartSwitches
$OMConfigurationCPPCompileSwitches ##### Compilation flags
#####
INCLUDE_QUALIFIER=-I LIB_PREFIX=Nu ##### Commands definition
#####
##### CPP=dcc.exe
LIB_CMD=dar.exe LINK_CMD=dld.exe LIB_FLAGS=$OMConfigurationLinkSwitches
LINK_FLAGS=$OMConfigurationLinkSwitches CP=cp RM=rm ##### Generated macros
##### $OMContextMacros
##### Predefined macros #####
#####
OBJ_DIR=$OMObjectsDir .IF "$(OBJ_DIR)"!=" create_obj_dir: @[ @echo off @if not exist
$(OBJ_DIR) mkdir $(OBJ_DIR) ] CREATE_OBJ_DIR= create_obj_dir CLEAN_OBJ_DIR=if exist
$(OBJ_DIR) rmdir $(OBJ_DIR) .ELIF CREATE_OBJ_DIR= CLEAN_OBJ_DIR= .ENDIF $(OBS) :
$(FLAGSFILE) $(RULESFILE) $(INST_LIBS) $(OXF_LIBS) LIB_POSTFIX= .IF
"$(BuildSet)"=="Release" LIB_POSTFIX=R .ENDIF .IF "$(INSTRUMENTATION)" == "Animation"
INST_FLAGS=-DOMANIMATOR INST_INCLUDES=-I$(OMROOT)\LangCpp\om
-I$(OMROOT)\LangCpp\tom INST_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomanim$(LIB_POSTFIX)$(LIB_EXT)
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfinst$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX)$(LIB_EXT)
SOCK_LIB=$(NU_SOCK_LIB) .ELIF "$(INSTRUMENTATION)" == "Tracing"
INST_FLAGS=-DOMTRACER INST_INCLUDES=-I$(OMROOT)\LangCpp\om
-I$(OMROOT)\LangCpp\tom
```

```

INST_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)tomtrace$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomtrace$(LIB_POSTFIX)$(LIB_EXT) OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfstt$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX)$(LIB_EXT)
SOCK_LIB=$(NU_SOCK_LIB) .ELIF "$(INSTRUMENTATION)" == "None" INST_FLAGS=
INST_INCLUDES= INST_LIBS=
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$(LIB_POSTFIX)$(LIB_EXT)
SOCK_LIB=$(NU_SOCK_LIB) .ELSE msg2: @[ @echo An invalid Instrumentation
$(INSTRUMENTATION) is specified. @error ] .ENDIF NU_ADAPTOR_OBJ=NuAppInit$(OBJ_EXT)
$(NU_ADAPTOR_OBJ) : NuAppInit$(CPP_EXT) $(CPP) $(ConfigurationCPPCompileSwitches)
NuAppInit$(CPP_EXT) NuAppInit$(CPP_EXT) : $(OMROOT)/MakeTpl/NuAppInit$(CPP_EXT) $(CP)
"$<" $@ ##### Generated dependencies #####
#####
$OMContextDependencies $OMFileObjPath : $OMMainImplementationFile $(OBJS)
$(NU_ADAPTOR_OBJ) $(FLAGSFILE) $(RULESFILE) $(CPP) $(ConfigurationCPPCompileSwitches)
-o $OMFileObjPath $OMMainImplementationFile ##### Linking instructions
#####
#####
LNK_OPTIONS_FILE=linker.opt $(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS)
$(NU_ADAPTOR_OBJ) $OMFileObjPath $OMModelLibs @echo Linking
$(TARGET_NAME)$(EXE_EXT) echo $(LINK_FLAGS) -t$(CPU):simple -o
$(TARGET_NAME)$(EXE_EXT) > $(LNK_OPTIONS_FILE) for %F in (*.o) do @echo %F >>
$(LNK_OPTIONS_FILE) echo $(ADDITIONAL_OBJS) $(LIBS) $(INST_LIBS) $(OXF_LIBS)
$(SOCK_LIB) >> $(LNK_OPTIONS_FILE) $(LINK_CMD) -@$(LNK_OPTIONS_FILE)
$(ATI_DIR)\PLUS\O\PLUS.LIB -ld -lc -lios -lram $(DLDFILE) $(TARGET_NAME)$(LIB_EXT) :
$(OBJS) $(ADDITIONAL_OBJS) @echo Building library $@ $(LIB_CMD) $(LIB_FLAGS) -r
$(TARGET_NAME)$(LIB_EXT) $(OBJS) $(ADDITIONAL_OBJS) $(LIB_CMD) -sR
$(TARGET_NAME)$(LIB_EXT) clean: @[ @echo off @echo Cleanup $OMCleanOBJS @if exist
$OMFileObjPath $(RM) $OMFileObjPath @if exist $(LNK_OPTIONS_FILE) $(RM)
$(LNK_OPTIONS_FILE) @if exist NuAppInit$(OBJ_EXT) $(RM) NuAppInit$(OBJ_EXT) @if exist
$(OBJ_DIR)/*$(OBJ_EXT) $(RM) $(OBJ_DIR)/*$(OBJ_EXT) @if exist $(TARGET_NAME)$(LIB_EXT)
$(RM) $(TARGET_NAME)$(LIB_EXT) @if exist $(TARGET_NAME)$(EXE_EXT) $(RM)
$(TARGET_NAME)$(EXE_EXT) $(CLEAN_OBJ_DIR) ]

```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the CPP\_CG::<Environment>::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

(Default = Blank)

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.



*Default = NULL*

## **ObjCleanCommand**

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default = @if exist \$OMFileObjPath \$(RM) \$OMFileObjPath*

## **ObjectName**

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = Empty string*

## **ObjectsDirectory**

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = Empty string*

## **ObjExtension**

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*Default = .o*

## **OSFileSystemCaseSensitive**

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Cleared*

## **ParseErrorDescript**

The ParseErrorDescript property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = ([^()]+)([([0-9]+)] [:] (error/warning/fatal error) (.\*)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = ([^()]+)(([0-9]+)) [:] (error/warning/fatal error)*

## ParseSeverityError

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^()]+)(([0-9]+)) [:] (error/fatal error)*

## ParseSeverityWarning

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^()]+)(([0-9]+)) [:] (warning)*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Cleared*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running Rational Rhapsody, whereas the target is the machine running the application. To run remotely, the UseRemoteHost property must be set to True. If UseRemoteHost is True and RemoteHost is blank, the current host name is used for the remote host. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = Empty string*

## ReusableStatechartSwitches

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts. See the upgrade history on the support site for information on this change.

*Default = -DOM\_REUSABLE\_STATECHART\_IMPLEMENTATION*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*Default = .h*

## UpdateBuildSettingsInIDE

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## UseNewBuildOutputWindow

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the CG::General::ShowLogViewAfterBuild property to True.

*Default = Checked*

## UseRemoteHost

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Checked*

## UseTemplateName

In the declarations for members of a template class, some compilers require that you use precede the member type with the keyword "typename" for types with dependent names. Since support for this keyword varies between compilers, the UseTemplateName property is used to specify whether the

"typename" keyword should be included in the generated code.

*Default = Cleared*

## Operation

The Operation metaclass contains properties that control operations.

### ActivityReferenceToAttributes

The ActivityReferenceToAttributes property specifies whether Rational Rhapsody should generate references in the functor object, thereby giving you direct access to the attributes of the class that owns the modeled operation (without the need for this\_). See the section on activity diagrams in the Rational Rhapsody help for information about modeled operations and functor classes.

*Default = Checked*

### AnimAllowInvocation

The AnimAllowInvocation property specifies whether primitive and triggered operations can be called during instrumentation. If an operation is called during animation, its return value is displayed in the output window; if it is traced, the return value is displayed in the console. The possible values are as follows:

- All - Enable all operation calls, regardless of visibility.
- None - Do not enable operation calls.
- Public - Enable calls to public operations only.
- Protected - Enable calls to protected operations only.

*Default = None*

### AnimateTriggeredOperationReturnValue

The AnimateTriggeredOperationReturnValue property allows you to specify that the return values of triggered operations should be displayed in animated sequence diagrams.

*Default = Checked*

### DeclarationModifier

The DeclarationModifier property is used to enable Rational Rhapsody to reverse engineer non-standard keywords that appear in operation declarations. Keywords that appear between the return type and the operation name are stored as the value of this property, and the property is then used during code

generation to re-create the original code.

Since this is a code generation property, it can also be used to add non-standard keywords to code even when reverse engineering is not used.

This property is used in conjunction with the `PreDeclarationModifier` and `PostDeclarationModifier` properties.

*Default = Blank*

## DescriptionInImplementation

By default, descriptions for operations are generated only in the specification files. Using the properties `GenerateDescriptionInImplementation` and `DescriptionInImplementation`, you can have a descriptive comment generated in the implementation file as well.

When the value of the property `GenerateDescriptionInImplementation` is set to `"UseDescriptionInImplementationProperty"`, the text entered for the property `DescriptionInImplementation` is generated as a comment before the operation definition.

If the value of the property `GenerateDescriptionInImplementation` is set to `"UseDescriptionInImplementationProperty"` and the value of the property `DescriptionInImplementation` is left blank, no comment is generated before the operation definition.

Note that the text that was specified for the `DescriptionInImplementation` property is always generated as a comment before the operation definition, whether the definition appears in the specification file or in the implementation file. Similarly, the text that was specified on the `Description` tab is always generated as a comment before the operation declaration, whether the declaration appears in the specification file or in the implementation file.

During roundtripping, any changes that were made to the comment that precedes the operation definition are ignored.

*Default = Blank*

## DescriptionTemplate

The `DescriptionTemplate` property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element

descriptions.

- `$Name` - the element name
- `$FullName` - the full path of the element (P1::P2::C.a)
- `$Description` - the text provided on the Description tab of the Features window for the element
- `$Requirements` - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the `$Requirements` keyword in the `DescriptionTemplate` property to get such comments generated for specific model elements.
- `$Login` - the login name of the user who generated the file
- `$CodeGeneratedDate` - the date on which the code was generated
- `$CodeGeneratedTime` - the time at which the code was generated
- `$RhapsodyVersion` - the version of Rhapsody that was used to generate the file
- For operations: `$Signature` - the operation signature, `$Arguments` - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: `$Type` - the type of the attribute
- For arguments: `$Type` - the argument type, `$Direction` - the argument direction (In, Out, or InOut)
- For events: `$Arguments` - the text provided on the Description tab of the Features window for the event arguments
- For association ends: `$Target` - the other end of the association
- For requirements: `$ID` - the ID specified for the requirement, `$Specification` - the specification text for the requirement
- The values of tags can be included by using the syntax `$tag_name`. For example, if you use a tag called `author` for your classes, you can use the keyword `$author` in the value of `DescriptionTemplate` in order to have the value of the `author` tag included in the element description.
- The values of properties can be included by using the syntax `$property_name`. For example, you can use the keyword `$Animate` in the value of `DescriptionTemplate` in order to have the value of the `Animate` property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as `$Name`)
- Tag keywords
- Property keywords

If the value of a keyword is a `MultiLine` value, then each new line (except the first one) starts with the value of the `<lang>_CG::Configuration::DescriptionBeginLine` property, and each line ends with the value of the `<lang>_CG::Configuration::DescriptionEndLine` property.

*Default = Blank*

## DisableAutoGeneratedInitializer

When Rational Rhapsody generates code, certain code is automatically generated in the initializer of the constructor. The `DisableAutoGeneratedInitializer` property allows you to disable the generation of this auto-generated initializer code for class elements in the constructor.

Note that when auto-generated initializer code is enabled (`DisableAutoGeneratedInitializer` is set to `False`), elements in the constructor's initialization list are ordered in a way that prevents potential compiler warnings. If you disable auto-generated initializer code by setting the value of `DisableAutoGeneratedInitializer` to `True`, then the initialization code that is generated is the exact code that you entered in the "Initializer" field in the Features window for the constructor.

If you reverse engineer a class, the value of this property is changed to `True` for the class.

*Default = False (default is True in code-centric settings)*

## EntryCondition

*The EntryCondition property specifies the task guard. Default = Empty string*

## FullQualifiedTypeName

To specify that the full qualified name should be generated when elements are used in contexts such as return types, set the value of the property `FullQualifiedTypeName` to `True`.

The value of this property can be set separately for attributes, relations, arguments, and operation return types (use `CPP_CG::Operation::FullQualifiedTypeName` for return types).

The property can be set for specific contexts such as specific attributes or operations, or it can be set at the package or project level, for example, using the full qualified name for return types in a specific package.

*Default = False*

## GenerateDescriptionForFlowchartActions

Beginning in release 8.0.6, when code is generated for flowcharts, comments are generated to represent the descriptions entered for actions and transitions. To preserve the previous code generation behavior for pre-8.0.6 models, the `[lang]_CG::Operation::GenerateDescriptionForFlowchartActions` and `[lang]_CG::Operation::GenerateDescriptionForFlowchartFlows` properties were added to the backward compatibility settings for C and C++ with a value of `False`. If you want to use the new code generation behavior with pre-8.0.6 models, change the value of these properties to `True` and reload the model.

*Default = True*

## GenerateDescriptionForFlowchartFlows

Beginning in release 8.0.6, when code is generated for flowcharts, comments are generated to represent the descriptions entered for actions and transitions. To preserve the previous code generation behavior for pre-8.0.6 models, the `[lang]_CG::Operation::GenerateDescriptionForFlowchartActions` and `[lang]_CG::Operation::GenerateDescriptionForFlowchartFlows` properties were added to the backward

compatibility settings for C and C++ with a value of False. If you want to use the new code generation behavior with pre-8.0.6 models, change the value of these properties to True and reload the model.

*Default = True*

## **GenerateDescriptionInImplementation**

By default, descriptions for operations are generated only in the specification files. Using the properties `GenerateDescriptionInImplementation` and `DescriptionInImplementation`, you can have a descriptive comment generated in the implementation file as well.

The possible values for `GenerateDescriptionInImplementation` are:

- `UseSpecificationText` - the text from the Description tab will be generated before the operation definition as well as before the operation declaration
- `UseDescriptionInImplementationProperty` - the text specified for the property `DescriptionInImplementation` will be generated as a comment before the operation definition

Note that the text that was specified for the `DescriptionInImplementation` property is always generated as a comment before the operation definition, whether the definition appears in the specification file or in the implementation file. Similarly, the text that was specified on the Description tab is always generated as a comment before the operation declaration, whether the declaration appears in the specification file or in the implementation file.

*Default = UseDescriptionInImplementationProperty*

## **GenerateImplementation**

The `GenerateImplementation` property specifies whether to generate the body for the operation. To generate Import pragmas in Rational Rhapsody Developer for Ada, set this property to False and add the "pragma..." declaration in the `CPP_CG::Operation::SpecificationEpilog` property.

*Default = True*

## **GenerateReturnStatementInVoidOperations**

When code is generated for flowcharts, all operations include a return statement, even operations declared as void. If you do not want to have a return statement generated for void operations, set the value of `GenerateReturnStatementInVoidOperations` to False.

*Default = True*

## **ImplementActivityDiagram**

The `ImplementActivity Diagram` property enables or disables code generation for activity diagrams.

*Default = Cleared*



## ImplementationEpilog

The ImplementationEpilog property enables you to add any code that you want to be added as verbatim text (to be ignored by Rational Rhapsody) to the end of the definition of a model element. For example, you could wrap a section of code with an `#ifdef-#endif` pair, add compiler-specific keywords, or add a `#pragma` statement. For example, to specify that an operation is available only when the code is compiled with `_DEBUG`, set the following properties for the operation:

- Set SpecificationProlog to `#ifdef _DEBUG cr.`
- Set SpecificationEpilog to `#endif.`
- Set ImplementationProlog to `#ifdef _DEBUG cr.`
- Set ImplementationEpilog to `#endif.`

The following table lists whether leading and trailing linefeeds are generated.

Metaclass	Leading Linefeed	Added?	Class	Yes	Package	No
-----------	------------------	--------	-------	-----	---------	----

*Default = Empty MultiLine*

## ImplementationName

The ImplementationName property gives an operation one model name and generate it with another name. It is introduced as a workaround that generates const and non-const operations with the same name. For example:

- Create a class A.
- Add a non-const operation `f()`.
- Add a const operation `f_const()`.
- Set the `CPP_CG::Operation::ImplementationName` property for `f_const()` to “f.”
- Generate the code.

The resulting code is as follows: `class A { ... void f(); /* the non const f */ ... void f() const; /* actually f_const() */ ... };` The creation of two operations with the same signature, differing only in whether it is a const, is a common practice in C++, especially for STL users.

*Default = Empty string*

## ImplementationProlog

The ImplementationProlog property enables you to add any code that you want to be added as verbatim text (to be ignored by the software) to the beginning of the definition of a model element. For example, you could wrap a section of code with an `#ifdef-#endif` pair, add compiler-specific keywords, or add a `#pragma` statement. For example, to specify that an operation is available only when the code is compiled with `_DEBUG`, set the following properties for the operation:

- Set SpecificationProlog to `#ifdef _DEBUG cr.`
- Set SpecificationEpilog to `#endif.`

- Set ImplementationProlog to `#ifdef _DEBUG` cr.
- Set ImplementationEpilog to `#endif`.

The following table lists whether leading and trailing linefeeds are generated.

Metaclass Trailing Linefeed Added? Class No Package Yes

*Default = Empty MultiLine*

## ImplementFlowchart

ImplementFlowchart property that specifies whether or not code should be generated for the flow charts created by the user. It can be set at the individual operation level or at higher levels, such as class or package.

*Default = Checked*

## Inline

The Inline property specifies how inline operations are generated. Which operations are affected by the Inline property depends on the metaclass:

- Attribute - Applies only to operations that handle attributes (such as accessors and mutators)
- Operation - Applies to all operations
- Relation - Applies only to operations that handle relations

Inlining in Rational Rhapsody Developer for C++ The possible values for the Inline property are as follows:

- none - The operation is not generated inline.
- in\_header - The operation is generated inline in the specification file.
- in\_source - The operation is generated inline in the implementation file.
- in\_declaration - A class operation is generated inline in the class declaration. A global function is generated inline in the package specification file.

Inlining an operation in the header might cause problems if the function body refers to other classes. For example, if the inlined code refers to another class (by way of a pointer such as `itsRelatedClass`), inlined code that is generated in a header might not compile. The implementation file for the class would have an `#include` for `RelatedClass`, but the specification file would not. The workaround is to create a Usage dependency of the class with the inlined function on the related class. This forces an `#include` of the related class to be generated in the header of the dependent class with the inlined function.

*Default = none*

## IsEntry

The IsEntry property indicates whether the operation is a task entry or a regular operation in `AdaTask` and `AdaTaskType` classes.

*Default = False*

## **IsExplicit**

The IsExplicit property is a Boolean value that allows you to specify that a constructor is an explicit constructor.

*Default = Cleared*

## **IsNative**

The IsNative property specifies whether the Java modifier “native” should be added to an operation in the source file. The body of such operations, if specified, is ignored by the code generator.

*Default = False*

## **IsVolatile**

The IsVolatile property allows you to specify that a specific operation argument should be declared as volatile.

*Default = Cleared*

## **Kind**

The Kind property specifies the kind of operation that should be generated for an element. The kind of operations that can be generated is language-dependent (for example, virtual and abstract exist only in C++ and Java). In Java, Kind can be defined only for attributes and operations, but not for relations. This property affects class operations, in addition to accessors and mutators for relations and attributes. The possible values are as follows:

- common - Class operations and accessor/mutator are non-virtual.
- abstract - Class operations and accessor/mutator are pure virtual. This type is valid for C++ and Java only.

*Default = common*

## **LocalVariablesDeclaration**

The LocalVariablesDeclaration property specifies variables that you want to appear in the declaration of the entypoint or operation.

*Default = Empty string*

## **MarkPrologEpilogInAnnotations**

The `MarkPrologEpilogInAnnotations` property specifies whether to generate ignore annotations for the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties so they are ignored during roundtrip. When you insert code element declarations (variables, types, functions, and so on) in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, after a full roundtrip those elements are added to the model and are duplicated on the next code generation. When you use the `MarkPrologEpilogInAnnotations` property, you can have Rational Rhapsody automatically ignore the information specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties instead of adding the ignore annotations manually. The possible values for the `MarkPrologEpilogInAnnotations` property are:

- **None** - Rational Rhapsody does not generate any annotations. Any models created before Version 4.1 automatically have this property setting.
- **Ignore** - Rational Rhapsody generates the `///  
ignore` annotation before the code specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, and generates the `///  
]` annotation after the code specified in those properties.
- **Auto** - If the code in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties is one line (it does not contain any newline characters (`\n`)), no annotations are generated (the same behavior has the **None** setting). If there is more than one line, Rational Rhapsody generates the `///  
ignore` annotation before the code specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, and generates the `///  
]` annotation after the code specified in those properties (the same behavior as the **Ignore** setting).

During roundtrip, any ignore annotations in the comments of the element are not included in its description. Because the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties are generated between the annotation and its declaration for the element, you cannot rename those elements on roundtrip. If you change the name of an element, it is removed from the model and added with the new name. Some model information (for example, property settings) might be lost.

*Default = None*

## Me

*The `Me` property specifies the name of the first argument to operations generated in C. (Default = `me`)*

## MeDeclType

The `MeDeclType` property is a string that specifies the type of the first argument to operations generated in C, as a pointer to an object or object type. The default value is as follows: `$ObjectName* const` The variable `$ObjectName` is replaced with the name of the object or object type.

## MultiLineArgumentList

The `MultiLineArgumentList` property specifies how the list of arguments for an operation should be formatted.

If set to `False`, all of the arguments appear on the same line as the operation name.

If set to True, each argument appears on a separate line.

*Default = False*

## OpeningBraceStyle

The OpeningBraceStyle property controls where the opening brace of the code block is positioned - on the same line as the element name (SameLine) or on the following line (NewLine).

*Default = SameLine*

## OrderedConstructorInitializer

In release 8.0.6, the order of items included in a constructor's initialization list was improved to prevent potential compiler warnings.

This improved code generation for initialization lists can be turned off by setting the value of the property OrderedConstructorInitializer to False.

Note that this improved ordering of elements in the initialization code is used only when the value of the property CPP\_CG::Operation::DisableAutoGeneratedInitializer is set to False. If you disable auto-generated initializer code by setting the value of DisableAutoGeneratedInitializer to True, then the initialization code that is generated will be the exact code that you entered in the "Initializer" field in the Features window for the constructor.

*Default = True*

## PostDeclarationModifier

The PostDeclarationModifier property is used to enable Rational Rhapsody to reverse engineer non-standard keywords that appear in operation declarations. Keywords that appear after the operation argument list are stored as the value of this property, and the property is then used during code generation to re-create the original code.

Since this is a code generation property, it can also be used to add non-standard keywords to code even when reverse engineering is not used.

This property is used in conjunction with the PreDeclarationModifier and DeclarationModifier properties.

*Default = Blank*

## PreDeclarationModifier

The PreDeclarationModifier property is used to enable Rational Rhapsody to reverse engineer non-standard keywords that appear in operation declarations. Keywords that appear before the return type are stored as the value of this property, and the property is then used during code generation to re-create the original code.

Since this is a code generation property, it can also be used to add non-standard keywords to code even when reverse engineering is not used.

This property is used in conjunction with the `DeclarationModifier` and `PostDeclarationModifier` properties.

*Default = Blank*

## PrivateQualifier

The `PrivateQualifier` property specifies the qualifier that is printed at the beginning of a private operation declaration or definition. You can set this property to an empty string to prevent the generation of the static qualifier in the private function declaration or definition.

*Default = static*

## ProtectedName

The `ProtectedName` property specifies the pattern used to generate names of private operations in C. The default value is as follows: `$opName` The `$opName` variable specifies the name of the operation. For example, the generated name of a private operation `go()` of an object A is generated as: `go()`

## PublicName

The `PublicName` property specifies the pattern used to generate names of public operations in C. The default value is as follows: `$objectName_$opName` The `$objectName` variable specifies the name of the object; the `$opName` variable specifies the name of the operation. For example, the generated name of a public operation `go()` of an object A is generated as: `A_go()`

## PublicQualifier

The `PublicQualifier` property specifies the qualifier that is printed at the beginning of a public operation declaration or definition. Note that the Static check box in the operation window UI is disabled in Rational Rhapsody Developer for C because the check box is associated with class-wide semantics that are not supported by Rational Rhapsody Developer for C. When loading models from previous versions, the Static check box is cleared; if the operation is public, the `C_CG::Operation::PublicQualifier` property value is set to Static in order to generate the same code.

*Default = Empty string*

## Renames

The `Renames` property enables one element to rename another element of the same type. You can also rename an element by using a renames dependency. In the case of a conflict, the dependency has precedence. Note the following:

- For attributes, this property works only for static attributes in a class or for attributes in a package.

- For operations, this property contains the name of the operation this operation in renaming. The signatures of the two operations must match.

*Default = Empty string*

## RenamesKind

The `RenamesKind` property specifies whether the renaming of the operation designated in the `CPP_CG::Operation::Renames` property is "as specification" or "as body."

*Default = Specification*

## ReturnTypeByAccess

The `ReturnTypeByAccess` property determines whether the return type is generated as an access type or a regular type. Note that this property is applicable only to classes for which an access type is generated.

*Default = False*

## Simplify

If you are using the Rational Rhapsody customizable code generation mechanism, the `Simplify` property can be used to change the way specific types of elements are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - The element is ignored.
- Copy - The element is copied from the original to the simplified model. It is not modified in any way.
- Default - Uses the standard simplification for this item, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for the element has been applied.

*Default = "Default"*

## SimplifyTriggeredOperation

If you are using the Rational Rhapsody customizable code generation mechanism, the `SimplifyTriggeredOperation` property can be used to change the way triggered operations are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - Triggered operations are ignored.
- Copy - Triggered operation are copied from the original to the simplified model. They are not modified

in any way.

- **Default** - Uses the standard simplification for triggered operations, as defined in Rational Rhapsody.
- **ByUser** - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- **ByUserPostDefault** - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for triggered operations has been applied.

*Default = "Default"*

## SpecificationEpilog

The SpecificationEpilog property allows you to add code to the end of the declaration of a model element.

For example, you can use this property to add an `#ifdef` to indicate that an operation is available only when the code is compiled with `_DEBUG`, by setting the following properties:

- For SpecificationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For SpecificationEpilog, enter the value `#endif`
- For ImplementationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For ImplementationEpilog, enter the value `#endif`

*Default = Blank*

## SpecificationProlog

The SpecificationProlog property allows you to add code to the beginning of the declaration of a model element.

For example, you can use this property to add an `#ifdef` to indicate that an operation is available only when the code is compiled with `_DEBUG`, by setting the following properties:

- For SpecificationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For SpecificationEpilog, enter the value `#endif`
- For ImplementationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For ImplementationEpilog, enter the value `#endif`

*Default = Blank*

## TaskDefaultScheme

The TaskDefaultScheme property sets the task default entry scheme. The possible values are as follows:

- Conditional
- Timed
- None



*Default = None*

## **TaskDefaultSchemeDelayStatement**

The TaskDefaultScheme property sets the task default entry statement for timed entry schemes.

*Default = Empty MultiLine*

## **ThisByAccess**

The ThisByAccess property specifies whether to pass the this parameter as an access mode parameter for a non-static operation.

*Default = False*

## **ThisName**

*The ThisName property specifies the name of the this parameter, which specifies the instance. (Default = this)*

## **ThrowExceptions**

The ThrowExceptions property specifies the exceptions that an operation can throw. Separate multiple exceptions with commas.

*Default = Empty string*

## **UsePre82Flowchart**

Prior to release 8.2, in cases where an "else" guard contained blank spaces, code was not generated correctly for flowcharts. This issue was corrected in 8.2. To preserve the previous code generation behavior for pre-8.2 models, the property [lang]\_CG::Operation::UsePre82Flowchart was added to the backward compatibility settings for C and C++ with a value of True.

## **UsePre821DescriptionInImplementation**

Prior to release 8.2.1, if you used the property DescriptionInImplementation to define a description to be generated in the implementation of an operation, and you also modified the value of the property TemplateDescription (for the specification description), the comment for the implementation description was not generated correctly. This issue was corrected in 8.2.1. To preserve the previous code generation behavior for pre-8.2.1 models, the property [lang]\_CG::Operation::UsePre821DescriptionInImplementation was added to the backward compatibility settings for C and CPP with a value of True.

## VirtualMethodGenerationScheme

The VirtualMethodGenerationScheme property enables backward-compatibility mode for methods of interface and abstract classes. The possible values are as follows:

- Default - The class type is class-wide, but the this parameters are not.
- ClassWideOperations - The class type is not class-wide, but the this parameters are.

*Default = Default*

## OsePPCDiab

The OsePPCDiab metaclass contains environment settings (Compiler, framework libraries, and so on) for OSE Delta RTOS compiled for PPC, by using Diab compiler.

## AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)/LangCpp/osconfig/OSE*

## AdditionalReservedWords

The AdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = Empty string*

## **BuildArgumentsInIDE**

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the CPP\_CG::[environment]::BuildCommandInIDE property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## **BuildCommandInIDE**

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## **BuildCommandSet**

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration.

To change this property, use the Configuration window in the browser - do not change it by using the Properties tab in the Features window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link. The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*Default = Debug*

## **BuildInIDE**

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## **CompileSwitches**

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property.

*Default =*

`-I. -I$OMDefaultSpecificationDirectory -I$(OMROOT)/LangCpp $(INST_FLAGS)  
$(INCLUDE_PATH) $(INST_INCLUDES)`

## **ConvertHostToIP**

The ConvertHostToIP property specifies whether to convert the host name to an IP number. This is necessary because pSOSystem does not include a name service.

*Default = Checked*

## **CPPCompileCommand**

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = Empty MultiLine*

## **CPPCompileDebug**

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

Default = -g

## **CPPCompileRelease**

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

Default = -O

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

*Default =*

\$OMFileObjPath : \$OMFileImpPath \$OMFileDependencies

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: \$OMFileObjPath : \$OMFileImpPath "\*.bmp" "\*.ico" "\*.rc2"

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction with an IDE such as Eclipse, the EnableDebugIntegrationWithIDE property can be used to specify whether or not the IDE debugger should be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to True, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's exit() function is called automatically. The system function calls the destructors of framework instances in the order it determines. The EndApplicationCode property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an endApplication() function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to endApplication(), it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The EntryPoint property specifies the name of the main program for a given environment.

*Default = rhposemain*

See also the definition of the EntryPointDeclarationModifier property for more information.

## EntryPointDeclarationModifier

The EntryPointDeclarationModifier property specifies a modifier for the entry point declaration. This property allows generation of the main() function in the specified syntax.

To modify the main() signature implemented in the OSE adapter, do the following:

- Add the EntryPointDeclarationModifier property to your environment properties and set it to the main return value and name. For example: "int main"
- Set the EntryPoint property to the main arguments. For example: "int a, long b, char\*\*"
- Generate the code.

You get the following main() declaration:

```
int main(int a, long b, char** c) { ... }
```

*Default = OS\_PROCESS*

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens has three parameters, each with an integer value:

- TotalNumberOfTokens - The number of tokens in the regular expression
- FileTokenPosition - The position of the file name token in the expression
- LineTokenPosition - The position of the line number token in the expression

*Default = TotalNumberOfTokens=3,FileTokenPosition=1,LineTokenPosition=2*

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the CPP\_CG::<Environment>::ExeName property plus the value of this property.

*(Default = .elf)*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the `CPP_CG::<Environment>::ExeExtension` property.

*(Default = Blank)*

## FileDependencies

The `FileDependencies` property specifies which framework specification files and implementation files should be included in model elements. The file inclusions are generated in the makefile.

*Default = Empty string*

## GeneratedAllDependencyRule

The `GeneratedAllDependencyRule` property specifies whether to automatically generate the “all:” rule as part of the expansion of the `$OMContextMacros` keyword in the makefile. If this is Cleared, you can define the makefile macros manually.

*Default = Cleared*

## ImpExtension

The `ImpExtension` property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment. The default values is as follows:

*Default = .cc*

## Include

The `Include` property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*Default = include*

## InvokeExecutable

The `InvokeExecutable` property specifies the command used to run an executable file.

*Default = "\$OMROOT/etc/osesfkRun.bat" \$executable*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

```
Default = "$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\oseppcdiabmake.bat\" $makefile  
$maketarget"
```

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

Default value = .a

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

*Default = -g*

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

*Default = Empty string*

## LinkSwitches



The LinkSwitches property specifies the standard link switches used to link in any mode.

*Default = \$OMLinkCommandSet*

## MainIncludes

The MainIncludes property is a string that specifies the files that need to be included in the main program generated for an application.

*Default = <ose.h>*

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the CPP\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*(Default = .mk)*

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration.

```
Default = ##### Target type (Debug/Release) #####  
#####  
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease  
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease  
ConfigurationCPPCompileSwitches=$OMReusableStatechartSwitches  
$OMConfigurationCPPCompileSwitches #####
```

```

Predefined macros ##### INCLUDE_QUALIFIER=-I LIB_CMD=dar LIB_FLAGS=rv
LINK_FLAGS=$OMConfigurationLinkSwitches #####
##### Context macros ##### $OMContextMacros
#####
#.PHONY : all .DEFAULT : all LIB_PREFIX = OSE LIB_POSTFIX = PPC$(PROCESSOR) .IF
$(TARGET_TYPE) == Executable OBJS += $OMFileObjPath .END .IF $(INSTRUMENTATION) ==
Animation INST_FLAGS=-DOMANIMATOR
INST_INCLUDES=$(INCLUDE_QUALIFIER)$(OMROOT)/LangCpp$/aom
$(INCLUDE_QUALIFIER)$(OMROOT)/LangCpp$/tom INST_LIBS=
$(OMROOT)/LangCpp$/lib$/$(LIB_PREFIX)aomanim$(LIB_POSTFIX)$(LIB_EXT)
OXF_LIBS=$(OMROOT)/LangCpp$/lib$/$(LIB_PREFIX)oxfirst$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)/LangCpp$/lib$/$(LIB_PREFIX)omcomappl$(LIB_POSTFIX)$(LIB_EXT) .ELIF
$(INSTRUMENTATION) == Tracing INST_FLAGS=-DOMTRACER
INST_INCLUDES=$(INCLUDE_QUALIFIER)$(OMROOT)/LangCpp$/aom
$(INCLUDE_QUALIFIER)$(OMROOT)/LangCpp$/tom
INST_LIBS=$(OMROOT)/LangCpp$/lib$/$(LIB_PREFIX)tomtrace$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)/LangCpp$/lib$/$(LIB_PREFIX)aomtrace$(LIB_POSTFIX)$(LIB_EXT) OXF_LIBS=
$(OMROOT)/LangCpp$/lib$/$(LIB_PREFIX)oxfirst$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)/LangCpp$/lib$/$(LIB_PREFIX)omComAppl$(LIB_POSTFIX)$(LIB_EXT) .ELIF
$(INSTRUMENTATION) == None INST_FLAGS= INST_INCLUDES= INST_LIBS=
OXF_LIBS=$(OMROOT)/LangCpp$/lib$/$(LIB_PREFIX)oxf$(LIB_POSTFIX)$(LIB_EXT)
SOCK_LIB= .ELSE MAKEFILE_ERROR = yes ERROR_TYPE = user ERROR_MSG = An invalid
Instrumentation INSTRUMENTATION=$(INSTRUMENTATION) is specified. .END
##### usage
.PHONY: $(ECHO)Available make targets are: $(ECHOEND) $(ECHO) clean - delete the directory
$(OBJ) and all its files.$(ECHOEND) $(ECHO) all - build executable file.$(ECHOEND)
$(ECHOEMPTY) #####
# SETS HOST TO EITHER UNIX OR WIN32
##### HOST =
$(eq,$(OS),unix UNIX WIN32)
##### # READ THE
USER CONFIGURATION FILE
##### # The
USERCONF macro can be overridden on the command line. E.g. # > dmake USERCONF=~/.myconf.mk all
#USERCONF *= ./userconf.mk #include $(USERCONF)
##### # THE USER
CONFIGURATION
##### USERCONF *=
$(OMROOT)/MakeTpl$/oseDiabPPCconf.mk include $(USERCONF)
#####
CXXFLAGS += $(ConfigurationCPPCompileSwitches) .IF $(COMPILER) == DIAB DEFINES +=
-D__DIAB .END LIBRARIES += $(INST_LIBS) $(OXF_LIBS) $(LIBS) $(SOCK_LIB)
##### OBJ =
./obj OBJ_SUBDIR = .IF $(OBJ) != $(NULL) .IF $(OBJ) != . $(OBJ) .IGNORE: $(ECHO)Create: $@
$(ECHOEND) $(MKDIR) $(OBJ) .IF $(OBJ_SUBDIR) != $(NULL) $(MKDIR) $@ .END all: $(OBJ)
CLEAN_OBJ .PHONY: $(RMDIR) $(OBJ) CLEAN += CLEAN_OBJ .END .END SRC = . INCLUDE +=
-I$(OBJ) INCLUDE += -I. EXAMPLES_COMMON_CONF *= $(EXAMPLES_COMMON)/conf
EXAMPLES_COMMON_INCLUDE *= $(EXAMPLES_COMMON)/include
EXAMPLES_COMMON_MAKE *= $(EXAMPLES_COMMON)/make EXAMPLES_COMMON_SRC *=
$(EXAMPLES_COMMON)/src INCLUDE += -I$(EXAMPLES_COMMON_INCLUDE) # Inclusion of
your common settings. # In this file, you can enter constants to be used for all # examples, such as
COMPILER, COMPILERROOT and so on include
$(EXAMPLES_COMMON_MAKE)/common_settings.mk .IF $(HOST) == UNIX include
$(EXAMPLES_COMMON_MAKE)/tools-unix.mk .ELSE include
$(EXAMPLES_COMMON_MAKE)/tools-win32.mk .END .IF $(TARGET_TYPE) == Library

```

```

$(TARGET_NAME)$(LIB_EXT) : $(OBJ)$/$(OBJS)} $(OBJ)$/$(ADDITIONAL_OBJS)}
$(OMMakefileName) @+echo Creating $@ library file $(ECHOEND) @$(LIB_CMD) $(LIB_FLAGS)
$(TARGET_NAME)$(LIB_EXT) $(OBJ)$/$(OBJS)} $(OBJ)$/$(ADDITIONAL_OBJS)} all:
$(TARGET_NAME)$(LIB_EXT) $OMModelLibs .END clean: @echo Cleanup .IF
$(ADDITIONAL_OBJS) != $(NULL) $(RM) $(OBJ)$/$(ADDITIONAL_OBJS)} .END .IF
$(TARGET_TYPE) == Library $(RMDIR) $(OBJ) $(RM) $(TARGET_NAME)$(LIB_EXT) .ELSE $(RM)
$(TARGET_NAME)$(EXE_EXT) .END ## Find out something about the specific target #
##### ## Define statements #
##### USE_OSEDEF_H *= yes
##### ## Fetch information on CPU and BSP
for the selected board # ##### include
$(EXAMPLES_COMMON_MAKE)$select_cpu_and_bsp.mk
##### ## Signal files #
#####
##### ## Objects #
##### .IF $(EXECUTABLE_FILE_TYPE) !=
load_module .IF $(INCLUDE_OSE_EFS) == yes OBJECTS += startefs.o # This file is located in #
<OSEROOT>/<PLATFORM>/src, and is # an example on how to start EFS .ELIF
$(INCLUDE_OSE_SHELL) == yes OBJECTS += startshell.o # This file is located in #
<OSEROOT>/<PLATFORM>/src, and is # an example on how to start SHELL .END .IF
$(INCLUDE_OSE_INET) == yes OBJECTS += startinet.o # This file is located in #
<OSEROOT>/<PLATFORM>/src, and is # an example on how to start INET .END .IF
$(INCLUDE_OSE_PRH) == yes OBJECTS += start_prh.o # This file is located in #
<OSEROOT>/<PLATFORM>/src, and is # an example on how to start PRH .END .END OBJECTS +=
$(OBJS) # Error handler: .IF $(EXECUTABLE_FILE_TYPE) != load_module OBJECTS += err_hnd.o
.END # Early Error Handler: # To be used if MMS or MMH (via PRH) .IF $(INCLUDE_OSE_MMS) ==
yes OBJECTS += early_error.o .ELIF $(INCLUDE_OSE_MMS) == mmh OBJECTS += early_error.o
.ELIF $(INCLUDE_OSE_PRH) == yes OBJECTS += early_error.o .END
##### ## Libraries #
#####
##### ## Contribution to architecture
specific kernel # configuration. # powerpc : ospp.con # mips : krn.con # arm : osarm.con # m68000 :
os68.con # ##### .IF $(TARGET_ARCH) ==
powerpc OSPP_CON_CONTRIBUTORS != $(OBJ)/osarch_con_from_example.con
$(OSPP_CON_CONTRIBUTORS) .ELIF $(TARGET_ARCH) == m68000 OS68_CON_CONTRIBUTORS
!= $(OBJ)/osarch_con_from_example.con $(OS68_CON_CONTRIBUTORS) .ELIF $(TARGET_ARCH)
== mips KRN_CON_CONTRIBUTORS != $(OBJ)/osarch_con_from_example.con
$(KRN_CON_CONTRIBUTORS) .ELIF $(TARGET_ARCH) == arm4t1e
OSARM_CON_CONTRIBUTORS != $(OBJ)/osarch_con_from_example.con
$(OSARM_CON_CONTRIBUTORS) .ELIF $(TARGET_ARCH) == arm4t1be
OSARM_CON_CONTRIBUTORS != $(OBJ)/osarch_con_from_example.con
$(OSARM_CON_CONTRIBUTORS) .ELIF $(TARGET_ARCH) == sarm1e
OSARM_CON_CONTRIBUTORS != $(OBJ)/osarch_con_from_example.con
$(OSARM_CON_CONTRIBUTORS) .ELIF $(TARGET_ARCH) == sarm1be
OSARM_CON_CONTRIBUTORS != $(OBJ)/osarch_con_from_example.con
$(OSARM_CON_CONTRIBUTORS) .END $(OBJ)/osarch_con_from_example.con .PRECIOUS:
$(MAKEFILE:s\-\f\ \) $(USERCONF) $(ECHO) Create: $@$(ECHOEND) $(ECHOEMPTY) >$@
##### ## Contribution to osemmain.con #
##### OSEMAIN_CON_CONTRIBUTORS
!= $(OBJ)/osemain_con_from_example.con $(OSEMAIN_CON_CONTRIBUTORS)
$(OBJ)/osemain_con_from_example.con .PRECIOUS: $(MAKEFILE:s\-\f\ \) $(USERCONF)
$(ECHOEMPTY)>$@ $(ECHO)/ * The entries below are added by makefile.mk */$(ECHOEND)>>$@
$(ECHO)/ * They represent the parameters for the application. */$(ECHOEND)>>$@ .IF
$(EXECUTABLE_FILE_TYPE) != load_module .IF $(INCLUDE_OSE_EFS) == yes
$(ECHO)PRI_PROC(start_efs, start_efs, 1023, 9, DEFAULT, 0, NULL)$(ECHOEND)>>$@ .ELIF

```

```

$(INCLUDE_OSE_SHELL) == yes $(ECHO)PRI_PROC(start_shell, start_shell, 1023, 9, DEFAULT, 0,
NULL)$(ECHOEND)>>@$@ .END .IF $(INCLUDE_OSE_INET) == yes $(ECHO)PRI_PROC(init_inet,
init_inet, 256, 9, DEFAULT, 0, NULL)$(ECHOEND)>>@$@ .END .END .IF $(INCLUDE_OSE_PRH) ==
yes $(ECHO)PRI_PROC(start_prh, start_prh, 256, 10, DEFAULT, 0, NULL)$(ECHOEND)>>@$@ #
$(ECHO)START_OSE_HOOK2(start_prh_hook) $(ECHOEND)>>@$@ .END .IF $(TARGET_TYPE) ==
Executable $(ECHO)PRI_PROC($OMMainName, $OMMainName, 1000, 5, DEFAULT, 0, NULL)
$(ECHOEND)>>@$@ .END ##### # #
Contribution to softose.con % Softkernel environments #
##### .IF $(USE_OSEDEF_H) == yes
include $(EXAMPLES_COMMON_MAKE)/osedef.mk .END
##### # #
Inclusion of OSE products #
##### include
$(EXAMPLES_COMMON_MAKE)/products.mk # The COMPILERMAKE macro is assigned in
commonsetup.mk # This has to be done late since this makefile might check things like # USE_MMS and
such things that need to modify like CRT0 EXECUTABLE_NAME = $(TARGET_NAME) include
$(EXAMPLES_COMMON_MAKE)/compiler.mk LCFDEFINES +=
-DIMAGE_START=$(IMAGE_START) LCFDEFINES +=
-DIMAGE_MAX_LENGTH=$(IMAGE_MAX_LENGTH) CXXFLAGS += -Xansi include
$(EXAMPLES_COMMON_MAKE)/compilation_rules.mk $(eq,$(TARGET_TYPE),Library) .EXIT:
.IGNORE: ) include $(EXAMPLES_COMMON_MAKE)/targets.mk
##### #
IMPORT SHELL ENVIRONMENT
##### # Import
the environment variable PATH #.IMPORT: PATH
##### # END
OF MAKEFILE
#####

```

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## ObjCleanCommand

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default = \$(RM) \$OMFileObjPath*

## ObjectName

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = Empty string*

## ObjectsDirectory

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = Empty string*

## ObjExtension

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*Default = .o*

## OSFileSystemCaseSensitive

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Checked*

## ParseErrorDescript

The ParseErrorDescript property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = "([^\"]+)"[,][ ]\*line ([0-9]+)[: ] (error/warning) (.\*)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = "([^\"]+)"[,][ ]\*line ([0-9]+)[: ] (error/warning)*

## ParseSeverityError

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = "[^"]+)",,[ ]\*line ([0-9]+)[: ] (error)*

## **ParseSeverityWarning**

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = "[^"]+)",,[ ]\*line ([0-9]+)[: ] (warning)*

## **QuoteOMROOT**

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Checked*

## **RemoteHost**

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running Rational Rhapsody, whereas the target is the machine running the application. To run remotely, the UseRemoteHost property must be set to True. If UseRemoteHost is True and RemoteHost is blank, the current host name is used for the remote host. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = Empty string*

## **ReusableStatechartSwitches**

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts. See the upgrade history on the support site for information on this change.

*Default = -DOM\_REUSABLE\_STATECHART\_IMPLEMENTATION*

## **SpecExtension**

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*Default = .h*

## **UnixPathNameForOMROOT**

The UnixPathNameForOMROOT property specifies whether the makefile must include UNIXstyle path



names.

The pRISM compilers do not tolerate DOS-style path conventions. If you do not set this property correctly, there might be many compilation problems.

Default value = Cleared

## UpdateBuildSettingsInIDE

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## UseNewBuildOutputWindow

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the CG::General::ShowLogViewAfterBuild property to True.

*Default = Checked*

## UseRemoteHost

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Checked*

## UseTemplateTypename

In the declarations for members of a template class, some compilers require that you use precede the member type with the keyword "typename" for types with dependent names. Since support for this keyword varies between compilers, the UseTemplateTypename property is used to specify whether the "typename" keyword should be included in the generated code.

*Default = Cleared*

## OseSfk

The OseSfk metaclass contains environment settings (Compiler, framework libraries, and so on) for OSE Delta RTOS Win32 simulator compiled by using the Microsoft VC++ compiler.

### AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)/LangCpp/osconfig/OSE*

### AdditionalReservedWords

The AdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default =*

receive \_\_asm \_\_finally naked \_\_based \_\_inline \_\_single\_inheritance \_\_cdecl \_\_int8 \_\_stdcall \_\_declspec  
\_\_int16 \_\_declspec \_\_int32 \_\_try \_\_declspec \_\_int64 \_\_virtual\_inheritance \_\_except \_\_leave \_\_fastcall  
\_\_multiple\_inheritance

### BuildArgumentsInIDE

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of



arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the `CPP_CG:[environment]:BuildCommandInIDE` property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the `BuildInIDE` property is set to `True`.

*Default = Blank*

## BuildCommandInIDE

The `BuildCommandInIDE` property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the `BuildInIDE` property is set to `True`.

*Default = Blank*

## BuildCommandSet

The `BuildCommandSet` property generates a set of commands in the makefile to build a debug or release version of the configuration.

To change this property, use the Configuration window in the browser - do not change it by using the Properties tab in the Features window or by modifying the `site.prp` file. Note that this property also affects the names of the framework libraries used in the link. The possible values are as follows:

- `Debug` - Generate the debug command that is set in the makefile.
- `DebugNoExp` - Generate the debug command that is set in the makefile without the exceptions flag (`:cx_option=exceptions`).
- `Release` - Generate the release command that is set in the makefile.
- `ReleaseNoExp` - Generate the release command that is set in the makefile without the exceptions flag (`:cx_option=exceptions`).

*Default = Debug*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The `buildFrameworkCommand` property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\$OMROOT/etc/Executer.exe" "%1" "\$OMROOT\etc\Osesfkmake.bat osesfkbuilt.mak buildLibs*

```
"LIB_DIR=.\lib\" "BUILD_SET=DEBUG\" "USE_STL=FALSE\" "USE_PDB=FALSE\"  
"BUILD_TARGET=clean all\" " "
```

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## CompileSwitches

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property.

*Default =*

```
-I. -I$OMDefaultSpecificationDirectory -I$(OMROOT)/LangCpp $(INST_FLAGS)  
$(INCLUDE_PATH) $(INST_INCLUDES)
```

## ConvertHostToIP

The ConvertHostToIP property specifies whether to convert the host name to an IP number. This is necessary because pSOSystem does not include a name service.

*Default = Checked*

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = Empty MultiLine*

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*Default = -DOS\_DEBUG /Zi /Od /MDd /Fd"\${TARGET\_NAME}"*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

Default = /Ox /MD /Fd"\${TARGET\_NAME}"

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

*Default =*

\$OMFileObjPath : \$OMFileImpPath \$OMFileDependencies

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: \$OMFileObjPath : \$OMFileImpPath "\*.bmp" "\*.ico" "\*.rc2"

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction with an IDE such as Eclipse, the EnableDebugIntegrationWithIDE property can be used to specify whether or not the IDE debugger should be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to True, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's exit() function is called automatically. The system function calls the destructors of framework instances in the order it determines. The EndApplicationCode property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an endApplication() function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to endApplication(), it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The EntryPoint property specifies the name of the main program for a given environment.

*Default = rhposemain*

See also the definition of the EntryPointDeclarationModifier property for more information.

## EntryPointDeclarationModifier

The EntryPointDeclarationModifier property specifies a modifier for the entry point declaration. This property allows generation of the main() function in the specified syntax.

To modify the main() signature implemented in the OSE adapter, do the following:

- Add the EntryPointDeclarationModifier property to your environment properties and set it to the main return value and name. For example: "int main"
- Set the EntryPoint property to the main arguments. For example: "int a, long b, char\*\*"
- Generate the code.

You get the following main() declaration:

```
int main(int a, long b, char** c) { ... }
```

*Default = OS\_PROCESS*

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens has three parameters, each with an integer value:

- TotalNumberOfTokens - The number of tokens in the regular expression
- FileTokenPosition - The position of the file name token in the expression
- LineTokenPosition - The position of the line number token in the expression

*Default = TotalNumberOfTokens=3,FileTokenPosition=1,LineTokenPosition=2*

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the CPP\_CG::<Environment>::ExeName property plus the value of this property.

*(Default = .exe)*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the CPP\_CG::<Environment>::ExeExtension property.

*(Default = Blank)*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements. The file inclusions are generated in the makefile.

*Default = Empty string*

## GeneratedAllDependencyRule

The GeneratedAllDependencyRule property specifies whether to automatically generate the “all:” rule as part of the expansion of the \$OMContextMacros keyword in the makefile. If this is Cleared, you can define the makefile macros manually.

*Default = Cleared*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment. The default values is as follows:

*Default = .cc*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*Default = include*

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*Default = "\$OMROOT/etc/osesfkRun.bat" \$executable*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

"rhapsody\_dir>\share/etc/Executer.exe""dir; dir | more"

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

"\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\vx6make.bat\" \$makefile \$maketarget 6.2 \$BSP gnu"

Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\osesfkmake.bat\" \$makefile \$maketarget"

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

Default value = .lib

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

*Default = Empty string*

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

*Default = Empty string*

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

*Default = -nologo \$OMLinkCommandSet*

## MainIncludes

The MainIncludes property is a string that specifies the files that need to be included in the main program generated for an application.

*Default = <ose.h>*

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the CPP\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*(Default = .mk)*

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or

Release version of a configuration.

```
Default = ##### Target type (Debug/Release) #####
#####
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease
ConfigurationCPPCompileSwitches=$OMReusableStatechartSwitches
$OMConfigurationCPPCompileSwitches #####
Predefined macros ##### INCLUDE_QUALIFIER = -I LIB_CMD=$(LD) -lib
LIB_FLAGS= LINK_FLAGS = $OMConfigurationLinkSwitches
##### Context macros #####
$OMContextMacros
#####
oseatexit.c: $(CP) "$(OMROOT)"\MakeTmpl\oseatexit.c oseatexit.c #.PHONY : all .DEFAULT : all
LIB_PREFIX = osesfk LIB_POSTFIX = .IF $(TARGET_TYPE) == Executable OBJS +=
$OMFileObjPath .END .IF $(INSTRUMENTATION) == Animation INST_FLAGS=-DOMANIMATOR
-GX INST_INCLUDES=$(INCLUDE_QUALIFIER) $(OMROOT)/LangCpp$/aom
$(INCLUDE_QUALIFIER) $(OMROOT)/LangCpp$/tom INST_LIBS=
$(OMROOT)/LangCpp$/lib$/$(LIB_PREFIX)aomanim$(LIB_POSTFIX)$(LIB_EXT)
OXF_LIBS=$(OMROOT)/LangCpp$/lib$/$(LIB_PREFIX)oxfinst$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)/LangCpp$/lib$/$(LIB_PREFIX)omcomappl$(LIB_POSTFIX)$(LIB_EXT) OBJS +=
oseatexit.o .ELIF $(INSTRUMENTATION) == Tracing INST_FLAGS=-DOMTRACER -GX
INST_INCLUDES=$(INCLUDE_QUALIFIER) $(OMROOT)/LangCpp$/aom
$(INCLUDE_QUALIFIER) $(OMROOT)/LangCpp$/tom
INST_LIBS=$(OMROOT)/LangCpp$/lib$/$(LIB_PREFIX)tomtrace$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)/LangCpp$/lib$/$(LIB_PREFIX)aomtrace$(LIB_POSTFIX)$(LIB_EXT) OXF_LIBS=
$(OMROOT)/LangCpp$/lib$/$(LIB_PREFIX)oxfinst$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)/LangCpp$/lib$/$(LIB_PREFIX)omComAppl$(LIB_POSTFIX)$(LIB_EXT) OBJS +=
oseatexit.o .ELIF $(INSTRUMENTATION) == None INST_FLAGS= INST_INCLUDES= INST_LIBS=
OXF_LIBS=$(OMROOT)/LangCpp$/lib$/$(LIB_PREFIX)oxf$(LIB_POSTFIX)$(LIB_EXT)
SOCK_LIB= .ELSE MAKEFILE_ERROR = yes ERROR_TYPE = user ERROR_MSG = An invalid
Instrumentation INSTRUMENTATION=$(INSTRUMENTATION) is specified. .END
##### usage
.PHONY: $(ECHO)Available make targets are: $(ECHOEND) $(ECHO) clean - delete the directory
$(OBJ) and all its files.$(ECHOEND) $(ECHO) all - build executable file.$(ECHOEND)
$(ECHOEMPTY) #####
# SETS HOST TO EITHER UNIX OR WIN32
##### HOST =
$(eq,$(OS),unix UNIX WIN32)
##### # READ THE
USER CONFIGURATION FILE
##### # The
USERCONF macro can be overridden on the command line. E.g. # > dmake USERCONF=~\myconf.mk all
#USERCONF *= .$/userconf.mk #include $(USERCONF)
##### # THE USER
CONFIGURATION
##### USERCONF *=
$(OMROOT)/MakeTmpl$/oseW32conf.mk include $(USERCONF)
#####
CXXFLAGS += $(ConfigurationCPPCompileSwitches) LIBRARIES += $(INST_LIBS) $(OXF_LIBS)
$(LIBS) $(SOCK_LIB)
##### OBJ =
.~/obj OBJ_SUBDIR = .IF $(OBJ) != $(NULL) .IF $(OBJ) != . $(OBJ) .IGNORE: $(ECHO)Create: $@
$(ECHOEND) $(MKDIR) $(OBJ) .IF $(OBJ_SUBDIR) != $(NULL) $(MKDIR) $@ .END all: oseatexit.c
$(OBJ) CLEAN_OBJ .PHONY: $(RMDIR) $(OBJ) CLEAN += CLEAN_OBJ .END .END SRC = .
```



```

INCLUDE += -I$(OBJ) INCLUDE += -I. EXAMPLES_COMMON_CONF *=
$(EXAMPLES_COMMON)/conf EXAMPLES_COMMON_INCLUDE *=
$(EXAMPLES_COMMON)/include EXAMPLES_COMMON_MAKE *=
$(EXAMPLES_COMMON)/make EXAMPLES_COMMON_SRC *= $(EXAMPLES_COMMON)/src
INCLUDE += -I$(EXAMPLES_COMMON_INCLUDE) # Inclusion of your common settings. # In this
file, you can enter constants to be used for all # examples, such as COMPILER, COMPILERROOT and so
on include $(EXAMPLES_COMMON_MAKE)/common_settings.mk .IF $(HOST) == UNIX include
$(EXAMPLES_COMMON_MAKE)/tools-unix.mk .ELSE include
$(EXAMPLES_COMMON_MAKE)/tools-win32.mk .END .IF $(TARGET_TYPE) == Library
$(TARGET_NAME)$(LIB_EXT) : $(OBJ)/$(OBJS) $(OBJ)/$(ADDITIONAL_OBJS)
$(OMMakefileName) @+echo Creating $@ library file $(ECHOEND) @$(LIB_CMD) $(LIB_FLAGS)
/OUT:$(TARGET_NAME)$(LIB_EXT) $(OBJ)/$(OBJS) $(OBJ)/$(ADDITIONAL_OBJS) all:
$(TARGET_NAME)$(LIB_EXT) $OMModelLibs .END clean: @echo Cleanup .IF
$(ADDITIONAL_OBJS) != $(NULL) $(RM) $(OBJ)/$(ADDITIONAL_OBJS) .END .IF
$(TARGET_TYPE) == Library $(RMDIR) $(OBJ) $(RM) $(TARGET_NAME)$(LIB_EXT) .ELSE $(RM)
$(TARGET_NAME)$(EXE_EXT) .END ## Find out something about the specific target #
##### # # Define statements #
##### USE_OSEDEF_H *= yes
##### # # Fetch information on CPU and BSP
for the selected board # ##### include
$(EXAMPLES_COMMON_MAKE)/select_cpu_and_bsp.mk
##### # # Signal files #
#####
##### # # objects #
##### .IF $(EXECUTABLE_FILE_TYPE) !=
load_module .IF $(INCLUDE_OSE_EFS) == yes OBJECTS += startefs.o # This file is located in #
<OSEROOT>/<PLATFORM>/src, and is # an example on how to start EFS .ELIF
$(INCLUDE_OSE_SHELL) == yes OBJECTS += startshell.o # This file is located in #
<OSEROOT>/<PLATFORM>/src, and is # an example on how to start SHELL .END .IF
$(INCLUDE_OSE_INET) == yes OBJECTS += startinet.o # This file is located in #
<OSEROOT>/<PLATFORM>/src, and is # an example on how to start INET .END .IF
$(INCLUDE_OSE_PRH) == yes OBJECTS += start_prh.o # This file is located in #
<OSEROOT>/<PLATFORM>/src, and is # an example on how to start PRH .END .END OBJECTS +=
$(OBJS) # Error handler: .IF $(EXECUTABLE_FILE_TYPE) != load_module OBJECTS += err_hnd.o
.END # Early Error Handler: # To be used if MMS or MMH (via PRH) .IF $(INCLUDE_OSE_MMS) ==
yes OBJECTS += early_error.o .ELIF $(INCLUDE_OSE_MMS) == mmh OBJECTS += early_error.o
.ELIF $(INCLUDE_OSE_PRH) == yes OBJECTS += early_error.o .END
##### # # Libraries #
#####
##### # # Contribution to architecture
specific kernel # configuration. # powerpc : ospp.con # mips : krn.con # arm : osarm.con # m68000 :
os68.con # ##### .IF $(TARGET_ARCH) ==
powerpc OSPP_CON_CONTRIBUTORS != $(OBJ)/osarch_con_from_example.con
$(OSPP_CON_CONTRIBUTORS) .ELIF $(TARGET_ARCH) == m68000 OS68_CON_CONTRIBUTORS
!= $(OBJ)/osarch_con_from_example.con $(OS68_CON_CONTRIBUTORS) .ELIF $(TARGET_ARCH)
== mips KRN_CON_CONTRIBUTORS != $(OBJ)/osarch_con_from_example.con
$(KRN_CON_CONTRIBUTORS) .ELIF $(TARGET_ARCH) == arm4t1e
OSARM_CON_CONTRIBUTORS != $(OBJ)/osarch_con_from_example.con
$(OSARM_CON_CONTRIBUTORS) .ELIF $(TARGET_ARCH) == arm4t1be
OSARM_CON_CONTRIBUTORS != $(OBJ)/osarch_con_from_example.con
$(OSARM_CON_CONTRIBUTORS) .ELIF $(TARGET_ARCH) == sarm1e
OSARM_CON_CONTRIBUTORS != $(OBJ)/osarch_con_from_example.con
$(OSARM_CON_CONTRIBUTORS) .ELIF $(TARGET_ARCH) == sarm1be
OSARM_CON_CONTRIBUTORS != $(OBJ)/osarch_con_from_example.con
$(OSARM_CON_CONTRIBUTORS) .END $(OBJ)/osarch_con_from_example.con .PRECIOUS:

```

```
$(MAKEFILE:s\-\f\\) $(USERCONF) $(ECHO) Create: @$$(ECHOEND) $(ECHOEMPTY) >$$@
##### # # Contribution to osemmain.con #
##### OSEMAIN_CON_CONTRIBUTORS
!:= $(OBJ)/osemain_con_from_example.con $(OSEMAIN_CON_CONTRIBUTORS)
$(OBJ)/osemain_con_from_example.con.PRECIOUS: $(MAKEFILE:s\-\f\\) $(USERCONF)
$(ECHOEMPTY)>$$@ $(ECHO)/* The entries below are added by makefile.mk */$(ECHOEND)>>$$@
$(ECHO)/* They represent the parameters for the application. */$(ECHOEND)>>$$@ .IF
$(EXECUTABLE_FILE_TYPE) != load_module .IF $(INCLUDE_OSE_EFS) == yes
$(ECHO)PRI_PROC(start_efs, start_efs, 1023, 9, DEFAULT, 0, NULL)$(ECHOEND)>>$$@ .ELIF
$(INCLUDE_OSE_SHELL) == yes $(ECHO)PRI_PROC(start_shell, start_shell, 1023, 9, DEFAULT, 0,
NULL)$(ECHOEND)>>$$@ .END .IF $(INCLUDE_OSE_INET) == yes $(ECHO)PRI_PROC(init_inet,
init_inet, 256, 9, DEFAULT, 0, NULL)$(ECHOEND)>>$$@ .END .END .IF $(INCLUDE_OSE_PRH) ==
yes $(ECHO)PRI_PROC(start_prh, start_prh, 256, 10, DEFAULT, 0, NULL)$(ECHOEND)>>$$@ #
$(ECHO)START_OSE_HOOK2(start_prh_hook) $(ECHOEND)>>$$@ .END .IF $(TARGET_TYPE) ==
Executable $(ECHO)PRI_PROC($OMMainName, $OMMainName, 1000, 5, DEFAULT, 0, NULL)
$(ECHOEND)>>$$@ .END ##### # #
Contribution to softose.con % Softkernel environments #
##### .IF $(USE_OSEDEF_H) == yes
include $(EXAMPLES_COMMON_MAKE)/osedef.mk .END
##### # #
Inclusion of OSE products #
##### include
$(EXAMPLES_COMMON_MAKE)/products.mk # The COMPILERMAKE macro is assigned in
commonsetup.mk # This has to be done late since this makefile might check things like # USE_MMS and
such things that need to modify like CRT0 EXECUTABLE_NAME = $(TARGET_NAME) include
$(EXAMPLES_COMMON_MAKE)/compiler.mk LCFDEFINES +=
-DIMAGE_START=$(IMAGE_START) LCFDEFINES +=
-DIMAGE_MAX_LENGTH=$(IMAGE_MAX_LENGTH) include
$(EXAMPLES_COMMON_MAKE)/compilation_rules.mk $(eq,$(TARGET_TYPE),Library) .EXIT:
.IGNORE: ) include $(EXAMPLES_COMMON_MAKE)/targets.mk
##### #
IMPORT SHELL ENVIRONMENT
##### # Import
the environment variable PATH #.IMPORT: PATH
##### # END
OF MAKEFILE
#####
```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the CPP\_CG::<Environment>::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

(Default = Blank)

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## **ObjCleanCommand**

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default = \$(RM) \$OMFileObjPath*

## **ObjectName**

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = Empty string*

## **ObjectsDirectory**

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = Empty string*

## **ObjExtension**

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*Default = .o*

## **OSFileSystemCaseSensitive**

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Cleared*

## **ParseErrorDescript**

The ParseErrorDescript property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = ([^()]+)([()([0-9]+)]\*)[:] (error|warning|fatal error) (.\*)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = ([^()]+)([()([0-9]+)]\*)[:] (error|warning|fatal error)*

## ParseSeverityError

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^()]+)([()([0-9]+)]\*)[:] (error|fatal error)*

## ParseSeverityWarning

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^()]+)([()([0-9]+)]\*)[:] (warning)*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Cleared*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running Rational Rhapsody, whereas the target is the machine running the application. To run remotely, the UseRemoteHost property must be set to True. If UseRemoteHost is True and RemoteHost is blank, the current host name is used for the remote host. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = Empty string*

## ReusableStatechartSwitches

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts. See the upgrade history on the support site for information on this change.

*Default = -DOM\_REUSABLE\_STATECHART\_IMPLEMENTATION*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*Default = .h*

## UnixPathNameForOMROOT

The UnixPathNameForOMROOT property specifies whether the makefile must include UNIXstyle path names.

The pRISM compilers do not tolerate DOS-style path conventions. If you do not set this property correctly, there might be many compilation problems.

Default value = Cleared

## UpdateBuildSettingsInIDE

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## UseNewBuildOutputWindow

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the CG::General::ShowLogViewAfterBuild property to True.

*Default = Checked*

## UseRemoteHost

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Checked*

## UseTemplateName

In the declarations for members of a template class, some compilers require that you use precede the member type with the keyword "typename" for types with dependent names. Since support for this keyword varies between compilers, the UseTemplateName property is used to specify whether the "typename" keyword should be included in the generated code.

*Default = Cleared*

## OSPL

The OSPL metaclass contains properties that are relevant for projects using the OpenSplice implementation of DDS.

## AnnotationLocation

For DDS projects, the property AnnotationLocation is used to specify where code should be generated for the supported DDS annotation types. The possible values are BEFORE\_STATEMENT, AFTER\_STATEMENT, and AFTER\_TYPE.

In order to have an effect on the generated code, this property must be modified for specific Configurations. It will not affect the generated code if you change the value at the package level.

*Default = AFTER\_TYPE*

## Package

The Package metaclass contains properties that affect packages.

## Animate

The Animate property specifies whether animation code is generated for an element. You can specify your own animation function by using the CPP\_CG::Type::AnimSerializeOperation property. The semantics of the Animate property is always in favor of the owner settings:

- If a package Animate property is set to Cleared, all the classes owned by the package are not animated, regardless of the class Animate settings.
- If a class Animate property is set to Cleared, all the elements in the class (attributes, operations, relations, and so on) are not animated.
- If an operation Animate property is set to Cleared, all the arguments are not animated.
- If the AnimateArguments property is set to Cleard, all the arguments are not animated, regardless of the specific argument Animate property settings.

*Default = Checked*

## ContributesToNamespace

The ContributesToNamespace property specifies whether the packages contained in this package is declared as children packages of this package. Regardless of the setting, a directory is created for the current package to hold its contained elements.

*Default = True*

## DefineNameSpace

The DefineNameSpace property specifies whether a package defines a namespace. A namespace is a declarative region that attaches an additional identifier to any names declared inside it.

*Default = Cleared*

## DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- \$Name - the element name
- \$FullName - the full path of the element (P1::P2::C.a)
- \$Description - the text provided on the Description tab of the Features window for the element
- \$Requirements - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model



elements in the configuration. If you elect to turn off this option, you can use the \$Requirements keyword in the DescriptionTemplate property to get such comments generated for specific model elements.

- \$Login - the login name of the user who generated the file
- \$CodeGeneratedDate - the date on which the code was generated
- \$CodeGeneratedTime - the time at which the code was generated
- \$RhapsodyVersion - the version of Rhapsody that was used to generate the file
- For operations: \$Signature - the operation signature, \$Arguments - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: \$Type - the type of the attribute
- For arguments: \$Type - the argument type, \$Direction - the argument direction (In, Out, or InOut)
- For events: \$Arguments - the text provided on the Description tab of the Features window for the event arguments
- For association ends: \$Target - the other end of the association
- For requirements: \$ID - the ID specified for the requirement, \$Specification - the specification text for the requirement
- The values of tags can be included by using the syntax \$tag\_name. For example, if you use a tag called author for your classes, you can use the keyword \$author in the value of DescriptionTemplate in order to have the value of the author tag included in the element description.
- The values of properties can be included by using the syntax \$property\_name. For example, you can use the keyword \$Animate in the value of DescriptionTemplate in order to have the value of the Animate property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as \$Name)
- Tag keywords
- Property keywords

If the value of a keyword is a MultiLine value, then each new line (except the first one) starts with the value of the <lang>\_CG::Configuration::DescriptionBeginLine property, and each line ends with the value of the <lang>\_CG::Configuration::DescriptionEndLine property.

*Default = Blank*

## EventsBaseID

All events are assigned an ID number that is used when code is generated.

If you would like the numbering of events in a package to start at a number different than the default start number used by Rational Rhapsody, you can use the EventsBaseID property to specify your own start number.

*Default = 1*



## GenerateDirectory

The GenerateDirectory property specifies whether to generate a separate directory for the package.

The possible values are as follows:

- Checked - The package generates a directory.
- Cleared - The package does not generate a directory. (This is the default.)

GenerateDirectory has an immediate effect on directory generation.

## IDLFileName

When code is generated for a DDS model, a separate .idl file is generated for each package that contains DDS elements such as topicStructs.

You can customize the name of the generated .idl file by modifying the value of the property IDLFileName.

*Default = \$(PackageName)\_IDL*

## ImplIncludes

The ImplIncludes property specifies the names (including full paths) of header files to be included at the top of implementation files generated for classes, objects or object types, or packages. Separate multiple file names by using commas, without spaces.

*Default = Empty string*

## ImplementationEpilog

The ImplementationEpilog property enables you to add any code that you want to be added as verbatim text (to be ignored by Rational Rhapsody) to the end of the definition of a model element. For example, you could wrap a section of code with an #ifdef-#endif pair, add compiler-specific keywords, or add a #pragma statement. For example, to specify that an operation is available only when the code is compiled with \_DEBUG, set the following properties for the operation:

- Set SpecificationProlog to #ifdef \_DEBUG cr.
- Set SpecificationEpilog to #endif.
- Set ImplementationProlog to #ifdef \_DEBUG cr.
- Set ImplementationEpilog to #endif.

The following table lists whether leading and trailing linefeeds are generated.

Metaclass	Leading	Linefeed	Added?	Class	Yes	Package	No
-----------	---------	----------	--------	-------	-----	---------	----

*Default = Empty MultiLine*

## ImplementationPragmas

The ImplementationPragmas property specifies the user-defined pragmas to generate in the body.

*Default = Empty MultiLine*

## ImplementationPragmasInContextClause

The ImplementationPragmasInContextClause property specifies the user-defined pragmas to generate in the context clause of the body.

*Default = Empty MultiLine*

## ImplementationProlog

The ImplementationProlog property enables you to add any code that you want to be added as verbatim text (to be ignored by the software) to the beginning of the definition of a model element. For example, you could wrap a section of code with an #ifdef-#endif pair, add compiler-specific keywords, or add a #pragma statement. For example, to specify that an operation is available only when the code is compiled with \_DEBUG, set the following properties for the operation:

- Set SpecificationProlog to #ifdef \_DEBUG cr.
- Set SpecificationEpilog to #endif.
- Set ImplementationProlog to #ifdef \_DEBUG cr.
- Set ImplementationEpilog to #endif.

The following table lists whether leading and trailing linefeeds are generated.

Metaclass	Trailing Linefeed	Added?	Class	No Package	Yes
-----------	-------------------	--------	-------	------------	-----

*Default = Empty MultiLine*

## InitializationCode

The InitializationCode property specifies the user-defined initialization code to add to the package body.  
(empty MultiLine)

## IsNested

The IsNested property specifies whether to generate the class or package as nested.

*Default = False*

## IsPrivate

The `IsPrivate` property specifies whether to generate the class or package as private.

*Default = False*

## MarkPrologEpilogInAnnotations

The `MarkPrologEpilogInAnnotations` property specifies whether to generate ignore annotations for the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties so they are ignored during roundtrip. When you insert code element declarations (variables, types, functions, and so on) in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, after a full roundtrip those elements are added to the model and are duplicated on the next code generation. When you use the `MarkPrologEpilogInAnnotations` property, you can have Rational Rhapsody automatically ignore the information specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties instead of adding the ignore annotations manually. The possible values for the `MarkPrologEpilogInAnnotations` property are:

- **None** - Rational Rhapsody does not generate any annotations. Any models created before Version 4.1 automatically have this property setting.
- **Ignore** - Rational Rhapsody generates the `///  
//#]` annotation after the code specified in those properties.
- **Auto** - If the code in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties is one line (it does not contain any newline characters (`\n`)), no annotations are generated (the same behavior has the **None** setting). If there is more than one line, Rational Rhapsody generates the `///  
//#]` annotation after the code specified in those properties (the same behavior as the **Ignore** setting).

During roundtrip, any ignore annotations in the comments of the element are not included in its description. Because the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties are generated between the annotation and its declaration for the element, you cannot rename those elements on roundtrip. If you change the name of an element, it is removed from the model and added with the new name. Some model information (for example, property settings) might be lost.

*Default = None*

## NamespaceName

By default, if you have set the `CPP_CG::Package::DefineNamespace` property to `True`, the name used for the namespace is the name of the package. The `NamespaceName` property allows you to specify a different name to use for the namespace.

*Default = Blank*

## NestingVisibility

The `NestingVisibility` property specifies the visibility of the generated specification of the nested class or

package.

*Default = Public*

## OpeningBraceStyle

The OpeningBraceStyle property controls where the opening brace of the code block is positioned - on the same line as the element name (SameLine) or on the following line (NewLine).

## PackageClassNamePolicy

The PackageClassNamePolicy property specifies the naming policy for classes generated by Rational Rhapsody. Rhapsody generates a class for each package in the Rational Rhapsody Developer for Java model. The possible values are as follows:

- Default - Use the default naming style (the package class name is the same as the package name).
- WithSuffix - Add a suffix to the class name. The suffix is “\_pkgClass”.

*Default = Default*

## PackageEventIdRange

The PackageEventIdRange property specifies the maximum number of events allowed in a package. This property is set on the component level.

*Default = 200*

## Renames

The Renames property enables one element to rename another element of the same type. You can also rename an element by using a renames dependency. In the case of a conflict, the dependency has precedence. Note the following:

- For attributes, this property works only for static attributes in a class or for attributes in a package.
- For operations, this property contains the name of the operation this operation in renaming. The signatures of the two operations must match.

*Default = Empty string*

## Simplify

If you are using the Rational Rhapsody customizable code generation mechanism, the Simplify property can be used to change the way specific types of elements are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - The element is ignored.
- Copy - The element is copied from the original to the simplified model. It is not modified in any way.
- Default - Uses the standard simplification for this item, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after Rational Rhapsody standard simplification for the element has been applied.

*Default = "Default"*

## SpecificationEpilog

The SpecificationEpilog property allows you to add code to the end of the declaration of a model element.

For example, you can use this property to add an `#ifdef` to indicate that an operation is available only when the code is compiled with `_DEBUG`, by setting the following properties:

- For SpecificationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For SpecificationEpilog, enter the value `#endif`
- For ImplementationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For ImplementationEpilog, enter the value `#endif`

*Default = Blank*

## SpecificationPragmas

The SpecificationPragmas property specifies the user-defined pragmas to generate in the specification.

*Default = Empty MultiLine*

## SpecificationPragmasInContextClause

The SpecificationPragmasInContextClause property specifies the user-defined pragmas to generate in the context clause of the specification.

*Default = Empty MultiLine*

## SpecificationProlog

The SpecificationProlog property allows you to add code to the beginning of the declaration of a model element.

For example, you can use this property to add an `#ifdef` to indicate that an operation is available only when

the code is compiled with `_DEBUG`, by setting the following properties:

- For SpecificationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For SpecificationEpilog, enter the value `#endif`
- For ImplementationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For ImplementationEpilog, enter the value `#endif`

*Default = Blank*

## SpecIncludes

The SpecIncludes property specifies the names (including full paths) of header files to be included at the top of specification files generated for classes (C++ and Java), objects or object types (C), and packages. Separate multiple file names by using commas, without spaces.

*Default = Empty string*

## Port

The Port metaclass controls whether code is generated for ports.

## DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- `$Name` - the element name
- `$FullName` - the full path of the element (P1::P2::C.a)
- `$Description` - the text provided on the Description tab of the Features window for the element
- `$Requirements` - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the `$Requirements` keyword in the DescriptionTemplate property to get such comments generated for specific model

elements.

- \$Login - the login name of the user who generated the file
- \$CodeGeneratedDate - the date on which the code was generated
- \$CodeGeneratedTime - the time at which the code was generated
- \$RhapsodyVersion - the version of Rhapsody that was used to generate the file
- For operations: \$Signature - the operation signature, \$Arguments - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: \$Type - the type of the attribute
- For arguments: \$Type - the argument type, \$Direction - the argument direction (In, Out, or InOut)
- For events: \$Arguments - the text provided on the Description tab of the Features window for the event arguments
- For association ends: \$Target - the other end of the association
- For requirements: \$ID - the ID specified for the requirement, \$Specification - the specification text for the requirement
- The values of tags can be included by using the syntax \$tag\_name. For example, if you use a tag called author for your classes, you can use the keyword \$author in the value of DescriptionTemplate in order to have the value of the author tag included in the element description.
- The values of properties can be included by using the syntax \$property\_name. For example, you can use the keyword \$Animate in the value of DescriptionTemplate in order to have the value of the Animate property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as \$Name)
- Tag keywords
- Property keywords

If the value of a keyword is a MultiLine value, then each new line (except the first one) starts with the value of the <lang>\_CG::Configuration::DescriptionBeginLine property, and each line ends with the value of the <lang>\_CG::Configuration::DescriptionEndLine property.

*Default = Blank*

## Generate

The Generate property specifies whether to generate code for a particular type of element.

*Default = Checked*

## HandleDisconnectedPort

The HandleDisconnectedPort property allows you to provide a code fragment that handles cases where the link of a port is not initialized.

The following is an example of such code:

```
short eventID = $RuntimeEventID;

if (eventID == -1)

{

    cout << "Warning: Operation $OpName was not sent by way of port $PortOwnerName:$PortName since
    link by way of interface $InterfaceName is not initialized\n" << endl;

}

else

{

    cout << "Warning: Event ID " << eventID << " was not sent by way of port $PortOwnerName:$PortName
    since link by way of interface $InterfaceName is not initialized\n" << endl;

}
```

The following keywords can be used in your code:

**\$OpName** - the name of the primitive operation being called by way of the port

**\$RuntimeEventID** - the event ID of the event that was sent to the port. In the case of primitive operations, -1 is used as the value of this keyword.

**\$PortName** - the name of the port

**\$PortOwnerName** - the name of the class or object that owns the port

**\$InterfaceName** - the interface that declared the service requested by way of the port

Note that this property is available only for explicit (non-rapid) ports, and only for C++.

*Default = Empty*

## HandleUnknownEvent

The HandleUnknownEvent property allows you to provide a code fragment that handles cases where an unknown event is sent by way of the port.

The following is an example of such code:

```
cout <<"Event ID " << $RuntimeEventID << " is not recognized in port $PortOwnerName:$PortName"
<< endl;
```

The following keywords can be used in your code:



\$OpName - the name of the primitive operation being called by way of the port

\$RuntimeEventID - the event ID of the event that was sent to the port. In the case of primitive operations, -1 is used as the value of this keyword.

\$PortName - the name of the port

\$PortOwnerName - the name of the class or object that owns the port

\$InterfaceName - the interface that declared the service requested by way of the port

Note that this property is available only for explicit (non-rapid) ports, and only for C++.

*Default = Empty*

## OptimizeCode

Code generation for ports and flow ports was optimized in version 7.5.3 of Rhapsody, relative to the code generated in previous versions. A new property named OptimizeCode was added with a default value of True. In the C++ backward compatibility profile for 7.5.3., the value for this property is set to False so that the old code generation mechanism will be used for ports and flow ports in older models.

## Simplify

If you are using the Rational Rhapsody customizable code generation mechanism, the Simplify property can be used to change the way specific types of elements are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - The element is ignored.
- Copy - The element is copied from the original to the simplified model. It is not modified in any way.
- Default - Uses the standard simplification for this item, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the standard simplification in the product for the element has been applied.

*Default = "Default"*

## SupportMulticast

Use this property to control the multicasting ability of a port. Use the property to enable data or events to be sent from one sender port to many.

The possible values are:

- Always - Rational Rhapsody always generates multicasting ability to each port in the model.

- Smart - Rational Rhapsody identifies ports that are connected to more than 1 port and generates code to support multicasting to those ports only.
- Never - Rational Rhapsody never generates code supporting multicasting of data/event through ports. This is the value for models created before Rational Rhapsody 7.5, so the old behavior will be the same.

*Default = Smart*

## UseExactTypeForReqPureReactiveInterface

The UseExactTypeForReqPureReactiveInterface property determines whether a port that only has pure reactive required interfaces (interfaces that only have event receptions) can be connected to a rapid port (port with no explicit contract - relays any kind of event) or only to a port that provides the required interface. The possible values are:

- False - The port can be connected to a rapid port.
- True - The port can only be connected to a port that provides the required interface.

*Default = False*

Note that this property is only relevant for ports that only have pure reactive required interfaces.

## UsePre82PortsCG

Prior to release 8.2, there were situations where Proxy Port internal links were not initialized correctly. This issue was corrected in 8.2. In order to preserve the previous code generation behavior for pre-8.2 models, the property CPP\_CG::Port::UsePre82PortsCG was added to the backward compatibility settings for C++, with a value of True.

## QNXNeutrinoMomentics

The QNXNeutrinoMomentics metaclass contains environment settings (Compiler, framework libraries, and so on) for QNX Neutrino RTOS compiled for X86, by using the GCC cross compiler.

## AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.

- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)/LangCpp/osconfig/QNX*

## **AdditionalReservedWords**

The AdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = Empty string*

## **BuildArgumentsInIDE**

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the CPP\_CG:[environment]:BuildCommandInIDE property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## **BuildCommandInIDE**

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## **BuildCommandSet**

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration.

To change this property, use the Configuration window in the browser - do not change it by using the Properties tab in the Features window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link. The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*Default = Debug*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\"\\etc\\qnxcwmake.bat qnxcwbuild.mak build \"CPU=\$CPU\" \"CPU\_SUFFIX=\$CPU\_SUFFIX\" \"*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## CompileSwitches

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property.

*Default = -lang-c++ -I. -I\$OMDefaultSpecificationDirectory -I\$(OMROOT) -I\$(OMROOT)/LangCpp -I\$(OMROOT)/LangCpp/oxf \$(INST\_FLAGS) \$(INCLUDE\_PATH) \$(INST\_INCLUDES) -DUSE\_Iostream -DOM\_USE\_NOthrow\_GEN \$OMCPCPPCompileCommandSet -c*

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The default is as follows:

```
@echo Compiling $OMFileImpPath $(CREATE_OBJ_DIR) @$(CC) $OMFileCPPCompileSwitches -o
$OMFileObjPath $OMFileImpPath
```

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

The default value is as follows: -g

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*Default = -O*

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

The default is as follows:

```
$OMFileObjPath : $OMFileImpPath $OMFileSpecPath $OMFileDependencies
```

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: \$OMFileObjPath : \$OMFileImpPath "\*.bmp" "\*.ico" "\*.rc2"

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction with an IDE such as Eclipse, the EnableDebugIntegrationWithIDE property can be used to specify whether or not the IDE debugger should be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to True, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's `exit()` function is called automatically. The system function calls the destructors of framework instances in the order it determines. The `EndApplicationCode` property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an `endApplication()` function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to `endApplication()`, it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The `EntryPoint` property specifies the name of the main program for a given environment.

*Default = main*

If applicable for the metaclass, see also the definition of the `EntryPointDeclarationModifier` property for more information.

## ErrorMessageTokensFormat

The `ErrorMessageTokensFormat`, working with the `ParseErrorMessage` property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. `ErrorMessageTokensFormat` defines the number and location of tokens within the regular expression defined by the `ParseErrorMessage` property. `ErrorMessageTokens` has three parameters, each with an integer value:

- `TotalNumberOfTokens` - The number of tokens in the regular expression
- `FileTokenPosition` - The position of the file name token in the expression
- `LineTokenPosition` - The position of the line number token in the expression

*Default = ToTalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2*

## ExeExtension

The `ExeExtension` property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the `CPP_CG::<Environment>::ExeName` property plus the value of this property.

*(Default = Blank)*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the CPP\_CG::<Environment>::ExeExtension property.

*(Default = Blank)*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements. The file inclusions are generated in the makefile.

*Default = \$OMSpecIncludeInElements \$OMImpIncludeInElements*

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile. Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

*Default =*

*\$(OMROOT)\LangCpp\lib\QNXCWWebComponents\$(CPU)\$(CPU\_SUFFIX)\$(LIB\_EXT),  
\$(OMROOT)\lib\QNXCWWebServices\$(CPU)\$(CPU\_SUFFIX)\$(LIB\_EXT), -lsocket*

## HasIDEInterface

The HasIDEInterface property is a Boolean value that specifies whether IDE support is enabled. If IDE support is enabled (Checked), the IDEInterfaceDLL property points to an IDE adapter that provides connection to the IDE. If the property is set to Cleared, IDE support is disabled and IDE services are not attempted. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = Cleared*

## IDEInterfaceDLL

The IDEInterfaceDLL property is a string that points to the IDE adapter DLL. You should not have any reason to modify this property. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = Empty string*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment. The default values are as follows:

*Default = .cpp*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*Default = include*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT/etc/vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT/etc/qnxcwmake.bat\" \$makefile \$maketarget"*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*



## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .a*

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

*Default = -g*

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

*Default = -O*

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

*Default = -static*

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the CPP\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*(Default = .mak)*

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags

- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration.

```
Default = ##### Target type (Debug/Release) #####
#####
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease
ConfigurationCPPCompileSwitches=$OMReusableStatechartSwitches
$OMConfigurationCPPCompileSwitches #####
Predefined macros ##### RM=rm -rf MD=mkdir -p INCLUDE_QUALIFIER=-I
CPU=$OMCPU CPU_SUFFIX=$OMCPU_SUFFIX CC=qcc -Vgcc_nto$(CPU)$(CPU_SUFFIX)
-I$(QNX_TARGET)/usr/include -lang-c++ -DUSE_Iostream
LIB_CMD=$(QNX_HOST)/usr/gcc/nto$(CPU)/bin/ar LINK_CMD=$(CC) LIB_FLAGS=rvu
LINK_FLAGS=-static ##### Context macros
##### $OMContextMacros #####
Predefined macros ##### $(OBJS) : $(INST_LIBS) $(OXF_LIBS)
OBJ_DIR=$OMObjectsDir ifeq ($(OBJ_DIR),) CREATE_OBJ_DIR= CLEAN_OBJ_DIR= else
CREATE_OBJ_DIR= $(MD) $(OBJ_DIR) CLEAN_OBJ_DIR= $(RM) $(OBJ_DIR) endif ifeq
$(INSTRUMENTATION),Animation) INST_FLAGS=-DOMANIMATOR
INST_INCLUDES=-I$(OMROOT)/LangCpp/aom -I$(OMROOT)/LangCpp/tom INST_LIBS=
$(OMROOT)/LangCpp/lib/QNXCWaoanim$(CPU)$(CPU_SUFFIX)$(LIB_EXT)
OXF_LIBS=$(OMROOT)/LangCpp/lib/QNXCWoxfinst$(CPU)$(CPU_SUFFIX)$(LIB_EXT)
$(OMROOT)/LangCpp/lib/QNXCWomcomappl$(CPU)$(CPU_SUFFIX)$(LIB_EXT)
SOCK_LIB=-lsocket else ifeq $(INSTRUMENTATION),Tracing) INST_FLAGS=-DOMTRACER
INST_INCLUDES=-I$(OMROOT)/LangCpp/aom -I$(OMROOT)/LangCpp/tom
INST_LIBS=$(OMROOT)/LangCpp/lib/QNXCWtomtrace$(CPU)$(CPU_SUFFIX)$(LIB_EXT)
$(OMROOT)/LangCpp/lib/QNXCWaoTRACE$(CPU)$(CPU_SUFFIX)$(LIB_EXT) OXF_LIBS=
$(OMROOT)/LangCpp/lib/QNXCWoxfinst$(CPU)$(CPU_SUFFIX)$(LIB_EXT)
$(OMROOT)/LangCpp/lib/QNXCWomcomappl$(CPU)$(CPU_SUFFIX)$(LIB_EXT)
SOCK_LIB=-lsocket else ifeq $(INSTRUMENTATION),None) INST_FLAGS= INST_INCLUDES=
INST_LIBS= OXF_LIBS=$(OMROOT)/LangCpp/lib/QNXCWoxf$(CPU)$(CPU_SUFFIX)$(LIB_EXT)
SOCK_LIB= else @echo An invalid Instrumentation $(INSTRUMENTATION) is specified. exit endif endif
endif .SUFFIXES: $(CPP_EXT)
#####
##### Context dependencies and commands #####
$OMContextDependencies $OMFileObjPath : $OMMainImplementationFile $(OBJS) @$(CC)
$(ConfigurationCPPCompileSwitches) -o $OMFileObjPath $OMMainImplementationFile
##### #
Predefined linking instructions # # INST_LIBS is included twice to solve bi-directional dependency
between libraries #
#####
$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS) $OMFileObjPath
$OMMakefileName $OMModelLibs @echo Linking $(TARGET_NAME)$(EXE_EXT) @$(LINK_CMD)
$OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \ $(OXF_LIBS) \
$(INST_LIBS) \ $(SOCK_LIB) \ $(LINK_FLAGS) -o $(TARGET_NAME)$(EXE_EXT)
```

```
$(TARGET_NAME)$(LIB_EXT) : $(OBJS) $(ADDITIONAL_OBJS) $OMMakefileName @echo Building
library $@ @$$(LIB_CMD) $(LIB_FLAGS) $(TARGET_NAME)$(LIB_EXT) $(OBJS)
$(ADDITIONAL_OBJS) clean: @echo Cleanup $OMCleanOBJS $(RM) $OMFileObjPath
$(ADDITIONAL_OBJS) $(RM) $(TARGET_NAME)$(LIB_EXT) $(RM) $(TARGET_NAME)$(EXE_EXT)
$(CLEAN_OBJ_DIR)
```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the CPP\_CG::<Environment>::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## ObjCleanCommand

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default = \$(RM) \$OMFileObjPath*

## ObjectName

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = Empty string*

## ObjectsDirectory

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = Empty string*

## ObjExtension

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*Default = .o*

## OMCPU

The OMCPU property is resolved in the MakeFileContent property as the CPU type. The QNXNeutrinoCW environment uses the custom keywords feature to enable you to select the CPU without modifying the makefile template.

*Default = x86*

## OMCPU\_SUFFIX

The OMCPU\_SUFFIX property is resolved in the MakeFileContent property as the CPU extension (which is required for PPC targets). The QNXNeutrinoCW environment uses the custom keywords feature to enable you to select the CPU without modifying the makefile template.

*Default = (\$NO\_CPU\_EXT)*

## OSFileSystemCaseSensitive

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Checked*

## ParseErrorDescript

The ParseErrorDescript property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = ([^:]+)[:](/[0-9]+)[:](.\*)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning). Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = ([^:]+)[:]?([0-9]+)[:]?]*

## ParseMakeError

The ParseMakeError property is used to define a regular expression that represents the format of make process or linker error messages. This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = (make)(.\*)(Error)*

## ParseSeverityError

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^:]+)[:]?([0-9]+)[:]?]*

## ParseSeverityWarning

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^:]+)[:]?([0-9]+)[:]? (warning)*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation.

*Default = /*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Cleared*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running Rational Rhapsody, whereas the target is the machine running the application. To run remotely, the UseRemoteHost property must be set to True. If UseRemoteHost is True and RemoteHost is blank, the current host name is used for the remote host. The

RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = Empty string*

## **ReusableStatechartSwitches**

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts. See the upgrade history on the support site for information on this change.

*Default = -DOM\_REUSABLE\_STATECHART\_IMPLEMENTATION*

## **SpecExtension**

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*Default = .h*

## **UnixLineTerminationStyle**

The UnixLineTerminationStyle property specifies whether generated files use the UNIX end-of-line style. If this property is set to Cleared, the end-of-line style depends on the host type (for example, DOS style on Windows machines, and UNIX style on Solaris machines).

*Default = Checked*

## **UnixPathNameForOMROOT**

The UnixPathNameForOMROOT property specifies whether the makefile must include UNIXstyle path names.

The pRISM compilers do not tolerate DOS-style path conventions. If you do not set this property correctly, there might be many compilation problems.

*Default = Checked*

## **UpdateBuildSettingsInIDE**

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## UseNewBuildOutputWindow

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the CG::General::ShowLogViewAfterBuild property to True.

*Default = Checked*

## UseNonZeroStdInputHandle

The UseNonZeroStdInputHandle property is a Boolean value that specifies whether to use a non-zero standard input handle.

*Default = Checked*

## UseRemoteHost

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Checked*

## UseShortPathNameForRoot

If your project uses the QNXNeutrinoMomentics environment, and the path to the Rhapsody "Share" directory contains one or more spaces, then the compiler will have trouble locating files from this directory that are referenced in the makefile. The UseShortPathNameForRoot property is used to resolve this problem. When the value of the property is set to True, a DOS-style path is used for the "Share" directory in the makefile, rather than the full path.

*Default = True*

## UseTemplateName

In the declarations for members of a template class, some compilers require that you use precede the member type with the keyword "typename" for types with dependent names. Since support for this keyword varies between compilers, the UseTemplateName property is used to specify whether the "typename" keyword should be included in the generated code.

*Default = Cleared*

# QNXNeutrinoGCC

The QNXNeutrinoGCC metaclass contains environment settings (Compiler, framework libraries, and so on) for QNX Neutrino RTOS compiled for X86, by using the GCC on-target compiler.

## AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)/LangCpp/osconfig/QNX*

## AdditionalReservedWords

The AdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = Empty string*

## BuildArgumentsInIDE

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the CPP\_CG:[environment]:BuildCommandInIDE property is not an empty string.



Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandInIDE

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration.

To change this property, use the Configuration window in the browser - do not change it by using the Properties tab in the Features window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link. The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*Default = Debug*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## CompileSwitches

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property.

*Default = -I. -I\$OMDefaultSpecificationDirectory -I\$(OMROOT) -I\$(OMROOT)/LangCpp -I\$(OMROOT)/LangCpp/oxf \$(INST\_FLAGS) \$(INCLUDE\_PATH) \$(INST\_INCLUDES) -DUSE\_Iostream -DOM\_USE\_NOTHROW\_GEN \$OMCPPCompileCommandSet -c*

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The default is as follows:

```
@echo Compiling $OMFileImpPath $(CREATE_OBJ_DIR) @$$(CC) $OMFileCPPCompileSwitches -o $OMFileObjPath $OMFileImpPath
```

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

The default value is as follows: -g

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*Default = -O*

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

The default is as follows:

```
$OMFileObjPath : $OMFileImpPath $OMFileSpecPath $OMFileDependencies
```

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: \$OMFileObjPath : \$OMFileImpPath "\*.bmp" "\*.ico" "\*.rc2"

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction with an IDE such as Eclipse, the `EnableDebugIntegrationWithIDE` property can be used to specify whether or not the IDE debugger should be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to `True`, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's `exit()` function is called automatically. The system function calls the destructors of framework instances in the order it determines. The `EndApplicationCode` property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an `endApplication()` function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to `endApplication()`, it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The `EntryPoint` property specifies the name of the main program for a given environment.

*Default = main*

If applicable for the metaclass, see also the definition of the `EntryPointDeclarationModifier` property for more information.

## ErrorMessageTokensFormat

The `ErrorMessageTokensFormat`, working with the `ParseErrorMessage` property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. `ErrorMessageTokensFormat` defines the number and location of tokens within the regular expression defined by the `ParseErrorMessage` property. `ErrorMessageTokens` has three parameters, each with an integer value:

- `TotalNumberOfTokens` - The number of tokens in the regular expression
- `FileTokenPosition` - The position of the file name token in the expression
- `LineTokenPosition` - The position of the line number token in the expression

*Default = TotalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2*

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the CPP\_CG::<Environment>::ExeName property plus the value of this property.

*(Default = Blank)*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the CPP\_CG::<Environment>::ExeExtension property.

*(Default = Blank)*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements. The file inclusions are generated in the makefile.

*Default = \$OMSpecIncludeInElements \$OMImpIncludeInElements*

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile. Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

*Default =*

`$(OMROOT)\LangCpp\lib\QNXWebComponents$(LIB_EXT),  
$(OMROOT)\lib\QNXWebServices$(LIB_EXT), -lsocket`

## HasIDEInterface

The HasIDEInterface property is a Boolean value that specifies whether IDE support is enabled. If IDE support is enabled (Checked), the IDEInterfaceDLL property points to an IDE adapter that provides

connection to the IDE. If the property is set to Cleared, IDE support is disabled and IDE services are not attempted. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = Cleared*

## IDEInterfaceDLL

The IDEInterfaceDLL property is a string that points to the IDE adapter DLL. You should not have any reason to modify this property. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = Empty string*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment. The default values are as follows:

*Default = .cpp*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*Default = include*

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*Default = \$executable*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

"\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\vx6make.bat\" \$makefile \$maketarget 6.2 \$BSP gnu"

*Default = Empty string*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .a*

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

*Default = -g*

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

*Default = -O*

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

*Default = \$OMLinkCommandSet*

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you

would like to use. Note that the first part of the filename can be customized by modifying the value of the CPP\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

(Default = .mak)

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration.

```
Default = ##### Target type (Debug/Release) #####
#####
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease
ConfigurationCPPCompileSwitches=$OMReusableStatechartSwitches
$OMConfigurationCPPCompileSwitches #####
Predefined macros ##### RM=/bin/rm -rf MD=/bin/mkdir -p INCLUDE_QUALIFIER=-I
CC=gcc -I/usr/include -DUSE_Iostream LIB_CMD=ar LINK_CMD=$(CC) LIB_FLAGS=rvu
LINK_FLAGS= /x86/lib/libm.so.1 -lstdc++ $OMConfigurationLinkSwitches
##### Context macros #####
$OMContextMacros ##### Predefined macros
##### $(OBJS) : $(INST_LIBS) $(OXF_LIBS) OBJ_DIR=$OMObjectsDir ifeq
$(OBJ_DIR,) CREATE_OBJ_DIR= CLEAN_OBJ_DIR= else CREATE_OBJ_DIR= $(MD) $(OBJ_DIR)
CLEAN_OBJ_DIR= $(RM) $(OBJ_DIR) endif ifeq $(INSTRUMENTATION),Animation)
INST_FLAGS=-DOMANIMATOR INST_INCLUDES=-I$(OMROOT)/LangCpp/aom
-I$(OMROOT)/LangCpp/tom INST_LIBS= $(OMROOT)/LangCpp/lib/QNXaomanim$(LIB_EXT)
OXF_LIBS=$(OMROOT)/LangCpp/lib/QNXoxfinst$(LIB_EXT)
$(OMROOT)/LangCpp/lib/QNXomcomappl$(LIB_EXT) SOCK_LIB=-lsocket else ifeq
$(INSTRUMENTATION),Tracing) INST_FLAGS=-DOMTRACER
INST_INCLUDES=-I$(OMROOT)/LangCpp/aom -I$(OMROOT)/LangCpp/tom
INST_LIBS=$(OMROOT)/LangCpp/lib/QNXtomtrace$(LIB_EXT)
$(OMROOT)/LangCpp/lib/QNXaomtrace$(LIB_EXT) OXF_LIBS=
$(OMROOT)/LangCpp/lib/QNXoxfinst$(LIB_EXT)
$(OMROOT)/LangCpp/lib/QNXomcomappl$(LIB_EXT) SOCK_LIB=-lsocket else ifeq
```

```

($(INSTRUMENTATION),None) INST_FLAGS= INST_INCLUDES= INST_LIBS=
OXF_LIBS=$(OMROOT)/LangCpp/lib/QNXoxf$(LIB_EXT) SOCK_LIB= else @echo An invalid
Instrumentation $(INSTRUMENTATION) is specified. exit endif endif endif .SUFFIXES: $(CPP_EXT)
#####
##### Context dependencies and commands #####
$OMContextDependencies $OMFileObjPath : $OMMainImplementationFile $(OBJS) @$(CC)
$(ConfigurationCPPCompileSwitches) -o $OMFileObjPath $OMMainImplementationFile
##### #
Predefined linking instructions # $INST_LIBS is included twice to solve bi-directional dependency
between libraries #
#####
$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS) $OMFileObjPath
$OMMakefileName $OMModelLibs @echo Linking $(TARGET_NAME)$(EXE_EXT) @$(LINK_CMD)
$OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \ $(OXF_LIBS) \
$(INST_LIBS) \ $(SOCK_LIB) \ $(LINK_FLAGS) -o $(TARGET_NAME)$(EXE_EXT)
$(TARGET_NAME)$(LIB_EXT) : $(OBJS) $(ADDITIONAL_OBJS) $OMMakefileName @echo Building
library @$@ @$(LIB_CMD) $(LIB_FLAGS) $(TARGET_NAME)$(LIB_EXT) $(OBJS)
$(ADDITIONAL_OBJS) clean: @echo Cleanup $OMCleanOBJS $(RM) $OMFileObjPath
$(ADDITIONAL_OBJS) $(RM) $(TARGET_NAME)$(LIB_EXT) $(RM) $(TARGET_NAME)$(EXE_EXT)
$(CLEAN_OBJ_DIR)

```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the CPP\_CG::<Environment>::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## ObjCleanCommand

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default = \$(RM) \$OMFileObjPath*

## ObjectName



The `ObjectName` property specifies an alternative name for the compiled object file in the generated makefile.

*Default = Empty string*

## ObjectsDirectory

The `ObjectsDirectory` property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = Empty string*

## ObjExtension

The `ObjExtension` property specifies the extension appended to compiled object files for a given environment.

*Default = .o*

## OSFileSystemCaseSensitive

The `OSFileSystemCaseSensitive` property specifies whether the OS file system for a given environment is case sensitive.

*Default = Checked*

## ParseErrorMessage

The `ParseErrorMessage` property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning) Along with the `ErrorMessageTokensFormat` property, `ParseErrorMessage` specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = ([^:]+):([0-9]+):[ ]*

## PathDelimiter

The `PathDelimiter` property specifies an alternative path separator for code generation.

*Default = /*

## QuoteOMROOT

The `QuoteOMROOT` property specifies whether to enclose the value of the OMROOT path variable in

double quotation marks in the generated makefile.

*Default = Checked*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running Rational Rhapsody, whereas the target is the machine running the application. To run remotely, the UseRemoteHost property must be set to True. If UseRemoteHost is True and RemoteHost is blank, the current host name is used for the remote host. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = Empty string*

## ReusableStatechartSwitches

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts. See the upgrade history on the support site for information on this change.

*Default = -DOM\_REUSABLE\_STATECHART\_IMPLEMENTATION*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*Default = .h*

## UnixLineTerminationStyle

The UnixLineTerminationStyle property specifies whether generated files use the UNIX end-of-line style. If this property is set to Cleared, the end-of-line style depends on the host type (for example, DOS style on Windows machines, and UNIX style on Solaris machines).

*Default = Checked*

## UnixPathNameForOMROOT

The UnixPathNameForOMROOT property specifies whether the makefile must include UNIXstyle path names.

The pRISM compilers do not tolerate DOS-style path conventions. If you do not set this property correctly, there might be many compilation problems.

*Default = Checked*

## UpdateBuildSettingsInIDE

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## UseNonZeroStdInputHandle

The UseNonZeroStdInputHandle property is a Boolean value that specifies whether to use a non-zero standard input handle.

*Default = Checked*

## UseRemoteHost

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Checked*

## UseTemplateTypename

In the declarations for members of a template class, some compilers require that you use precede the member type with the keyword "typename" for types with dependent names. Since support for this keyword varies between compilers, the UseTemplateTypename property is used to specify whether the "typename" keyword should be included in the generated code.

*Default = Cleared*

## Relation

The Relation metaclass contains properties that affect relations.

### Add

The Add property specifies the command used to add an item to a container.

*Default = Add\_\$target:c*

## AddGenerate

The AddGenerate property specifies whether to generate an Add() operation for relations.

*Default = True*

## Animate

The Animate property specifies whether animation code is generated for an element. You can specify your own animation function by using the CPP\_CG::Type::AnimSerializeOperation property. The semantics of the Animate property is always in favor of the owner settings:

- If a package Animate property is set to False, all the classes owned by the package are not animated, regardless of the class Animate settings.
- If a class Animate property is set to False, all the elements in the class (attributes, operations, relations, and so on) are not animated.
- If an operation Animate property is set to False, all the arguments are not animated.
- If the AnimateArguments property is set to False, all the arguments are not animated, regardless of the specific argument Animate property settings.

*Default = Checked*

## Clear

The Clear property specifies the name of an operation that removes all items from a relation.

*Default = Clear\_\$target:c*

## ClearGenerate

The ClearGenerate property specifies whether to generate a Clear() operation for relations.

*Default = True*

## CreateComponent

The CreateComponent property specifies the name of an operation that creates a new component in a composite class.

*Default = New\_\$target:c*

## CreateComponentGenerate

The CreateComponentGenerate property specifies whether to generate a CreateComponent operation for

composite objects. Setting this property to False is one way to optimize your code for size.

*Default = True*

## DataMemberVisibility

The DataMemberVisibility property specifies the visibility of the relation data member. For example, if the relation is implemented as a pointer, this property determines whether the pointer data member is declared as public, private, or protected. The default value for C++ is Protected.

## DeleteComponent

The DeleteComponent property specifies the name of an operation that deletes a component from a composite class.

*Default = Delete\_\$target:c*

## DeleteComponentGenerate

The DeleteComponentGenerate property specifies whether to generate a DeleteComponent() operation for composite objects.

*Default = True*

## DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- \$Name - the element name
- \$FullName - the full path of the element (P1::P2::C.a)
- \$Description - the text provided on the Description tab of the Features window for the element
- \$Requirements - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model

elements in the configuration. If you elect to turn off this option, you can use the \$Requirements keyword in the DescriptionTemplate property to get such comments generated for specific model elements.

- \$Login - the login name of the user who generated the file
- \$CodeGeneratedDate - the date on which the code was generated
- \$CodeGeneratedTime - the time at which the code was generated
- \$RhapsodyVersion - the version of Rhapsody that was used to generate the file
- For operations: \$Signature - the operation signature, \$Arguments - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: \$Type - the type of the attribute
- For arguments: \$Type - the argument type, \$Direction - the argument direction (In, Out, or InOut)
- For events: \$Arguments - the text provided on the Description tab of the Features window for the event arguments
- For association ends: \$Target - the other end of the association
- For requirements: \$ID - the ID specified for the requirement, \$Specification - the specification text for the requirement
- The values of tags can be included by using the syntax \$tag\_name. For example, if you use a tag called author for your classes, you can use the keyword \$author in the value of DescriptionTemplate in order to have the value of the author tag included in the element description.
- The values of properties can be included by using the syntax \$property\_name. For example, you can use the keyword \$Animate in the value of DescriptionTemplate in order to have the value of the Animate property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as \$Name)
- Tag keywords
- Property keywords

If the value of a keyword is a MultiLine value, then each new line (except the first one) starts with the value of the <lang>\_CG::Configuration::DescriptionBeginLine property, and each line ends with the value of the <lang>\_CG::Configuration::DescriptionEndLine property.

*Default = Blank*

## Find

The Find property specifies the name of an operation that locates an item among relational objects.

*Default = Find\_\$target:c*

## FindGenerate

The FindGenerate property specifies whether to generate a Find() operation for relations.

*Default = False*

## FullQualifiedTypeName

To specify that the the full qualified name should be generated when elements are used in contexts such as return types, set the value of the property FullQualifiedTypeName to True.

The value of this property can be set separately for attributes, relations, arguments, and operation return types (use CPP\_CG::Operation::FullQualifiedTypeName for return types).

The property can be set for specific contexts such as specific attributes or operations, or it can be set at the package or project level, for example, using the full qualified name for return types in a specific package.

*Default = False*

## Get

The Get property specifies the name of an operation that retrieves the relation currently pointed to by the iterator.

*Default = Get\_\$target:c*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation by using an index. The ContainerTypes::Relationtype::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the at() operation for the container to retrieve the item at the indexed position:  
\$cname-at(\$index)

*Default = get\$cname:cAt*

## GetAtGenerate

The GetAtGenerate property specifies whether to generate a getAt() operation for relations. The possible values are as follows:

- Checked - Generate a getAt() operation for relations.
- Cleared - Do not generate a getAt() operation for relations. Setting the GetAtGenerate property to False is one way to optimize your code for size.

*Default = Cleared*

## GetEnd

The `GetEnd` property specifies the name of an operation that points the iterator to the last item in a collection.

*Default = `Get_$target:cEnd`*

## GetEndGenerate

The `GetEndGenerate` property specifies whether to generate a `GetEnd()` operation for relations.

*Default = `True`*

## GetGenerate

The `GetGenerate` property specifies whether to generate accessor operations for relations.

*Default = `True`*

## GetKey

The `GetKey` property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key. For example, the following command retrieves an item based on the key name by using the subscript operator `[]`, which has been overloaded according to the STL definition for maps: `$cname-operator[]($keyName)`

*Default = `get$cname:c`*

## GetKeyGenerate

The `GetKeyGenerate` property specifies whether to generate `getKey()` operations for relations. Setting this property to `False` is one way to optimize your code for size.

*Default = `Checked`*

## ImplementationEpilog

The `ImplementationEpilog` property enables you to add any code that you want to be added as verbatim text (to be ignored by Rational Rhapsody) to the end of the definition of a model element. For example, you could wrap a section of code with an `#ifdef-#endif` pair, add compiler-specific keywords, or add a `#pragma` statement.

For example, to specify that an operation is available only when the code is compiled with `_DEBUG`, set the following properties for the operation:

- Set `SpecificationProlog` to `#ifdef _DEBUG cr.`
- Set `SpecificationEpilog` to `#endif.`
- Set `ImplementationProlog` to `#ifdef _DEBUG cr.`



- Set ImplementationEpilog to #endif.

The following table lists whether leading and trailing linefeeds are generated.

Metaclass Leading Linefeed Added? Class Yes Package No

*Default = Empty MultiLine*

## ImplementationProlog

The ImplementationProlog property enables you to add any code that you want to be added as verbatim text (to be ignored by the software) to the beginning of the definition of a model element.

For example, you could wrap a section of code with an #ifdef-#endif pair, add compiler-specific keywords, or add a #pragma statement. For example, to specify that an operation is available only when the code is compiled with \_DEBUG, set the following properties for the operation:

- Set SpecificationProlog to #ifdef \_DEBUG cr.
- Set SpecificationEpilog to #endif.
- Set ImplementationProlog to #ifdef \_DEBUG cr.
- Set ImplementationEpilog to #endif.

The following table lists whether leading and trailing linefeeds are generated.

Metaclass Trailing Linefeed Added? Class No Package Yes

*Default = Empty MultiLine*

## ImplementWithStaticArray

The ImplementWithStaticArray property specifies whether to implement relations as static arrays. The possible values are as follows:

- Default - Rational Rhapsody provides the appropriate implementation for all fixed and bounded relations.
- FixedAndBounded - All fixed and bounded relations are generated into static arrays.

To generate C-like code in C++ or Java, modify the value of the ImplementWithStaticArray property to FixedAndBounded.

*Default = FixedAndBounded*

## InitializationStyle

InitializationStyle controls the type of code that is generated if you have provided an initial value for a static association. The possible values for the property are:

- ByAssignment - assignment is used to set the initial value, for example, generator\* vehicle::itsGenerator = &myGen;

- ByInitializer - initializer is used to set the initial value, for example, generator  
vehicle::itsGenerator(myGen);
- ByAggregationKind - the type of code that is generated depends on the type of association: initializer is used for compositions, assignment is used for aggregations and ordinary associations.

*Default = ByAggregationKind*

## InitializeComposition

The InitializeComposition property controls how a composition relation is initialized. The possible values are as follows:

- InInitializer
- InRecordType
- None

*Default = InInitializer*

## InitialValue

Use the InitialValue property to set an initial value; for example, if you want to specify an initial value for a static association.

See also the `<lang>_CG::Relation:Static` property.

*Default = Empty string*

## Inline

The Inline property specifies how inline operations are generated. Which operations are affected by the Inline property depends on the metaclass:

- Attribute - Applies only to operations that handle attributes (such as accessors and mutators)
- Operation - Applies to all operations
- Relation - Applies only to operations that handle relations

Inlining in Rational Rhapsody Developer for C++ The possible values for the Inline property are as follows:

- none - The operation is not generated inline.
- in\_header - The operation is generated inline in the specification file.
- in\_source - The operation is generated inline in the implementation file.
- in\_declaration - A class operation is generated inline in the class declaration. A global function is generated inline in the package specification file.

Inlining an operation in the header might cause problems if the function body refers to other classes. For example, if the inlined code refers to another class (by way of a pointer such as itsRelatedClass), inlined code that is generated in a header might not compile. The implementation file for the class would have an

#include for RelatedClass, but the specification file would not. The workaround is to create a Usage dependency of the class with the inlined function on the related class. This forces an #include of the related class to be generated in the header of the dependent class with the inlined function.

*Default = none*

## IsAliased

The IsAliased property is a Boolean value that specifies whether attributes are aliased.

*Default = False*

## Kind

The Kind property specifies the kind of operation that should be generated for an element. The kind of operations that can be generated is language-dependent (for example, virtual and abstract exist only in C++ and Java). In Java, Kind can be defined only for attributes and operations, but not for relations. This property affects class operations, in addition to accessors and mutators for relations and attributes. The possible values are as follows:

- common - Class operations and accessor/mutator are non-virtual.
- virtual - Class operations and accessor/mutator are virtual. This type is valid for C++ and Java only.
- abstract - Class operations and accessor/mutator are pure virtual. This type is valid for C++ and Java only.

*Default = common*

## ObjectInitialization

The ObjectInitialization property specifies what kind of initialization occurs for the initial instances of a configuration. The possible values are as follows:

- Full - Instances are initialized and their behavior is started.
- Creation - Instances are initialized but their behavior is not started.
- None - Instances are not initialized and their behavior is not started.

*Default = Full*

## Remove

The Remove property specifies the name of an operation that removes an item from a relation.

*Default = Remove\_\$target:c*

## RemoveGenerate

The RemoveGenerate property specifies whether to generate a Remove() operation for relations.

*Default = True*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: \$cname-erase(\$keyName)

*Default = remove\$cname:c*

## RemoveKeyGenerate

The RemoveKeyGenerate property specifies whether to generate a removeKey() operation for qualified relations. Setting this property to Cleared is one way to optimize your code for size.

*Default = Checked*

## RemoveKeyHelpersGenerate

The RemoveKeyHelpersGenerate property controls the generation of the relation helper methods (for example, \_removeItsX() and \_\_removeItsX()). The possible values are as follows:

- True - Generate the helpers whenever code generation analysis determines that the methods are needed.
- False - Never generate the helpers.
- FromModifier - Generate the helpers based on the value of the CPP\_CG::Relation::RemoveKey property.

*Default = True*

## ScalarContainment

The ScalarContainment property determines how code will be generated for association ends that have been set to "Reference". If the value of the property is set to Reference, then the the reference (& ) sign is generated for the association rather than the pointer symbol.

Setting the property value to Reference also results in the following:

- Removes link initialization to NULL in class constructor
- Removes the call to the CleanUpRelation function in the class destructor, and also the creation of the function
- Updates all setter/getter code accordingly
- User needs to disable automatic generation of default constructors, and, if needed, add a default constructor manually

*Default = Pointer*

## **SafeInitScalar**

The SafeInitScalar property specifies whether to initialize scalar relations as null pointers.

*Default = False*

## **Set**

The Set property specifies the name of the mutator generated for scalar relations.

*Default = Set\_\$target:c*

## **SetGenerate**

The SetGenerate property specifies whether to generate mutators for relations.

*Default = True*

## **Simplify**

If you are using the Rational Rhapsody customizable code generation mechanism, the Simplify property can be used to change the way specific types of elements are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - The element is ignored.
- Copy - The element is copied from the original to the simplified model. It is not modified in any way.
- Default - Uses the standard simplification for this item, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for the element has been applied.

*Default = "Default"*

## **SpecificationEpilog**

The SpecificationEpilog property allows you to add code to the end of the declaration of a model element.

For example, you can use this property to add an `#ifdef` to indicate that an operation is available only when the code is compiled with `_DEBUG`, by setting the following properties:

- For SpecificationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For SpecificationEpilog, enter the value `#endif`
- For ImplementationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For ImplementationEpilog, enter the value `#endif`

*Default = Blank*

## SpecificationProlog

The SpecificationProlog property adds code to the beginning of the declaration of a model element (such as a configuration or class). For example, to create an abstract class in Java, you can set the SpecificationProlog property for the class to "abstract." You must include the space after the word "abstract." If the visibility for the class is set to default, the following class declaration is generated in the .java file: `abstract class classname { ... }` The SpecificationProlog property allows you to add compiler-specific keywords, add a `#pragma` statement, or wrap a section of code with an `#ifdef-#endif` pair. For example, to specify that an operation is available only when the code is compiled with `_DEBUG`, set the following properties for the operation:

- Set SpecificationProlog to `#ifdef _DEBUG cr.`
- Set SpecificationEpilog to `#endif.`
- Set ImplementationProlog to `#ifdef _DEBUG cr.`
- Set ImplementationEpilog to `#endif.`

The following table lists whether leading and trailing linefeeds are generated.

Metaclass	Leading Linefeed Added?	Trailing Linefeed Added?	Class	Yes	No	Package	Yes	Yes
(empty MultiLine)								

(empty MultiLine)

## Static

The Static property is a Boolean value that determines whether class-wide relations are enabled. Class-wide members of a class are shared between all instances of that class and are mapped as static. When a relation is tagged as static:

- The data member is generated as static (with the static keyword).
- The relation accessors are generated as static.
- The mutators of directional relations are generated as static. The mutators of symmetric relations are generated as common (non-static) operations.

Note the following behavior and restrictions:

- If there are links between instances based on static relations, code generation initializes all the valid links. In case of a limited relation size, the last initialization is preserved.
- When you generate instrumented code (animation or tracing), relation NOTIFY calls are not added to static relation mutators.
- Animation associates static relations with the class instances, not the class itself.
- In an instrumented application (animation or tracing), the static relations names appear in each instance

node; however, the values of directional static relations are not visible.

See also the `CG::Relation::Containment`, `Containertype::Relationtype::CreateStatic`, and `Containertype::Relationtype::InitStatic` properties.

*Default = Cleared*

## Requirement

The Requirement metaclass controls whether code is generated for requirements.

### DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- `$Name` - the element name
- `$FullName` - the full path of the element (P1::P2::C.a)
- `$Description` - the text provided on the Description tab of the Features window for the element
- `$Requirements` - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the `$Requirements` keyword in the DescriptionTemplate property to get such comments generated for specific model elements.
- `$Login` - the login name of the user who generated the file
- `$CodeGeneratedDate` - the date on which the code was generated
- `$CodeGeneratedTime` - the time at which the code was generated
- `$RhapsodyVersion` - the version of Rhapsody that was used to generate the file
- For operations: `$Signature` - the operation signature, `$Arguments` - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: `$Type` - the type of the attribute
- For arguments: `$Type` - the argument type, `$Direction` - the argument direction (In, Out, or InOut)

- For events: \$Arguments - the text provided on the Description tab of the Features window for the event arguments
- For association ends: \$Target - the other end of the association
- For requirements: \$ID - the ID specified for the requirement, \$Specification - the specification text for the requirement
- The values of tags can be included by using the syntax \$tag\_name. For example, if you use a tag called author for your classes, you can use the keyword \$author in the value of DescriptionTemplate in order to have the value of the author tag included in the element description.
- The values of properties can be included by using the syntax \$property\_name. For example, you can use the keyword \$Animate in the value of DescriptionTemplate in order to have the value of the Animate property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as \$Name)
- Tag keywords
- Property keywords

If the value of a keyword is a MultiLine value, then each new line (except the first one) starts with the value of the <lang>\_CG::Configuration::DescriptionBeginLine property, and each line ends with the value of the <lang>\_CG::Configuration::DescriptionEndLine property.

*Default = Realizes requirement \$Name[[ # \$ID]]*

## DescriptionTemplateForImplementation

If you are generating C or C++ code from your model, you can specify that requirements should be generated as comments in the implementation files, next to the operations that realize the requirements. There is also an option to have requirements generated as comments in both the specification and implementation files.

If you use one of these two options, you use the DescriptionTemplateForImplementation property to specify the text of the comments that are generated in the implementation files.

*Default = Realizes requirement \$Name[[ # \$ID]][[: \$Specification]]*

## Generate

The Generate property can be used to control whether comments are generated in the code to reflect the fulfillment of a requirement by specific model elements.

In general, the policy for generating requirement comments in code is controlled by the CG::Configuration::IncludeRequirementsAsComment property. The Generate property can be used to override the global policy for specific groups of requirements, for example high-level requirements.

You can set the value of the Generate property to False for a stereotype and then apply the stereotype to requirements that do not have to be reflected in the generated code.



Note that if `IncludeRequirementsAsComment` is set to `False`, comments will not be generated for requirements even if the `Generate` property is set to `True`.

*Default = True*

## RTI

The RTI metaclass contains properties that are relevant for projects using the RTI implementation of DDS.

### AnnotationLocation

For DDS projects, the property `AnnotationLocation` is used to specify where code should be generated for the supported DDS annotation types. The possible values are `BEFORE_STATEMENT`, `AFTER_STATEMENT`, and `AFTER_TYPE`.

In order to have an effect on the generated code, this property must be modified for specific Configurations. It will not affect the generated code if you change the value at the package level.

*Default = AFTER\_STATEMENT*

## Solaris2

The Solaris2 metaclass contains environment settings (Compiler, framework libraries, and so on) for Solaris 2, by using the Sun compiler.

### AdaptorSearchPath

The `AdaptorSearchPath` property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the `AdaptorSearchPath` property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)/LangCpp/osconfig/Solaris2*

## **AdditionalReservedWords**

The AdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = Empty string*

## **BuildArgumentsInIDE**

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the CPP\_CG:[environment]:BuildCommandInIDE property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## **BuildCommandInIDE**

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## **BuildCommandSet**

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration.

To change this property, use the Configuration window in the browser - do not change it by using the Properties tab in the Features window or by modifying the site.prp file. Note that this property also affects

the names of the framework libraries used in the link. The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*Default = Debug*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## CompileSwitches

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property.

*Default =*

`-I. -I$OMDefaultSpecificationDirectory -I$(OMROOT) -I$(OMROOT)/LangCpp  
-I$(OMROOT)/LangCpp/oxf $(INST_FLAGS) $(INCLUDE_PATH) $(INST_INCLUDES)  
-DUSE_Iostream $OMCPPCompileCommandSet -c`

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The default is as follows:

`@echo Compiling $OMFileImpPath $(CREATE_OBJ_DIR) @$(CC) $OMFileCPPCompileSwitches -o  
$OMFileObjPath $OMFileImpPath`

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a

debug version of the component.

*Default = -g*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*Default = -O*

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

The default is as follows:

```
$OMFileObjPath : $OMFileImpPath $OMFileSpecPath $OMFileDependencies
```

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: `$OMFileObjPath : $OMFileImpPath "*.bmp" "*.ico" "*.rc2"`

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction with an IDE such as Eclipse, the EnableDebugIntegrationWithIDE property can be used to specify whether or not the IDE debugger should be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to True, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's exit() function is called automatically. The system function calls the destructors of framework instances in the order it determines. The EndApplicationCode property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an endApplication() function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to endApplication(), it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The EntryPoint property specifies the name of the main program for a given environment.

Default = main

See also the definition of the EntryPointDeclarationModifier property for more information.

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens has three parameters, each with an integer value:

- TotalNumberOfTokens - The number of tokens in the regular expression
- FileTokenPosition - The position of the file name token in the expression
- LineTokenPosition - The position of the line number token in the expression

*Default = TotalNumberOfTokens=3,FileTokenPosition=1,LineTokenPosition=2*

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the CPP\_CG::<Environment>::ExeName property plus the value of this property.

*(Default = Blank)*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the CPP\_CG::<Environment>::ExeExtension property.

*(Default = Blank)*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements. The file inclusions are generated in the makefile.

*Default = \$OMSpecIncludeInElements \$OMImpIncludeInElements.*

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile. Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

The default is as follows:

```
$(OMROOT)\LangCpp\lib\sol2WebComponents$(LIB_EXT),  
$(OMROOT)\lib\sol2WebServices$(LIB_EXT), -lsocket -lnsl
```

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment. The default values are:

*Default = .cpp*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*Default = include*

## InvokeAnimatedExecutable

The AnimationPortRange value in the rhapsody.ini file makes it possible to use animation with multiple instances of Rhapsody running simultaneously.

Ordinarily, when you run an application with animation, the executable is launched using the command specified by the InvokeExecutable property. However, if you have specified a value greater than zero for AnimationPortRange in your rhapsody.ini file, the executable is launched using the command specified by the InvokeAnimatedExecutable property. By default, the value of InvokeAnimatedExecutable includes a reference to the port to use for animation.

*Default = xterm -e \$executable -port \$port*

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*Default = xterm -e \$executable*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

*Default = \$OMROOT/etc/sol2make \$makefile \$maketarget*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .a*

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

*Default = -g*

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

*Default = -O*

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

*Default = \$OMLinkCommandSet*

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the CPP\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*(Default = .mak)*

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration.

```
Default = ##### Target type (Debug/Release) #####  
#####  
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease
```



```

LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease
ConfigurationCPPCompileSwitches=$OMReusableStatechartSwitches
$OMConfigurationCPPCompileSwitches #####
Predefined macros ##### RM=/bin/rm -rf MD=/bin/mkdir -p INCLUDE_QUALIFIER=-I
CC=gcc -I/usr/include -DUSE_Iostream LIB_CMD=ar LINK_CMD=$(CC) LIB_FLAGS=rvu
LINK_FLAGS=-lposix4 -lpthread -lstdc++ $OMConfigurationLinkSwitches
##### Context macros #####
$OMContextMacros OBJ_DIR=$OMObjectsDir ifeq ($(OBJ_DIR),) CREATE_OBJ_DIR=
CLEAN_OBJ_DIR= else CREATE_OBJ_DIR= $(MD) $(OBJ_DIR) CLEAN_OBJ_DIR= $(RM)
$(OBJ_DIR) endif ##### Predefined macros
##### $(OBJS) : $(INST_LIBS) $(OXF_LIBS) ifeq ($(INSTRUMENTATION),Animation)
INST_FLAGS=-DOMANIMATOR INST_INCLUDES=-I$(OMROOT)/LangCpp/aom
-I$(OMROOT)/LangCpp/tom INST_LIBS= $(OMROOT)/LangCpp/lib/sol2aomanimGNU$(LIB_EXT)
OXF_LIBS=$(OMROOT)/LangCpp/lib/sol2oxfinstGNU$(LIB_EXT)
$(OMROOT)/LangCpp/lib/sol2omcomapplGNU$(LIB_EXT) SOCK_LIB=-lsocket -lnsl else ifeq
$(INSTRUMENTATION),Tracing INST_FLAGS=-DOMTRACER
INST_INCLUDES=-I$(OMROOT)/LangCpp/aom -I$(OMROOT)/LangCpp/tom
INST_LIBS=$(OMROOT)/LangCpp/lib/sol2tomtraceGNU$(LIB_EXT)
$(OMROOT)/LangCpp/lib/sol2aomtraceGNU$(LIB_EXT) OXF_LIBS=
$(OMROOT)/LangCpp/lib/sol2oxfinstGNU$(LIB_EXT)
$(OMROOT)/LangCpp/lib/sol2omcomapplGNU$(LIB_EXT) SOCK_LIB=-lsocket -lnsl else ifeq
$(INSTRUMENTATION),None INST_FLAGS= INST_INCLUDES= INST_LIBS=
OXF_LIBS=$(OMROOT)/LangCpp/lib/sol2oxfGNU$(LIB_EXT) SOCK_LIB= else @echo An invalid
Instrumentation $(INSTRUMENTATION) is specified. exit endif endif endif .SUFFIXES: $(CPP_EXT)
#####
##### Context dependencies and commands #####
$OMContextDependencies $OMFileObjPath : $OMMainImplementationFile $(OBJS) @$(CC)
$(ConfigurationCPPCompileSwitches) -o $OMFileObjPath $OMMainImplementationFile
##### #
Predefined linking instructions # # INST_LIBS is included twice to solve bi-directional dependency
between libraries #
#####
$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS) $OMFileObjPath
$OMMakefileName $OMModelLibs @echo Linking $(TARGET_NAME)$(EXE_EXT) @$(LINK_CMD)
$OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \ $(OXF_LIBS) \
$(INST_LIBS) \ $(SOCK_LIB) \ $(LINK_FLAGS) -o $(TARGET_NAME)$(EXE_EXT)
$(TARGET_NAME)$(LIB_EXT) : $(OBJS) $(ADDITIONAL_OBJS) $OMMakefileName @echo Building
library @@ @$(LIB_CMD) $(LIB_FLAGS) $(TARGET_NAME)$(LIB_EXT) $(OBJS)
$(ADDITIONAL_OBJS) clean: @echo Cleanup $OMCleanOBJS $(RM) $OMFileObjPath
$(ADDITIONAL_OBJS) $(RM) $(TARGET_NAME)$(LIB_EXT) $(RM) $(TARGET_NAME)$(EXE_EXT)
$(CLEAN_OBJ_DIR)

```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the CPP\_CG::<Environment>::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## **NullValue**

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## **ObjCleanCommand**

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default = \$(RM) \$OMFileObjPath*

## **ObjectName**

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = Empty string*

## **ObjectsDirectory**

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = Empty string*

## **ObjExtension**

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*Default = .o*

## **OSFileSystemCaseSensitive**

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Checked*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = ([^:]+)[:]?([0-9]+)[:]?*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation.

*Default = /*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Checked*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running Rational Rhapsody, whereas the target is the machine running the application. To run remotely, the UseRemoteHost property must be set to True. If UseRemoteHost is True and RemoteHost is blank, the current host name is used for the remote host. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = Empty string*

## ReusableStatechartSwitches

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts. See the upgrade history on the support site for information on this change.

*Default = -DOM\_REUSABLE\_STATECHART\_IMPLEMENTATION*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*Default = .h*

## **UnixLineTerminationStyle**

The UnixLineTerminationStyle property specifies whether generated files use the UNIX end-of-line style. If this property is set to Cleared, the end-of-line style depends on the host type (for example, DOS style on Windows machines, and UNIX style on Solaris machines).

*Default = Checked*

## **UnixPathNameForOMROOT**

The UnixPathNameForOMROOT property specifies whether the makefile must include UNIXstyle path names.

The pRISM compilers do not tolerate DOS-style path conventions. If you do not set this property correctly, there might be many compilation problems.

*Default = Checked*

## **UpdateBuildSettingsInIDE**

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## **UseRemoteHost**

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Checked*

## **UseTemplateName**

In the declarations for members of a template class, some compilers require that you use precede the member type with the keyword "typename" for types with dependent names. Since support for this keyword varies between compilers, the UseTemplateName property is used to specify whether the "typename" keyword should be included in the generated code.

*Default = Cleared*

## Solaris2GNU

The Solaris2GNU metaclass contains environment settings (Compiler, framework libraries, and so on) for Solaris 2, by using the GCC compiler .

### AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)/LangCpp/osconfig/Solaris2*

### AdditionalReservedWords

The AdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = Empty string*

### BuildArgumentsInIDE

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the CPP\_CG:[environment]:BuildCommandInIDE property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandInIDE

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration.

To change this property, use the Configuration window in the browser - do not change it by using the Properties tab in the Features window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link. The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*Default = Debug*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## CompileSwitches

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property.

Default =

```
-I. -I$OMDefaultSpecificationDirectory -I$(OMROOT) -I$(OMROOT)/LangCpp  
-I$(OMROOT)/LangCpp/oxf $(INST_FLAGS) $(INCLUDE_PATH) $(INST_INCLUDES)  
-DUSE_Iostream $OMCPPCompileCommandSet -c
```

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The default is as follows:

```
@echo Compiling $OMFileImpPath $(CREATE_OBJ_DIR) @$$(CC) $OMFileCPPCompileSwitches -o  
$OMFileObjPath $OMFileImpPath
```

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*Default = -g*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*Default = -O*

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

*Default =*

```
$OMFileObjPath : $OMFileImpPath $OMFileSpecPath $OMFileDependencies
```

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: \$OMFileObjPath : \$OMFileImpPath "\*.bmp" "\*.ico" "\*.rc2"

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction with an IDE such as Eclipse, the `EnableDebugIntegrationWithIDE` property can be used to specify whether or not the IDE debugger should be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to `True`, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's `exit()` function is called automatically. The system function calls the destructors of framework instances in the order it determines. The `EndApplicationCode` property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an `endApplication()` function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to `endApplication()`, it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The `EntryPoint` property specifies the name of the main program for a given environment.

Default value = `main`

If applicable, see also the definition of the `EntryPointDeclarationModifier` property for more information.

## ErrorMessageTokensFormat

The `ErrorMessageTokensFormat`, working with the `ParseErrorMessage` property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. `ErrorMessageTokensFormat` defines the number and location of tokens within the regular expression defined by the `ParseErrorMessage` property. `ErrorMessageTokens` has three parameters, each with an integer value:

- `TotalNumberOfTokens` - The number of tokens in the regular expression
- `FileTokenPosition` - The position of the file name token in the expression
- `LineTokenPosition` - The position of the line number token in the expression

*Default = `TotalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2`*



## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the CPP\_CG::<Environment>::ExeName property plus the value of this property.

*(Default = Blank)*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the CPP\_CG::<Environment>::ExeExtension property.

*(Default = Blank)*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements. The file inclusions are generated in the makefile.

*Default = \$OMSpecIncludeInElements \$OMImpIncludeInElements.*

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile. Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

The default is as follows:

```
$(OMROOT)\LangCpp\lib\sol2WebComponentsGNU$(LIB_EXT),  
$(OMROOT)\lib\sol2WebServicesGNU$(LIB_EXT),-lsocket -lnsl
```

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment. The default values are as follows:

*Default = .cpp*

## **Include**

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*Default = include*

## **InvokeAnimatedExecutable**

The AnimationPortRange value in the rhapsody.ini file makes it possible to use animation with multiple instances of Rhapsody running simultaneously.

Ordinarily, when you run an application with animation, the executable is launched using the command specified by the InvokeExecutable property. However, if you have specified a value greater than zero for AnimationPortRange in your rhapsody.ini file, the executable is launched using the command specified by the InvokeAnimatedExecutable property. By default, the value of InvokeAnimatedExecutable includes a reference to the port to use for animation.

*Default = xterm -e \$executable -port \$port*

## **InvokeExecutable**

The InvokeExecutable property specifies the command used to run an executable file.

*Default = xterm -e \$executable*

## **InvokeMake**

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT/etc/vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

*Default = \$OMROOT/etc/sol2make \$makefile \$maketarget*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .a*

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

*Default = -g*

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

*Default = -O*

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

*Default = \$OMLinkCommandSet*

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the CPP\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

(Default = .mak)

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration.

```
Default = ##### Target type (Debug/Release) #####
#####
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease
ConfigurationCPPCompileSwitches=$OMReusableStatechartSwitches
$OMConfigurationCPPCompileSwitches #####
Predefined macros ##### RM=/bin/rm -rf MD=/bin/mkdir -p INCLUDE_QUALIFIER=-I
TMPL_DIR=./Tmpl$(TARGET_NAME) CACHE_DIR=./SunWS_cache CC=CC -mt -ptr$(TMPL_DIR)
LIB_CMD=$(CC) LINK_CMD=$(CC) LIB_FLAGS=-xar $OMConfigurationLinkSwitches
LINK_FLAGS= -lposix4 -lpthread $OMConfigurationLinkSwitches
##### Context macros #####
$OMContextMacros OBJ_DIR=$OMObjectsDir ifeq ($(OBJ_DIR),) CREATE_OBJ_DIR=
CLEAN_OBJ_DIR= else CREATE_OBJ_DIR= $(MD) $(OBJ_DIR) CLEAN_OBJ_DIR= $(RM)
$(OBJ_DIR) endif ##### Predefined macros
##### $(OBS) : $(INST_LIBS) $(OXF_LIBS) ifeq ($(INSTRUMENTATION),Animation)
INST_FLAGS=-DOMANIMATOR INST_INCLUDES=-I$(OMROOT)/LangCpp/aom
-I$(OMROOT)/LangCpp/tom INST_LIBS= $(OMROOT)/LangCpp/lib/sol2aomanim$(LIB_EXT)
OXF_LIBS=$(OMROOT)/LangCpp/lib/sol2oxfinst$(LIB_EXT)
$(OMROOT)/LangCpp/lib/sol2omcomappl$(LIB_EXT) SOCK_LIB= -liostream -lsocket -lintl -lnsl -lCrun
-lCstd else ifeq ($(INSTRUMENTATION),Tracing) INST_FLAGS=-DOMTRACER
INST_INCLUDES=-I$(OMROOT)/LangCpp/aom -I$(OMROOT)/LangCpp/tom
INST_LIBS=$(OMROOT)/LangCpp/lib/sol2tomtrace$(LIB_EXT)
$(OMROOT)/LangCpp/lib/sol2aomtrace$(LIB_EXT) OXF_LIBS=
$(OMROOT)/LangCpp/lib/sol2oxfinst$(LIB_EXT) $(OMROOT)/LangCpp/lib/sol2omcomappl$(LIB_EXT)
SOCK_LIB= -liostream -lsocket -lintl -lnsl -lCrun -lCstd else ifeq ($(INSTRUMENTATION),None)
INST_FLAGS= INST_INCLUDES= INST_LIBS=
OXF_LIBS=$(OMROOT)/LangCpp/lib/sol2oxf$(LIB_EXT) SOCK_LIB= else @echo An invalid
Instrumentation $(INSTRUMENTATION) is specified. exit endif endif endif .SUFFIXES: $(CPP_EXT)
#####
##### Context dependencies and commands #####
```

```

$OMContextDependencies $OMFileObjPath : $OMMainImplementationFile $(OBJS) @$(CC)
$(ConfigurationCPPCompileSwitches) -o $OMFileObjPath $OMMainImplementationFile
##### #
Predefined linking instructions ## INST_LIBS is included twice to solve bi-directional dependency
between libraries #
##### #
$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS) $OMFileObjPath
$OMMakefileName $OMModelLibs @echo Linking $(TARGET_NAME)$(EXE_EXT) @$(LINK_CMD)
$OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \ $(OXF_LIBS) \
$(INST_LIBS) \ $(SOCK_LIB) \ $(LINK_FLAGS) -o $(TARGET_NAME)$(EXE_EXT)
$(TARGET_NAME)$(LIB_EXT) : $(OBJS) $(ADDITIONAL_OBJS) $OMMakefileName @echo Building
library @@ @$(LIB_CMD) $(LIB_FLAGS) -o $(TARGET_NAME)$(LIB_EXT) $(OBJS)
$(ADDITIONAL_OBJS) clean: @echo Cleanup $OMCleanOBJS $(RM) $OMFileObjPath
$(ADDITIONAL_OBJS) $(RM) $(TMPL_DIR) $(CACHE_DIR) $(RM) $(TARGET_NAME)$(LIB_EXT)
$(RM) $(TARGET_NAME)$(EXE_EXT) $(CLEAN_OBJ_DIR)

```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the CPP\_CG::<Environment>::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## ObjCleanCommand

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default = \$(RM) \$OMFileObjPath*

## ObjectName

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = Empty string*

## ObjectsDirectory

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = Empty string*

## ObjExtension

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*Default = .o*

## OSFileSystemCaseSensitive

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Checked*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = "([^\"]+)"[,][ ]line ([0-9]+)[:] (Error/Warning)*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation.

*Default = /*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Checked*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running Rational Rhapsody, whereas the target is the machine running the application. To run remotely, the UseRemoteHost property must be set to True. If UseRemoteHost is True and RemoteHost is blank, the current host name is used for the remote host. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = Empty string*

## ReusableStatechartSwitches

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts. See the upgrade history on the support site for information on this change.

*Default = -DOM\_REUSABLE\_STATECHART\_IMPLEMENTATION*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*Default = .h*

## UnixLineTerminationStyle

The UnixLineTerminationStyle property specifies whether generated files use the UNIX end-of-line style. If this property is set to Cleared, the end-of-line style depends on the host type (for example, DOS style on Windows machines, and UNIX style on Solaris machines).

*Default = Checked*

## UnixPathNameForOMROOT

The UnixPathNameForOMROOT property specifies whether the makefile must include UNIXstyle path names.

The pRISM compilers do not tolerate DOS-style path conventions. If you do not set this property correctly, there might be many compilation problems.

*Default = Checked*

## UpdateBuildSettingsInIDE

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## UseRemoteHost

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Checked*

## UseTemplateName

In the declarations for members of a template class, some compilers require that you use precede the member type with the keyword "typename" for types with dependent names. Since support for this keyword varies between compilers, the UseTemplateName property is used to specify whether the "typename" keyword should be included in the generated code.

*Default = Cleared*

# State

The State metaclass contains code generation properties related to states.

## DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- \$Name - the element name



- **\$FullName** - the full path of the element (P1::P2::C.a)
- **\$Description** - the text provided on the Description tab of the Features window for the element
- **\$Requirements** - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the **\$Requirements** keyword in the DescriptionTemplate property to get such comments generated for specific model elements.
- **\$Login** - the login name of the user who generated the file
- **\$CodeGeneratedDate** - the date on which the code was generated
- **\$CodeGeneratedTime** - the time at which the code was generated
- **\$RhapsodyVersion** - the version of Rhapsody that was used to generate the file
- For operations: **\$Signature** - the operation signature, **\$Arguments** - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: **\$Type** - the type of the attribute
- For arguments: **\$Type** - the argument type, **\$Direction** - the argument direction (In, Out, or InOut)
- For events: **\$Arguments** - the text provided on the Description tab of the Features window for the event arguments
- For association ends: **\$Target** - the other end of the association
- For requirements: **\$ID** - the ID specified for the requirement, **\$Specification** - the specification text for the requirement
- The values of tags can be included by using the syntax **\$tag\_name**. For example, if you use a tag called author for your classes, you can use the keyword **\$author** in the value of DescriptionTemplate in order to have the value of the author tag included in the element description.
- The values of properties can be included by using the syntax **\$property\_name**. For example, you can use the keyword **\$Animate** in the value of DescriptionTemplate in order to have the value of the Animate property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as **\$Name**)
- Tag keywords
- Property keywords

If the value of a keyword is a MultiLine value, then each new line (except the first one) starts with the value of the `<lang>_CG::Configuration::DescriptionBeginLine` property, and each line ends with the value of the `<lang>_CG::Configuration::DescriptionEndLine` property.

*Default = State \$Name [[Description: \$Description]]*

## EncloseFlowchartActionDescriptionTemplate

Ordinarily, when you specify the text to use as the value of the property DescriptionTemplate for a type of element, it is your responsibility to specify the comment symbols to use at the beginning and end of the text by providing values for the configuration-level properties DescriptionBeginLine and

DescriptionEndLine.

Prior to release 8.1.3, the descriptions for actions in flowcharts were an exception to this rule - comment symbols were added automatically by Rhapsody. This behavior was changed in 8.1.3 to align with the behavior for other types of elements. To make it possible for users to restore the previous code generation behavior for actions in flowcharts, the property EncloseFlowchartActionDescriptionTemplate was added in release 8.1.3 with a default value of False (value is True in the backward compatibility settings).

*Default = False (True in the backward compatibility settings)*

## Statechart

The Statechart metaclass contains the statechart code generation properties.

### AddAnnotationToAllTransitions

Prior to version 8.0, the code for certain transitions did not have Rhapsody annotations preceding it, resulting in problems with actions like roundtripping. Beginning in 8.0, all transition code is preceded by an appropriate annotation. To preserve the previous code generation behavior for pre-8.0 models, the lang\_CG::Statechart::AddAnnotationToAllTransitions property was added to the backward compatibility settings for C and C++ with a value of False.

*Default = False*

### AddDescriptionToActions

Prior to version 8.0, if no description was provided for an action in an activity, the generated code included a comment that reflected the description provided for the class that the action belongs to. Beginning in 8.0, the generated code no longer reflects the description of the owner class in such cases. To preserve the previous code generation behavior for pre-8.0 models, the lang\_CG::Statechart::AddDescriptionToActions property was added to the backward compatibility settings for C and C++ with a value of True.

*Default = True*

### GenerateActionOnExitOrderForNestedStatechartOldWay

Before version 7.5.3, the code generated for actions on exit was not put in the correct location in the generated code. This was corrected in version 7.5.3. In order to maintain the previous code generation behavior for older models, a property called [lang]\_CG::Statechart::GenerateActionOnExitOrderForNestedStatechartOldWay was added to the C, CPP, and Java backward compatibility profiles for 7.5.3 with a value of True.

## Simplify

If you are using the Rational Rhapsody customizable code generation mechanism, the Simplify property can be used to change the way specific types of elements are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - The element is ignored.
- Copy - The element is copied from the original to the simplified model. It is not modified in any way.
- Default - Uses the standard simplification for this item, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for the element has been applied.

*Default = "Default"*

## SortAttributesByType

When statechart code is generated, a number of attributes are added to the code for the relevant class. Prior to version 8.0, these statechart-related attributes were not ordered by type in the generated code. Beginning in 8.0, these statechart-related attributes are ordered by type in order to make more efficient use of memory. To preserve the previous code generation behavior for pre-8.0 models, the `CPP_CG::Statechart::SortAttributesByType` property was added to the backward compatibility settings for C++ with a value of `False`.

## StatechartImplementation

Prior to version 7.3 of Rational Rhapsody, the transition-handling code that is generated by Rational Rhapsody used a switch statement to represent the possible states. Beginning with version 7.3, this code uses an if/else structure. To allow older models to use the previous code generation behavior, a property called `StatechartImplementation` was added to the Pre73 backward compatibility profiles. The possible values for the property are:

- SwitchOnly - transition-handling code uses a switch statement to represent the possible states
- Default - the transition-handling code uses an if/else structure to represent the possible states

*Default = SwitchOnly*

## StatechartStateOperations

The `StatechartStateOperations` property controls the state operations in a statechart.

*Default = None*

## UsePreviousTimeoutCancellationCode

For C++ applications, the code for timeout cancellation was replaced by a single function in version 7.6.1.

In order to maintain the previous code generation behavior for pre-7.6.1 models, the UsePreviousTimeoutCancellationCode property was added to the C++ backward compatibility settings with a value of True.

## Transition

The Transition metaclass contains code generation properties related to transitions.

## DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- `$Name` - the element name
- `$FullName` - the full path of the element (P1::P2::C.a)
- `$Description` - the text provided on the Description tab of the Features window for the element
- `$Requirements` - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the `$Requirements` keyword in the DescriptionTemplate property to get such comments generated for specific model elements.
- `$Login` - the login name of the user who generated the file
- `$CodeGeneratedDate` - the date on which the code was generated
- `$CodeGeneratedTime` - the time at which the code was generated
- `$RhapsodyVersion` - the version of Rhapsody that was used to generate the file
- For operations: `$Signature` - the operation signature, `$Arguments` - the text provided on the Description tab of the Features window for the operation arguments

- For attributes: \$Type - the type of the attribute
- For arguments: \$Type - the argument type, \$Direction - the argument direction (In, Out, or InOut)
- For events: \$Arguments - the text provided on the Description tab of the Features window for the event arguments
- For association ends: \$Target - the other end of the association
- For requirements: \$ID - the ID specified for the requirement, \$Specification - the specification text for the requirement
- The values of tags can be included by using the syntax \$tag\_name. For example, if you use a tag called author for your classes, you can use the keyword \$author in the value of DescriptionTemplate in order to have the value of the author tag included in the element description.
- The values of properties can be included by using the syntax \$property\_name. For example, you can use the keyword \$Animate in the value of DescriptionTemplate in order to have the value of the Animate property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as \$Name)
- Tag keywords
- Property keywords

If the value of a keyword is a MultiLine value, then each new line (except the first one) starts with the value of the <lang>\_CG::Configuration::DescriptionBeginLine property, and each line ends with the value of the <lang>\_CG::Configuration::DescriptionEndLine property.

*Default = [[Description: \$Description]]*

## Type

The Type metaclass contains a property that affects the visibility of data types.

### AnimEnumerationTypeImage

The AnimEnumerationTypeImage property is a Boolean value that determines whether the Image attribute is used for enumerated types when you use animation.

*Default = False*

### AnimSerializeOperation

The AnimSerializeOperation property specifies the name of an external function used to animate all attributes and arguments that are of that type. Rational Rhapsody can animate (display) the values of simple types and one-dimensional arrays without any problem. To display the current values of such

attributes during an animation session, run the features window for the instance. However, if you want to animate a more complex type, such as a date, the type must be converted to a string (char \*) for Rational Rhapsody to display it. This is generally done by writing a global function, an instrumentation function, that takes one argument of the type you want to display, and returns a char \*. You must disable animation of the instrumentation function itself (by using the Animate and AnimateArguments properties for the function). For example, you can have a type tDate, defined as follows:

```
typedef struct date { int day; int month; int year; } %s;
```

You can have an object with an attribute count of type int, and an attribute date of type tDate. The object can have an initializer with the following body:

```
me-date.month = 5; me-date.day = 12; me-date.year = 2000; If you want to animate the date attribute, the AnimSerializeOperation property for date must be set to the name of a function that converts the type tDate to char *. For example, you can set the property to a function named showDate. This function name must be entered without any parentheses. It must take an attribute of type tDate and return a char *. The Animate and AnimateArguments properties for the showDate function must be set to False. The implementation of the showDate function might be as follows: showDate(tDate aDate) { char* buff; buff = (char*) malloc(sizeof(char) * 20); sprintf(buff,"%d %d %d", aDate.month,aDate.day,aDate.year); return buff; }
```

When you run this model with animation, instances of this object display a value of 5 12 2000 for the date attribute in the browser. If the showDate function is defined in the same class that the attribute belongs to and the function is not static, the AnimSerializeOperation property value should be similar to the following: myReal-showDate This value shows that the function is called from the serializeAttributes function, located in the class OMAAnimatedclassname. The showDate function must allocate memory for the returned string by way of the malloc/alloc/calloc function in C, or the new operator in C++. Otherwise, the system shuts down unexpectedly.

*Default = Empty string*

## AnimUnserializeOperation

The AnimUnserializeOperation property converts a string to the value of an element (the opposite of the AnimSerializeOperation property). Unserialize functions are used for event generation or operation invocation by using the Animation toolbar to convert the string (received from the user) to the value of the event or operation before the event generation or operation invocation. For example, your serialization operation might look similar to the following:

```
char* myX2String(const Rec f) { char* cS = new char[OutputStringLength]; /* conversion from the Rec type to string */ return (cS); } The unserialization operation would be: Rec myString2X (char* C, Rec T) { T = new Trc; /* conversion of the string C to the Rec type */ delete C; return (T); }
```

*Default = Empty string*

## AnimUseMultipleSerializationFunctions

The AnimSerializeOperation and AnimUnserializeOperation properties are used to specify user-provided functions for serialization/unserialization of objects to allow inclusion of such objects in animation.

Because Rational Rhapsody allows you to fine-tune code generation of arguments by using the In, Out, InOut, and TriggerArgument properties, you might need to provide multiple serialization/unserialization functions to handle these different types of arguments. You can use the AnimUseMultipleSerializationFunctions property to instruct Rational Rhapsody to use multiple user-provided serialization/unserialization functions.

If you set the value of this property to Checked, Rational Rhapsody searches for user-provided serialization functions whose names consist of the string entered for the AnimSerializeOperation property and the suffixes "In", "Out", "InOut", and "TriggerArgument".

The same is true for unserialization functions. However, for unserialization, Rational Rhapsody cannot handle Out arguments, so the relevant suffixes are "In", "InOut", and "TriggerArgument".

Since the AnimSerializeOperation and AnimUnserializeOperation properties exist under both the Type metaclass and the Class metaclass, the AnimUseMultipleSerializationFunctions property also exists under both these metaclasses.

*Default = Cleared*

## DeclarationModifier

The DeclarationModifier property is used to enable Rational Rhapsody to reverse engineer non-standard keywords that appear in type declarations. Keywords that appear between the type of the type (structure, enumeration, or union) and the type name are stored as the value of this property, and the property is then used during code generation to re-create the original code.

Since this is a code generation property, it can also be used to add non-standard keywords to code even when reverse engineering is not used.

This property is used in conjunction with the PreDeclarationModifier and PostDeclarationModifier properties.

*Default = Blank*

## DeclarationPosition

The DeclarationPosition property specifies where the type declaration is displayed. The possible values are as follows:

- BeforeClassRecord - The type declaration is displayed before the class record (CR) declaration if CR has a visibility set to public, and before the class record forward declaration if CR has a visibility set to private.
- AfterClassRecord - The type declaration is displayed after the class record declaration if CR has a visibility set to public, and after the class record forward declaration if CR has a visibility set to private.
- StartOfDeclaration - The type declaration is displayed among the first declarations (together with other types having the same settings) in the public section if CR has a visibility set to public, and among the first declarations in the private section (together with other types having the same settings) if CR has a visibility set to private.

- EndOfDeclaration - The type declaration is displayed among the last declarations (together with other types having the same settings) in the public section if CR has a visibility set to public, and among the last declarations in the private section (together with other types having the same settings) if CR has a visibility set to private.

If the `CPP_CG::Type::Visibility` property is set to "Body", no matter the settings of `CPP_CG::Type::DeclarationPosition` property, the type declaration still displays in the package body.

*Default = BeforeClassRecord*

## DefaultValue

When you use ports, it is possible to have a situation where your application tries to call an operation that is part of a required interface for a port, but the service is not available. In such cases, this might result in problems with the class/type that is supposed to be returned by that operation.

The DefaultValue property allows you to define a value, such as null, that can be returned in such situations.

*Default = Blank*

## DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- \$Name - the element name
- \$FullName - the full path of the element (P1::P2::C.a)
- \$Description - the text provided on the Description tab of the Features window for the element
- \$Requirements - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the \$Requirements keyword in the DescriptionTemplate property to get such comments generated for specific model elements.
- \$Login - the login name of the user who generated the file
- \$CodeGeneratedDate - the date on which the code was generated
- \$CodeGeneratedTime - the time at which the code was generated



- `$RhapsodyVersion` - the version of Rhapsody that was used to generate the file
- For operations: `$Signature` - the operation signature, `$Arguments` - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: `$Type` - the type of the attribute
- For arguments: `$Type` - the argument type, `$Direction` - the argument direction (In, Out, or InOut)
- For events: `$Arguments` - the text provided on the Description tab of the Features window for the event arguments
- For association ends: `$Target` - the other end of the association
- For requirements: `$ID` - the ID specified for the requirement, `$Specification` - the specification text for the requirement
- The values of tags can be included by using the syntax `$tag_name`. For example, if you use a tag called `author` for your classes, you can use the keyword `$author` in the value of `DescriptionTemplate` in order to have the value of the `author` tag included in the element description.
- The values of properties can be included by using the syntax `$property_name`. For example, you can use the keyword `$Animate` in the value of `DescriptionTemplate` in order to have the value of the `Animate` property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as `$Name`)
- Tag keywords
- Property keywords

If the value of a keyword is a `MultiLine` value, then each new line (except the first one) starts with the value of the `<lang>_CG::Configuration::DescriptionBeginLine` property, and each line ends with the value of the `<lang>_CG::Configuration::DescriptionEndLine` property.

*Default = Blank*

## EnumerationAsTypedef

The `EnumerationAsTypedef` property specifies whether the generated enum should be wrapped by a typedef. This property is applicable to enumeration types in C and C++.

*Default = Cleared*

## ImplementationEpilog

The `ImplementationEpilog` property enables you to add any code that you want to be added as verbatim text (to be ignored by Rational Rhapsody) to the end of the definition of a model element.

For example, you could wrap a section of code with an `#ifdef-#endif` pair, add compiler-specific keywords, or add a `#pragma` statement. For example, to specify that an operation is available only when the code is compiled with `_DEBUG`, set the following properties for the operation:

- Set SpecificationProlog to `#ifdef _DEBUG cr.`
- Set SpecificationEpilog to `#endif.`
- Set ImplementationProlog to `#ifdef _DEBUG cr.`
- Set ImplementationEpilog to `#endif.`

The following table lists whether leading and trailing linefeeds are generated.

Metaclass	Leading Linefeed Added?	Class	Yes	Package	No

*Default = Empty multiLine*

## ImplementationName

The ImplementationName property enables you to give a type one model name and generate it with another name.

*Default = Empty string*

## ImplementationProlog

The ImplementationProlog property enables you to add any code that you want to be added as verbatim text (to be ignored by the software) to the beginning of the definition of a model element.

For example, you could wrap a section of code with an `#ifdef-#endif` pair, add compiler-specific keywords, or add a `#pragma` statement. For example, to specify that an operation is available only when the code is compiled with `_DEBUG`, set the following properties for the operation:

- Set SpecificationProlog to `#ifdef _DEBUG cr.`
- Set SpecificationEpilog to `#endif.`
- Set ImplementationProlog to `#ifdef _DEBUG cr.`
- Set ImplementationEpilog to `#endif.`

The following table lists whether leading and trailing linefeeds are generated.

Metaclass	Trailing Linefeed Added?	Class	No	Package	Yes

*Default = Empty MultiLine)*

## In

The In property specifies how code is generated when the type is used with an argument that has the modifier "In".

*Default = const \$type&*

## InOut

The InOut property specifies how code is generated when the type is used with an argument that has the modifier "InOut".

*Default = \$type&*

## IsLimited

The IsLimited property determines whether the class or record type is generated as limited.

*Default = False*

## LanguageMap

The LanguageMap property specifies the Ada declaration for Rational Rhapsody language-independent types.

*Default = Empty string*

## OpeningBraceStyle

The OpeningBraceStyle property controls where the opening brace of the code block is positioned - on the same line as the element name (SameLine) or on the following line (NewLine).

## Out

The Out property specifies how code is generated when the type is used with an argument that has the modifier "Out".

*Default = \$type\*&*

## PostDeclarationModifier

The PostDeclarationModifier property is used to enable Rational Rhapsody to reverse engineer non-standard keywords that appear in type declarations. Keywords that appear after the type name are stored as the value of this property, and the property is then used during code generation to re-create the original code.

Since this is a code generation property, it can also be used to add non-standard keywords to code even when reverse engineering is not used.

This property is used in conjunction with the PreDeclarationModifier and DeclarationModifier properties.

*Default = Blank*

## PreDeclarationModifier

The PreDeclarationModifier property is used to enable Rational Rhapsody to reverse engineer non-standard keywords that appear in type declarations. Keywords that appear before the type of the type (structure, enumeration, or union) are stored as the value of this property, and the property is then used during code generation to re-create the original code.

Since this is a code generation property, it can also be used to add non-standard keywords to code even when reverse engineering is not used.

This property is used in conjunction with the DeclarationModifier and PostDeclarationModifier properties.

*Default = Blank*

## PrivateName

The PrivateName property specifies the pattern used to generate names of private operations in C.

*Default = \$typeName*

## PublicName

The PublicName property specifies the pattern used to generate names of public operations in C.

*Default = \$objectName\_\$typeName*

## ReferenceImplementationPattern

The ReferenceImplementationPattern property specifies how the "Reference" option for attribute/typedefs (composite types) is mapped to code.

*Default = \**

## ReturnType

The ReturnType property specifies how code is generated when the type is used as a return type.

*Default = \$type\**

## Simplify

If you are using the Rational Rhapsody customizable code generation mechanism, the Simplify property can be used to change the way specific types of elements are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - The element is ignored.
- Copy - The element is copied from the original to the simplified model. It is not modified in any way.
- Default - Uses the standard simplification for this item, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for the element has been applied.

*Default = "Default"*

## SpecificationEpilog

The SpecificationEpilog property allows you to add code to the end of the declaration of a model element.

For example, you can use this property to add an `#ifdef` to indicate that a type is available only when the code is compiled with `_DEBUG`, by setting the following properties:

- For SpecificationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For SpecificationEpilog, enter the value `#endif`
- For ImplementationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For ImplementationEpilog, enter the value `#endif`

*Default = Blank*

## SpecificationProlog

The SpecificationProlog property allows you to add code to the beginning of the declaration of a model element.

For example, you can use this property to add an `#ifdef` to indicate that a type is available only when the code is compiled with `_DEBUG`, by setting the following properties:

- For SpecificationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For SpecificationEpilog, enter the value `#endif`
- For ImplementationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For ImplementationEpilog, enter the value `#endif`

*Default = Blank*

## StructAsTypedef

The StructAsTypedef property specifies whether the generated struct should be wrapped by a typedef. This property is applicable to structure types in C and C++.

*Default = Cleared*

## TriggerArgument

The TriggerArgument property is used for mapping event and triggered operation arguments to code instead of the In, InOut, and Out properties. A different property is required because of code generation limitations related to event arguments. See also:

- In
- InOut
- Out

*Default = \$type*

## UnionAsTypedef

The UnionAsTypedef property specifies whether the generated union should be wrapped by a typedef. This property is applicable to union types in C and C++.

*Default = Cleared*

## Visibility

The Visibility property specifies the visibility of the model element. Code generation maps the visibility specified for an element to the same visibility in the generated language. The possible values are as follows:

- Public - The model element is public.
- Protected - The model element is protected.
- Private - The element is private.

*Default = Public*

## VxWorks

The VxWorks metaclass contains environment settings (Compiler, framework libraries, and so on) for VxWorks 5.4.

## AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the `AdaptorSearchPath` property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)/LangCpp/osconfig/VxWorks*

## **AdditionalReservedWords**

The `AdditionalReservedWords` property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = Empty string*

## **BSP**

The `BSP` property specifies the board support package (BSP) for the system. If you need to change the value of the CPU, you can simply reset the value in this property instead of changing the value in the `MakeFileContent` property.

*Default = PENTIUM*

## **BuildArgumentsInIDE**

The `BuildArgumentsInIDE` property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the `CPP_CG:[environment]:BuildCommandInIDE` property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the `BuildInIDE` property is set to `True`.

*Default = Blank*

## BuildCommandInIDE

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration.

To change this property, use the Configuration window in the browser - do not change it by using the Properties tab in the Features window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link. The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*Default = Debug*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\$OMROOT/etc/Executer.exe" "\""\$OMROOT\etc\vx6make.bat" vxbuild.mak buildLibs  
\"CPU=\$BSP\" \"BUILD=\$BuildCommandSet\" \" \"*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features



window for configurations.

*Default = Cleared*

## CodeTestSettings

The CodeTestSettings property specifies the compiler command settings. This property supports integration with Applied Microsystems Corporation CodeTest.

*Default = CXX=\$AMC\_HOME)\bin\ctcxx*

## CompileSwitches

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property.

*Default =*

*-I\$OMDefaultSpecificationDirectory -I\$(OMROOT) -I\$(OMROOT)/LangCpp  
-I\$(OMROOT)/LangCpp/oxf -DVxWorks \$(INST\_FLAGS) \$(INCLUDE\_PATH)  
\$OMCPPCompileCommandSet -c*

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The default is as follows:

*@echo Compiling \$OMFileImpPath \$(CREATE\_OBJ\_DIR) @\$(CXX) \$(C++FLAGS)  
\$OMFileCPPCompileSwitches -o \$OMFileObjPath \$OMFileImpPath*

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*Default = -O0 -g*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*Default = Empty string*

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

The default is as follows:

```
$OMFileObjPath : $OMFileImpPath $OMFileSpecPath $OMFileDependencies
```

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: `$OMFileObjPath : $OMFileImpPath "*.bmp" "*.ico" "*.rc2"`

## DuplicateLibsListInMakeFile

The DuplicateLibsListInMakeFile property is a Boolean value that specifies whether Rational Rhapsody should duplicate the libraries list in the generated makefile link command. This property supports linkers that are sensitive to library order in the link command.

*Default = Checked*

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction with an IDE such as Eclipse, the EnableDebugIntegrationWithIDE property can be used to specify whether or not the IDE debugger should be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to True, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's exit() function is called automatically. The system function calls the destructors of framework instances in the order it determines. The EndApplicationCode property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an endApplication() function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to endApplication(), it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The EntryPoint property specifies the name of the main program for a given environment.

*Default = vxmain*

If applicable for the metaclass, see also the definition of the EntryPointDeclarationModifier property for more information.

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens has three parameters, each with an integer value:

- TotalNumberOfTokens - The number of tokens in the regular expression
- FileTokenPosition - The position of the file name token in the expression
- LineTokenPosition - The position of the line number token in the expression

*Default = TotalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2*

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the CPP\_CG::<Environment>::ExeName property plus the value of this property.

*(Default = .out)*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the CPP\_CG::<Environment>::ExeExtension property.

*(Default = Blank)*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements. The file inclusions are generated in the makefile.

*Default = \$OMSpecIncludeInElements \$OMImpIncludeInElements*

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile. Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

*Default =*

`$(OMROOT)/LangCpp/lib/vxWebComponents$(CPU)$(LIB_EXT),  
$(OMROOT)/lib/vxWebServices$(CPU)$(LIB_EXT)`

## HasIDEInterface

The HasIDEInterface property is a Boolean value that specifies whether IDE support is enabled. If IDE support is enabled (Checked), the IDEInterfaceDLL property points to an IDE adapter that provides connection to the IDE. If the property is set to Cleared, IDE support is disabled and IDE services are not attempted. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = Checked*

## IDEInterfaceDLL

The IDEInterfaceDLL property is a string that points to the IDE adapter DLL. You should not have any reason to modify this property. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = \$OMROOT/DLLs/TornadoIDE.dll*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment. The default values are as follows:

*Default = .cpp*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to

include other makefiles.

*Default = include*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\vxmake.bat\" \$makefile \$maketarget"*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .a*

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

*Default = -g*

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

*Default = Empty string*

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

*Default = \$OMLinkCommandSet*

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the CPP\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*(Default = .mak)*

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration.

```
Default = ##### Target type (Debug/Release) #####
#####
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease
ConfigurationCPPCompileSwitches=$OMReusableStatechartSwitches
$OMConfigurationCPPCompileSwitches #####
##### Definitions and flags #####
```

```
##### CPU = $BSP TOOL = gnu include
$(WIND_BASE)/target/h/make/defs.bsp.cpp.o : @ $(RM) $@ $(CXX) $(C++FLAGS)
$(OPTION_OBJECT_ONLY) $< $OMCodeTestSettings INCLUDE_QUALIFIER=-I LIB_CMD=$(AR)
LINK_CMD=$(LD) LIB_FLAGS=$(ARFLAGS) #LINK_FLAGS=$OMConfigurationLinkSwitches -r
$(LDLFLAGS) LINK_FLAGS=$OMConfigurationLinkSwitches -r
##### Context generated
macros ##### $OMContextMacros OBJ_DIR=$OMObjectsDir ifeq ($(OBJ_DIR),)
CREATE_OBJ_DIR= CLEAN_OBJ_DIR= else CREATE_OBJ_DIR= if not exist $(OBJ_DIR) mkdir
$(OBJ_DIR) CLEAN_OBJ_DIR= if exist $(OBJ_DIR) rmdir $(OBJ_DIR) endif
##### Predefined macros
##### $(OBJS) : $(INST_LIBS) $(OXF_LIBS) ifeq ($(INSTRUMENTATION),Animation)
INST_FLAGS=-DOMANIMATOR-DUSE_Iostream INST_INCLUDES=-I$(OMROOT)/LangCpp/aom
-I$(OMROOT)/LangCpp/tom INST_LIBS= $(OMROOT)/LangCpp/lib/vxaomanim$(CPU)$(LIB_EXT)
OXF_LIBS=$(OMROOT)/LangCpp/lib/vxoxfinst$(CPU)$(LIB_EXT)
$(OMROOT)/LangCpp/lib/vxomcomappl$(CPU)$(LIB_EXT) SOCK_LIB= else ifeq
$(INSTRUMENTATION),Tracing INST_FLAGS=-DOMTRACER-DUSE_Iostream
INST_INCLUDES=-I$(OMROOT)/LangCpp/aom -I$(OMROOT)/LangCpp/tom
INST_LIBS=$(OMROOT)/LangCpp/lib/vxtomtrace$(CPU)$(LIB_EXT)
$(OMROOT)/LangCpp/lib/vxaomtrace$(CPU)$(LIB_EXT) OXF_LIBS=
$(OMROOT)/LangCpp/lib/vxoxfinst$(CPU)$(LIB_EXT)
$(OMROOT)/LangCpp/lib/vxomcomappl$(CPU)$(LIB_EXT) SOCK_LIB= else ifeq
$(INSTRUMENTATION),None INST_FLAGS= INST_INCLUDES= INST_LIBS=
OXF_LIBS=$(OMROOT)/LangCpp/lib/vxoxf$(CPU)$(LIB_EXT) SOCK_LIB= else echo 'An invalid
Instrumentation $(INSTRUMENTATION) is specified.' exit endif endif endif
#####
Context generated dependencies ##### $OMContextDependencies $OMFileObjPath :
$OMMainImplementationFile $(OBJS) @echo Compiling $OMMainImplementationFile @$(CXX)
$(C++FLAGS) $(ConfigurationCPPCompileSwitches) -o $OMFileObjPath $OMMainImplementationFile
##### #
Predefined linking instructions # # INST_LIBS is included twice to solve bi-directional dependency
between libraries #
#####
$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS) $OMFileObjPath
$OMMakefileName $OMModelLibs @echo Linking and Munching $(TARGET_NAME)$(EXE_EXT)
@$(LINK_CMD) $(LINK_FLAGS) -o $(TARGET_NAME).tmp \ $OMFileObjPath $(OBJS)
$(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \ $(OXF_LIBS) \ $(INST_LIBS) \ $(SOCK_LIB) @
$(RM) $(TARGET_NAME)$(EXE_EXT) ctdt.c ctdt.o @$(NM) $(TARGET_NAME).tmp / $(MUNCH) >
ctdt.c @$(CC) -c ctdt.c @$(LD) -r $OMLinkCommandSet -o $@ $(TARGET_NAME).tmp ctdt.o @ $(RM)
ctdt.c ctdt.o $(TARGET_NAME).tmp $(TARGET_NAME)$(LIB_EXT) : $(OBJS) $(ADDITIONAL_OBJS)
$OMMakefileName @echo Building library $@ @$(LIB_CMD) $(LIB_FLAGS)
$(TARGET_NAME)$(LIB_EXT) $(OBJS) $(ADDITIONAL_OBJS) clean: cleanall: clean @echo Cleanup
$(RM) $OMFileObjPath $(RM) $(TARGET_NAME)$(LIB_EXT) $(RM) $(TARGET_NAME)$(EXE_EXT)
$OMCleanOBJS $(CLEAN_OBJ_DIR)
```

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## ObjCleanCommand

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default = \$(RM) \$OMFileO*

## ObjectName

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = Empty string*

## ObjectsDirectory

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = Empty string*

## ObjExtension

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*Default = .o*

## OSFileSystemCaseSensitive

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Cleared*

## ParseErrorDescript

The ParseErrorDescript property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = ([^:]+):([0-9]+):(.\*)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and



(Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = ([^:]+):([0-9]+):*

## ParseMakeError

The ParseMakeError property is used to define a regular expression that represents the format of make process or linker error messages. This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = (make)(.)\*(Error)*

## ParseSeverityError

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^:]+):([0-9]+):*

## ParseSeverityWarning

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^:]+):([0-9]+): (warning)*

## PathWhiteSpaceHandling

For different operating systems, there are different methods for handling spaces in file paths, for example, enclosing the entire path in quotation marks. The PathWhiteSpaceHandling property allows you to specify the method that should be used for a given environment. The possible values are:

- NoHandling - the path should be left as is, with no special handling for spaces
- SurroundingQuotes - the entire path should be enclosed in quotation marks
- BackslashBeforeSpace - spaces in paths should be preceded by backslashes, as is the practice in VxWorks platforms

*Default = BackslashBeforeSpace*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in

double quotation marks in the generated makefile.

*Default = Checked*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running Rational Rhapsody, whereas the target is the machine running the application. To run remotely, the UseRemoteHost property must be set to True. If UseRemoteHost is True and RemoteHost is blank, the current host name is used for the remote host. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = Empty string*

## ReusableStatechartSwitches

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts. See the upgrade history on the support site for information on this change.

*Default = -DOM\_REUSABLE\_STATECHART\_IMPLEMENTATION*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*Default = .h*

## UnixLineTerminationStyle

The UnixLineTerminationStyle property specifies whether generated files use the UNIX end-of-line style. If this property is set to Cleared, the end-of-line style depends on the host type (for example, DOS style on Windows machines, and UNIX style on Solaris machines).

*Default = Checked*

## UpdateBuildSettingsInIDE

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## UseNewBuildOutputWindow

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the CG::General::ShowLogViewAfterBuild property to True.

*Default = Checked*

## UseNonZeroStdInputHandle

The UseNonZeroStdInputHandle property is a Boolean value that specifies whether to use a non-zero standard input handle.

*Default = Checked*

## UseRemoteHost

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Checked*

## UseTemplateTypename

In the declarations for members of a template class, some compilers require that you use precede the member type with the keyword "typename" for types with dependent names. Since support for this keyword varies between compilers, the UseTemplateTypename property is used to specify whether the "typename" keyword should be included in the generated code.

*Default = Cleared*

## VxWorks6diab

The VxWorks6diab metaclass contains settings (Compiler, framework libraries, and so on) for VxWorks 6.x, by using the WindRiver-Compiler compiler (previously named "Diab").

## AdaptorSearchPath

The `AdaptorSearchPath` property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the `AdaptorSearchPath` property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)/LangCpp/osconfig/VxWorks*

## **AdditionalReservedWords**

The `AdditionalReservedWords` property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = Empty string*

## **BSP**

The `BSP` property specifies the board support package (BSP) for the system. If you need to change the value of the CPU, you can simply reset the value in this property instead of changing the value in the `MakeFileContent` property.

*Default = PENTIUM*

## **BuildArgumentsInIDE**

The `BuildArgumentsInIDE` property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the `CPP_CG:[environment]:BuildCommandInIDE` property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the `BuildInIDE` property is set to

True.

*Default = Blank*

## BuildCommandInIDE

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration.

To change this property, use the Configuration window in the browser - do not change it by using the Properties tab in the Features window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link. The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*Default = Debug*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\$OMROOT/etc/Executer.exe" "\""\$OMROOT/etc/vx6make.bat" vxbuild.mak buildLibs 6.5 \"CPU=\$BSP\" \"TOOL=diab\" \"TOOL\_FAMILY=diab\" \"BUILD=\$BuildCommandSet\" \" \"*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform

the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## CodeTestSettings

The CodeTestSettings property specifies the compiler command settings. This property supports integration with Applied Microsystems Corporation CodeTest.

*Default = CXX = \$(AMC\_HOME)\bin\ctcxx*

## CompileSwitches

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property.

*Default =*

*\$IgnoreSwitches -I\$OMDefaultSpecificationDirectory -I\$(OMROOT) -I\$(OMROOT)/LangCpp  
-I\$(OMROOT)/LangCpp/oxf -DVxWorks \$(INST\_FLAGS) \$(INCLUDE\_PATH)  
\$OMCPPCompileCommandSet -c*

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The default is as follows:

*@echo Compiling \$OMFileImpPath \$(CREATE\_OBJ\_DIR) @\$(CXX) \$(C++FLAGS)  
\$OMFileCPPCompileSwitches -o \$OMFileObjPath \$OMFileImpPath*

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*Default = -g*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*Default = Empty string*

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

The default is as follows:

```
$OMFileObjPath : $OMFileImpPath $OMFileSpecPath $OMFileDependencies
```

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: `$OMFileObjPath : $OMFileImpPath "*.bmp" "*.ico" "*.rc2"`

## DuplicateLibsListInMakeFile

The DuplicateLibsListInMakeFile property is a Boolean value that specifies whether Rational Rhapsody should duplicate the libraries list in the generated makefile link command. This property supports linkers that are sensitive to library order in the link command.

*Default = Checked*

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction with an IDE such as Eclipse, the EnableDebugIntegrationWithIDE property can be used to specify whether or not the IDE debugger should be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to True, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's exit() function is called automatically. The system function calls the destructors of framework instances in the order it determines. The EndApplicationCode property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an endApplication() function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to endApplication(), it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The EntryPoint property specifies the name of the main program for a given environment.

*Default = vxmain*

If applicable for the metaclass, see also the definition of the EntryPointDeclarationModifier property for more information.

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens has three parameters, each with an integer value:

- TotalNumberOfTokens - The number of tokens in the regular expression
- FileTokenPosition - The position of the file name token in the expression
- LineTokenPosition - The position of the line number token in the expression

*Default = ToTalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2*

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the CPP\_CG::<Environment>::ExeName property plus the value of this property.

*(Default = .out)*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the CPP\_CG::<Environment>::ExeExtension property.



*(Default = Blank)*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements. The file inclusions are generated in the makefile.

*Default = \$OMSpecIncludeInElements \$OMImpIncludeInElements*

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile. Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

*Default =*

`$(OMROOT)/LangCpp/lib/vxWebComponents$(CPU)$(TOOL)$(LIB_EXT),  
$(OMROOT)/lib/vxWebServices$(CPU)$(TOOL)$(LIB_EXT)`

## HasIDEInterface

The HasIDEInterface property is a Boolean value that specifies whether IDE support is enabled. If IDE support is enabled (Checked), the IDEInterfaceDLL property points to an IDE adapter that provides connection to the IDE. If the property is set to Cleared, IDE support is disabled and IDE services are not attempted. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = Checked*

## IDEInterfaceDLL

The IDEInterfaceDLL property is a string that points to the IDE adapter DLL. You should not have any reason to modify this property. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = \$OMROOT/DLLs/WorkbenchDebuggerIDE.dll*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment. The default values are as follows:

*Default = .cpp*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*Default = include*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\vx6make.bat\" \$makefile \$maketarget"*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .a*

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

*Default = -X -r*

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

*Default = -X -r*

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

*Default = \$OMLinkCommandSet*

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the CPP\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*(Default = .mak)*

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration.

```
Default = ##### Target type (Debug/Release) #####  
#####  
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease
```

```

LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease
ConfigurationCPPCompileSwitches=$OMReusableStatechartSwitches
$OMConfigurationCPPCompileSwitches #####
##### Definitions and flags #####
##### CPU = $BSP TOOL = diab include
$(WIND_BASE)/target/h/make/defs.bsp.cpp.o : @ $(RM) $@ $(CXX) $(C++FLAGS)
$(OPTION_OBJECT_ONLY) $< $OMCodeTestSettings INCLUDE_QUALIFIER=-I LIB_CMD=$(AR)
LINK_CMD=$(LD) $(CC_ARCH_SPEC) LIB_FLAGS=$(ARFLAGS)
LINK_FLAGS=$OMConfigurationLinkSwitches -r5
##### Context generated
macros ##### $OMContextMacros OBJ_DIR=$OMObjectsDir ifeq ($(OBJ_DIR),)
CREATE_OBJ_DIR= CLEAN_OBJ_DIR= else CREATE_OBJ_DIR= if not exist $(OBJ_DIR) mkdir
$(OBJ_DIR) CLEAN_OBJ_DIR= if exist $(OBJ_DIR) rmdir $(OBJ_DIR) endif
##### Predefined macros
##### $(OBJS) : $(INST_LIBS) $(OXF_LIBS) ifeq ($(INSTRUMENTATION),Animation)
INST_FLAGS=-DOMANIMATOR -DUSE_Iostream INST_INCLUDES=-I$(OMROOT)/LangCpp/aom
-I$(OMROOT)/LangCpp/tom INST_LIBS=
$(OMROOT)/LangCpp/lib/vxaomanim$(CPU)$(TOOL)$(LIB_EXT)
OXF_LIBS=$(OMROOT)/LangCpp/lib/vxoxfinst$(CPU)$(TOOL)$(LIB_EXT)
$(OMROOT)/LangCpp/lib/vxomcomappl$(CPU)$(TOOL)$(LIB_EXT) SOCK_LIB= else ifeq
$(INSTRUMENTATION),Tracing INST_FLAGS=-DOMTRACER -DUSE_Iostream
INST_INCLUDES=-I$(OMROOT)/LangCpp/aom -I$(OMROOT)/LangCpp/tom
INST_LIBS=$(OMROOT)/LangCpp/lib/vxtomtrace$(CPU)$(TOOL)$(LIB_EXT)
$(OMROOT)/LangCpp/lib/vxaomtrace$(CPU)$(TOOL)$(LIB_EXT) OXF_LIBS=
$(OMROOT)/LangCpp/lib/vxoxfinst$(CPU)$(TOOL)$(LIB_EXT)
$(OMROOT)/LangCpp/lib/vxomcomappl$(CPU)$(TOOL)$(LIB_EXT) SOCK_LIB= else ifeq
$(INSTRUMENTATION),None INST_FLAGS= INST_INCLUDES= INST_LIBS=
OXF_LIBS=$(OMROOT)/LangCpp/lib/vxoxf$(CPU)$(TOOL)$(LIB_EXT) SOCK_LIB= else echo 'An
invalid Instrumentation $(INSTRUMENTATION) is specified.' exit endif endif endif
#####
Context generated dependencies ##### $OMContextDependencies $OMFileObjPath :
$OMMainImplementationFile $(OBJS) @echo Compiling $OMMainImplementationFile @$(CXX)
$(C++FLAGS) $(ConfigurationCPPCompileSwitches) -o $OMFileObjPath $OMMainImplementationFile
##### #
Predefined linking instructions # # INST_LIBS is included twice to solve bi-directional dependency
between libraries #
#####
$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS) $OMFileObjPath
$OMMakefileName $OMModelLibs @echo Linking and Munching $(TARGET_NAME)$(EXE_EXT)
@$(LINK_CMD) $(LINK_FLAGS) -o $(TARGET_NAME).tmp \ $OMFileObjPath $(OBJS)
$(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \ $(OXF_LIBS) \ $(INST_LIBS) \ $(SOCK_LIB) @
$(RM) $(TARGET_NAME)$(EXE_EXT) ctdt.c ctdt.o @$(NM) $(TARGET_NAME).tmp | $(MUNCH) >
ctdt.c @$(CC) $(CC_ARCH_SPEC) -c ctdt.c @$(LINK_CMD) $OMLinkCommandSet -r4 -o $@
$(TARGET_NAME).tmp ctdt.o @ $(RM) ctdt.c ctdt.o $(TARGET_NAME).tmp
$(TARGET_NAME)$(LIB_EXT) : $(OBJS) $(ADDITIONAL_OBJS) $OMMakefileName @echo Building
library @@ @$(LIB_CMD) $(LIB_FLAGS) $(TARGET_NAME)$(LIB_EXT) $(OBJS)
$(ADDITIONAL_OBJS) clean: cleanall: clean @echo Cleanup $(RM) $OMFileObjPath $(RM)
$(TARGET_NAME)$(LIB_EXT) $(RM) $(TARGET_NAME)$(EXE_EXT) $OMCleanOBJS
$(CLEAN_OBJ_DIR)

```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use

for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the `CPP_CG::<Environment>::MakeExtension` property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## **NullValue**

The `NullValue` property specifies an alternative expression for `NULL` in the generated code.

*Default = NULL*

## **ObjCleanCommand**

The `ObjCleanCommand` property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default = \$(RM) \$OMFileObjPath*

## **ObjectName**

The `ObjectName` property specifies an alternative name for the compiled object file in the generated makefile.

*Default = Empty string*

## **ObjectsDirectory**

The `ObjectsDirectory` property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = Empty string*

## **ObjExtension**

The `ObjExtension` property specifies the extension appended to compiled object files for a given environment.

*Default = .o*

## **OSFileSystemCaseSensitive**

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Cleared*

## OSVersion

Used for the various Wind River environments, the OSVersion property represents the version number of the product. The value of the property is used in the batch files that build applications and framework libraries for these environments. In these batch files, the value of the property is provided as a parameter for both the "wrenv" command and the "make" command.

*Default = 6.8*

## ParseErrorDescript

The ParseErrorDescript property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = [""]([^\:]+)[""][,][ ]line ([0-9]+)[:] (.\*)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = [""]([^\:]+)[""][,][ ]line ([0-9]+)[:]*

## ParseMakeError

The ParseMakeError property is used to define a regular expression that represents the format of make process or linker error messages. This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = (.\*)(make)(.)(Error)*

## ParseSeverityError

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = [""]([^\:]+)[""][,][ ]line ([0-9]+)[\:]*

## **ParseSeverityWarning**

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = [""]([^\:]+)[""][,][ ]line ([0-9]+)[\:] (warning)*

## **PathWhiteSpaceHandling**

For different operating systems, there are different methods for handling spaces in file paths, for example, enclosing the entire path in quotation marks. The PathWhiteSpaceHandling property allows you to specify the method that should be used for a given environment. The possible values are:

- NoHandling - the path should be left as is, with no special handling for spaces
- SurroundingQuotes - the entire path should be enclosed in quotation marks
- BackslashBeforeSpace - spaces in paths should be preceded by backslashes, as is the practice in VxWorks platforms

*Default = BackslashBeforeSpace*

## **QuoteOMROOT**

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Checked*

## **RemoteHost**

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running Rational Rhapsody, whereas the target is the machine running the application. To run remotely, the UseRemoteHost property must be set to True. If UseRemoteHost is True and RemoteHost is blank, the current host name is used for the remote host. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = Empty string*

## **ReusableStatechartSwitches**

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts. See the upgrade history on the support site for information on this change.

*Default = -DOM\_REUSABLE\_STATECHART\_IMPLEMENTATION*

## **SpecExtension**

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*Default = .h*

## **UpdateBuildSettingsInIDE**

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## **UseNewBuildOutputWindow**

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the CG::General::ShowLogViewAfterBuild property to True.

*Default = Checked*

## **UseNonZeroStdInputHandle**

The UseNonZeroStdInputHandle property is a Boolean value that specifies whether to use a non-zero standard input handle.

*Default = Checked*

## **UseRemoteHost**

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Checked*



## UseTemplateTypename

In the declarations for members of a template class, some compilers require that you use precede the member type with the keyword "typename" for types with dependent names. Since support for this keyword varies between compilers, the UseTemplateTypename property is used to specify whether the "typename" keyword should be included in the generated code.

*Default = Cleared*

## VxWorks6diab\_RTP

The VxWorks6diab\_RTP metaclass contains environment settings (Compiler, framework libraries, and so on).

## AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)/LangCpp/osconfig/VxWorks*

## AdditionalReservedWords

The AdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = Empty string*

## BSP

The BSP property specifies the board support package (BSP) for the system. If you need to change the value of the CPU, you can simply reset the value in this property instead of changing the value in the MakeFileContent property.

*Default = PENTIUM*

## BuildArgumentsInIDE

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the CPP\_CG:[environment]:BuildCommandInIDE property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandInIDE

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration.

To change this property, use the Configuration window in the browser - do not change it by using the Properties tab in the Features window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link. The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.

- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*Default = Debug*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\$OMROOT/etc/Executer.exe" "\""\$OMROOT\etc\vx6make.bat" vxbuild.mak buildLibs 6.5  
\"CPU=\$BSP\" \"TOOL=diab\" \"TOOL\_FAMILY=diab\" \"BUILD=\$BuildCommandSet\"  
\"USE\_RTP=TRUE\" \" \"*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## CodeTestSettings

The CodeTestSettings property specifies the compiler command settings. This property supports integration with Applied Microsystems Corporation CodeTest.

*Default = CXX = \$(AMC\_HOME)\bin\ctcxx*

## CompileSwitches

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property.

*Default =*

*\$IgnoreSwitches -I\$OMDefaultSpecificationDirectory -I\$(OMROOT) -I\$(OMROOT)/LangCpp  
-I\$(OMROOT)/LangCpp/oxf -DVxWorks \$(INST\_FLAGS) \$(INCLUDE\_PATH)  
\$OMCPPCompileCommandSet -c*

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The default is as follows:

```
@echo Compiling $OMFileImpPath $(CREATE_OBJ_DIR) @$(CXX) $(C++FLAGS)
$OMFileCPPCompileSwitches -o $OMFileObjPath $OMFileImpPath
```

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*Default = -g*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*Default = Empty string*

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

The default is as follows:

```
$OMFileObjPath : $OMFileImpPath $OMFileSpecPath $OMFileDependencies
```

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: `$OMFileObjPath : $OMFileImpPath "*.bmp" "*.ico" "*.rc2"`

## DuplicateLibsListInMakeFile

The DuplicateLibsListInMakeFile property is a Boolean value that specifies whether Rational Rhapsody should duplicate the libraries list in the generated makefile link command. This property supports linkers that are sensitive to library order in the link command.

*Default = Checked*

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction with an IDE such as Eclipse, the `EnableDebugIntegrationWithIDE` property can be used to specify whether or not the IDE debugger should be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to `True`, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's `exit()` function is called automatically. The system function calls the destructors of framework instances in the order it determines. The `EndApplicationCode` property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an `endApplication()` function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to `endApplication()`, it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The `EntryPoint` property specifies the name of the main program for a given environment.

*Default = main*

If applicable for the metaclass, see also the definition of the `EntryPointDeclarationModifier` property for more information.

## ErrorMessageTokensFormat

The `ErrorMessageTokensFormat`, working with the `ParseErrorMessage` property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. `ErrorMessageTokensFormat` defines the number and location of tokens within the regular expression defined by the `ParseErrorMessage` property. `ErrorMessageTokens` has three parameters, each with an integer value:

- `TotalNumberOfTokens` - The number of tokens in the regular expression
- `FileTokenPosition` - The position of the file name token in the expression
- `LineTokenPosition` - The position of the line number token in the expression

*Default = TotalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2*

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the CPP\_CG::<Environment>::ExeName property plus the value of this property.

*(Default = .vxe)*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the CPP\_CG::<Environment>::ExeExtension property.

*(Default = Blank)*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements. The file inclusions are generated in the makefile.

*Default = \$OMSpecIncludeInElements \$OMImpIncludeInElements*

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile. Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

*Default =*

`$(OMROOT)/LangCpp/lib/vxWebComponents$(CPU)$(RTP_SUFFIX)$(TOOL)$(RHP_LIB_EXT),  
$(OMROOT)/lib/vxWebServices$(CPU)$(RTP_SUFFIX)$(TOOL)$(RHP_LIB_EXT)`

## HasIDEInterface

The HasIDEInterface property is a Boolean value that specifies whether IDE support is enabled. If IDE support is enabled (Checked), the IDEInterfaceDLL property points to an IDE adapter that provides

connection to the IDE. If the property is set to Cleared, IDE support is disabled and IDE services are not attempted. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = Cleared*

## IDEInterfaceDLL

The IDEInterfaceDLL property is a string that points to the IDE adapter DLL. You should not have any reason to modify this property. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = Empty string*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment.

*Default = .cpp*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*Default = include*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\vx6make.bat\" \$makefile \$maketarget 6.2 \$BSP diab"*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## **LibExtension**

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .a*

## **LinkDebug**

The LinkDebug property specifies the special link switches used to link in debug mode.

*Default = Empty string*

## **LinkRelease**

The LinkRelease property specifies the special link switches used to link in release mode.

*Default = Empty string*

## **LinkSwitches**

The LinkSwitches property specifies the standard link switches used to link in any mode.

*Default = \$OMLinkCommandSet*

## **MakeExtension**

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the CPP\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*(Default = .mak)*



## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration.

```
Default = ##### Target type (Debug/Release) #####
#####
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease
ConfigurationCPPCompileSwitches=$OMReusableStatechartSwitches
$OMConfigurationCPPCompileSwitches #####
##### Definitions and flags #####
##### CPU = $BSP TOOL = diab
RHP_LIB_EXT = $OMLibExt RTP_SUFFIX = _RTP_LINK_CMD=$(CXX) $(CC_ARCH_SPEC) -Xansi
-Xforce-declarations -Xmake-dependency=0xd LINK_FLAGS=$OMConfigurationLinkSwitches
RTP_LIBS = -L$(WIND_USR_LIB_PATH) -lstdc++ $OMCodeTestSettings INCLUDE_QUALIFIER=-I
LIB_CMD=$(AR) LIB_FLAGS=$(ARFLAGS)
##### Context generated
macros ##### $OMContextMacros OBJ_DIR=$OMObjectsDir ifeq ($(OBJ_DIR),)
CREATE_OBJ_DIR= CLEAN_OBJ_DIR= else CREATE_OBJ_DIR= if not exist $(OBJ_DIR) mkdir
$(OBJ_DIR) CLEAN_OBJ_DIR= if exist $(OBJ_DIR) rmdir $(OBJ_DIR) endif
##### Predefined macros
##### $(OBJS) : $(INST_LIBS) $(OXF_LIBS) ifeq ($(INSTRUMENTATION),Animation)
INST_FLAGS=-DOMANIMATOR -DUSE_Iostream INST_INCLUDES=-I$(OMROOT)/LangCpp/aom
-I$(OMROOT)/LangCpp/tom INST_LIBS=
$(OMROOT)/LangCpp/lib/vxaomanim$(CPU)$(RTP_SUFFIX)$(TOOL)$(RHP_LIB_EXT)
OXF_LIBS=$(OMROOT)/LangCpp/lib/vxoxfinst$(CPU)$(RTP_SUFFIX)$(TOOL)$(RHP_LIB_EXT)
$(OMROOT)/LangCpp/lib/vxomcomappl$(CPU)$(RTP_SUFFIX)$(TOOL)$(RHP_LIB_EXT)
SOCK_LIB=$(RTP_LIBS) else ifeq ($(INSTRUMENTATION),Tracing) INST_FLAGS=-DOMTRACER
-DUSE_Iostream INST_INCLUDES=-I$(OMROOT)/LangCpp/aom -I$(OMROOT)/LangCpp/tom
INST_LIBS=$(OMROOT)/LangCpp/lib/vxtomtrace$(CPU)$(RTP_SUFFIX)$(TOOL)$(RHP_LIB_EXT)
$(OMROOT)/LangCpp/lib/vxaomtrace$(CPU)$(RTP_SUFFIX)$(TOOL)$(RHP_LIB_EXT) OXF_LIBS=
$(OMROOT)/LangCpp/lib/vxoxfinst$(CPU)$(RTP_SUFFIX)$(TOOL)$(RHP_LIB_EXT)
$(OMROOT)/LangCpp/lib/vxomcomappl$(CPU)$(RTP_SUFFIX)$(TOOL)$(RHP_LIB_EXT)
SOCK_LIB=$(RTP_LIBS) else ifeq ($(INSTRUMENTATION),None) INST_FLAGS= INST_INCLUDES=
INST_LIBS=
OXF_LIBS=$(OMROOT)/LangCpp/lib/vxoxf$(CPU)$(RTP_SUFFIX)$(TOOL)$(RHP_LIB_EXT)
```

```

SOCK_LIB=$(RTP_LIBS) else echo 'An invalid Instrumentation $(INSTRUMENTATION) is specified.'
exit endif endif endif #####
##### Context generated dependencies ##### $OMContextDependencies
$OMFileObjPath : $OMMainImplementationFile $(OBJS) @echo Compiling
$OMMainImplementationFile @$(CXX) $(C++FLAGS) $(ConfigurationCPPCompileSwitches) -o
$OMFileObjPath $OMMainImplementationFile
##### #
Predefined linking instructions # # INST_LIBS is included twice to solve bi-directional dependency
between libraries #
#####
$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS) $OMFileObjPath
$OMMakefileName $OMModelLibs @echo Linking $(TARGET_NAME)$(EXE_EXT) @$(LINK_CMD)
$(LINK_FLAGS) $OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \
$(OXF_LIBS) \ $(INST_LIBS) \ $(SOCK_LIB) -o $$@ $(TARGET_NAME)$(RHP_LIB_EXT) : $(OBJS)
$(ADDITIONAL_OBJS) $OMMakefileName @echo Building library $$@ @$(LIB_CMD) $(LIB_FLAGS)
$(TARGET_NAME)$(RHP_LIB_EXT) $(OBJS) $(ADDITIONAL_OBJS) clean: cleanall: clean @echo
Cleanup $(RM) $OMFileObjPath $(RM) $(TARGET_NAME)$(RHP_LIB_EXT) $(RM)
$(TARGET_NAME)$(EXE_EXT) $OMCleanOBJS $(CLEAN_OBJ_DIR) include
$(WIND_USR)/make/rules.rtp

```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the CPP\_CG::<Environment>::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## ObjCleanCommand

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default = \$(RM) \$OMFileObjPath*

## ObjectName

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = Empty string*

## ObjectsDirectory

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = Empty string*

## ObjExtension

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*Default = .o*

## OSFileSystemCaseSensitive

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Cleared*

## OSVersion

Used for the various Wind River environments, the OSVersion property represents the version number of the product. The value of the property is used in the batch files that build applications and framework libraries for these environments. In these batch files, the value of the property is provided as a parameter for both the "wrenv" command and the "make" command.

*Default = 6.8*

## ParseErrorDescript

The ParseErrorDescript property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = ["](^[^:]+)["], [line ([0-9]+):] (.\*)*

## ParseErrorMessage

The `ParseErrorMessage` property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning). Along with the `ErrorMessageTokensFormat` property, `ParseErrorMessage` specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = [""]([^\:]+)[""][,][ ]line ([0-9]+)[\:]*

## ParseMakeError

The `ParseMakeError` property is used to define a regular expression that represents the format of make process or linker error messages. This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = (.\*)(make)(.)(Error)*

## ParseSeverityError

The `ParseSeverityError` property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = [""]([^\:]+)[""][,][ ]line ([0-9]+)[\:]*

## ParseSeverityWarning

The `ParseSeverityWarning` property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = [""]([^\:]+)[""][,][ ]line ([0-9]+)[\:] (warning)*

## PathWhiteSpaceHandling

For different operating systems, there are different methods for handling spaces in file paths, for example, enclosing the entire path in quotation marks. The `PathWhiteSpaceHandling` property allows you to specify the method that should be used for a given environment. The possible values are:

- `NoHandling` - the path should be left as is, with no special handling for spaces
- `SurroundingQuotes` - the entire path should be enclosed in quotation marks
- `BackslashBeforeSpace` - spaces in paths should be preceded by backslashes, as is the practice in VxWorks platforms

*Default = BackslashBeforeSpace*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Checked*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running Rational Rhapsody, whereas the target is the machine running the application. To run remotely, the UseRemoteHost property must be set to True. If UseRemoteHost is True and RemoteHost is blank, the current host name is used for the remote host. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = Empty string*

## ReusableStatechartSwitches

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts. See the upgrade history on the support site for information on this change.

*Default = -DOM\_REUSABLE\_STATECHART\_IMPLEMENTATION*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*Default = .h*

## UpdateBuildSettingsInIDE

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## UseNewBuildOutputWindow

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the `CG::General::ShowLogViewAfterBuild` property to `True`.

*Default = Checked*

## UseNonZeroStdInputHandle

The `UseNonZeroStdInputHandle` property is a Boolean value that specifies whether to use a non-zero standard input handle.

*Default = Checked*

## UseRemoteHost

The `UseRemoteHost` property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Checked*

## UseTemplateName

In the declarations for members of a template class, some compilers require that you use precede the member type with the keyword "typename" for types with dependent names. Since support for this keyword varies between compilers, the `UseTemplateName` property is used to specify whether the "typename" keyword should be included in the generated code.

*Default = Cleared*

## VxWorks6gnu

The `VxWorks6gnu` metaclass contains environment settings (Compiler, framework libraries, and so on) for VxWorks 6.x, by using the GNU compiler (GCC).

## AdaptorSearchPath

The `AdaptorSearchPath` property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adaptor builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the `AdaptorSearchPath` property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)/LangCpp/osconfig/VxWorks*

## **AdditionalReservedWords**

The `AdditionalReservedWords` property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = Empty string*

## **BSP**

The `BSP` property specifies the board support package (BSP) for the system. If you need to change the value of the CPU, you can simply reset the value in this property instead of changing the value in the `MakeFileContent` property.

*Default = PENTIUM*

## **BuildArgumentsInIDE**

The `BuildArgumentsInIDE` property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the `CPP_CG::[environment]::BuildCommandInIDE` property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the `BuildInIDE` property is set to `True`.

*Default = Blank*

## **BuildCommandInIDE**

The `BuildCommandInIDE` property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build

command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration.

To change this property, use the Configuration window in the browser - do not change it by using the Properties tab in the Features window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link. The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*Default = Debug*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\$OMROOT/etc/Executer.exe" "\""\$OMROOT\etc\vx6make.bat" vxbuild.mak buildLibs 6.5  
\"CPU=\$BSP\" \"TOOL=gnu\" \"TOOL\_FAMILY=gnu\" \"BUILD=\$BuildCommandSet\" \" \"*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*



## CodeTestSettings

The CodeTestSettings property specifies the compiler command settings. This property supports integration with Applied Microsystems Corporation CodeTest.

*Default = CXX = \$(AMC\_HOME)\bin\ctcxx*

## CompileSwitches

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property.

*Default =*

*-I\$OMDefaultSpecificationDirectory -fno-merge-templates -I\$(OMROOT) -I\$(OMROOT)/LangCpp  
-I\$(OMROOT)/LangCpp/oxf -DVxWorks \$(INST\_FLAGS) \$(INCLUDE\_PATH)  
\$OMCPPCompileCommandSet -c*

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The default is as follows:

*@echo Compiling \$OMFileImpPath \$(CREATE\_OBJ\_DIR) @\$(CXX) \$(C++FLAGS)  
\$OMFileCPPCompileSwitches -o \$OMFileObjPath \$OMFileImpPath*

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*Default = -O0 -g*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*Default = Empty string*

## DependencyRule

The `DependencyRule` property specifies how file dependencies for a configuration are generated in the makefile.

The default is as follows:

```
$OMFileObjPath : $OMFileImpPath $OMFileSpecPath $OMFileDependencies
```

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: `$OMFileObjPath : $OMFileImpPath "*.bmp" "*.ico" "*.rc2"`

## DuplicateLibsListInMakeFile

The `DuplicateLibsListInMakeFile` property is a Boolean value that specifies whether Rational Rhapsody should duplicate the libraries list in the generated makefile link command. This property supports linkers that are sensitive to library order in the link command.

*Default = Checked*

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction with an IDE such as Eclipse, the `EnableDebugIntegrationWithIDE` property can be used to specify whether or not the IDE debugger should be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to `True`, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's `exit()` function is called automatically. The system function calls the destructors of framework instances in the order it determines. The `EndApplicationCode` property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an `endApplication()` function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to `endApplication()`, it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The `EntryPoint` property specifies the name of the main program for a given environment.

*Default = vxmain*

If applicable for the metaclass, see also the definition of the EntryPointDeclarationModifier property for more information.

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens has three parameters, each with an integer value:

- TotalNumberOfTokens - The number of tokens in the regular expression
- FileTokenPosition - The position of the file name token in the expression
- LineTokenPosition - The position of the line number token in the expression

*Default = TotalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2*

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the CPP\_CG::<Environment>::ExeName property plus the value of this property.

*(Default = .out)*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the CPP\_CG::<Environment>::ExeExtension property.

*(Default = Blank)*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files

should be included in model elements. The file inclusions are generated in the makefile.

*Default = \$OMSpecIncludeInElements \$OMImpIncludeInElements*

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile. Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

*Default =*

`$(OMROOT)/LangCpp/lib/vxWebComponents$(CPU)$(TOOL)$(LIB_EXT),  
$(OMROOT)/lib/vxWebServices$(CPU)$(TOOL)$(LIB_EXT)`

## HasIDEInterface

The HasIDEInterface property is a Boolean value that specifies whether IDE support is enabled. If IDE support is enabled (Checked), the IDEInterfaceDLL property points to an IDE adapter that provides connection to the IDE. If the property is set to Cleared, IDE support is disabled and IDE services are not attempted. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = Checked*

## IDEInterfaceDLL

The IDEInterfaceDLL property is a string that points to the IDE adapter DLL. You should not have any reason to modify this property. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = \$OMROOT/DLLs/WorkbenchDebuggerIDE.dll*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment. The default values are as follows:

*Default = .cpp*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*Default = include*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\vx6make.bat\" \$makefile \$maketarget"*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .a*

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

*Default = -g*

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

*Default = Empty string*

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

*Default = \$OMLinkCommandSet*

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the CPP\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*(Default = .mak)*

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration.

```
Default = ##### Target type (Debug/Release) #####
#####
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease
ConfigurationCPPCompileSwitches=$OMReusableStatechartSwitches
$OMConfigurationCPPCompileSwitches #####
##### Definitions and flags #####
##### CPU = $BSP TOOL = gnu include
$(WIND_BASE)/target/h/make/defs.bsp .cpp.o : @ $(RM) $@ $(CXX) $(C++FLAGS)
```

```

$(OPTION_OBJECT_ONLY) $< $OMCodeTestSettings INCLUDE_QUALIFIER=-I LIB_CMD=$(AR)
LINK_CMD=$(LD) LIB_FLAGS=$(ARFLAGS) LINK_FLAGS=$OMConfigurationLinkSwitches -r
##### Context generated
macros ##### $OMContextMacros OBJ_DIR=$OMObjectsDir ifeq ($(OBJ_DIR),)
CREATE_OBJ_DIR= CLEAN_OBJ_DIR= else CREATE_OBJ_DIR= if not exist $(OBJ_DIR) mkdir
$(OBJ_DIR) CLEAN_OBJ_DIR= if exist $(OBJ_DIR) rmdir $(OBJ_DIR) endif
##### Predefined macros
##### $(OBJS) : $(INST_LIBS) $(OXF_LIBS) ifeq ($(INSTRUMENTATION),Animation)
INST_FLAGS=-DOMANIMATOR -DUSE_Iostream INST_INCLUDES=-I$(OMROOT)/LangCpp/aom
-I$(OMROOT)/LangCpp/tom INST_LIBS=
$(OMROOT)/LangCpp/lib/vxaomanim$(CPU)$(TOOL)$(LIB_EXT)
OXF_LIBS=$(OMROOT)/LangCpp/lib/vxoxfinst$(CPU)$(TOOL)$(LIB_EXT)
$(OMROOT)/LangCpp/lib/vxomcomappl$(CPU)$(TOOL)$(LIB_EXT) SOCK_LIB= else ifeq
$(INSTRUMENTATION),Tracing INST_FLAGS=-DOMTRACER -DUSE_Iostream
INST_INCLUDES=-I$(OMROOT)/LangCpp/aom -I$(OMROOT)/LangCpp/tom
INST_LIBS=$(OMROOT)/LangCpp/lib/vxtomtrace$(CPU)$(TOOL)$(LIB_EXT)
$(OMROOT)/LangCpp/lib/vxaomtrace$(CPU)$(TOOL)$(LIB_EXT) OXF_LIBS=
$(OMROOT)/LangCpp/lib/vxoxfinst$(CPU)$(TOOL)$(LIB_EXT)
$(OMROOT)/LangCpp/lib/vxomcomappl$(CPU)$(TOOL)$(LIB_EXT) SOCK_LIB= else ifeq
$(INSTRUMENTATION),None INST_FLAGS= INST_INCLUDES= INST_LIBS=
OXF_LIBS=$(OMROOT)/LangCpp/lib/vxoxf$(CPU)$(TOOL)$(LIB_EXT) SOCK_LIB= else echo 'An
invalid Instrumentation $(INSTRUMENTATION) is specified.' exit endif endif endif
#####
Context generated dependencies ##### $OMContextDependencies $OMFileObjPath :
$OMMainImplementationFile $(OBJS) @echo Compiling $OMMainImplementationFile @$(CXX)
$(C++FLAGS) $(ConfigurationCPPCompileSwitches) -o $OMFileObjPath $OMMainImplementationFile
#####
Predefined linking instructions ##### $(TARGET_NAME)$(EXE_EXT): $(OBJS)
$(ADDITIONAL_OBJS) $OMFileObjPath $OMMakefileName $OMModelLibs @echo Linking and
Munching $(TARGET_NAME)$(EXE_EXT) @$(LINK_CMD) $(LINK_FLAGS) -o
$(TARGET_NAME).tmp \ $OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \
$(OXF_LIBS) \ $(INST_LIBS) \ $(SOCK_LIB) @ $(RM) $(TARGET_NAME)$(EXE_EXT) ctdt.c ctdt.o
@$(NM) $(TARGET_NAME).tmp | $(MUNCH) > ctdt.c @$(CC) $(CC_ARCH_SPEC) -c ctdt.c
@$(LINK_CMD) -r $OMLinkCommandSet -o @@ $(TARGET_NAME).tmp ctdt.o @ $(RM) ctdt.c ctdt.o
$(TARGET_NAME).tmp $(TARGET_NAME)$(LIB_EXT) : $(OBJS) $(ADDITIONAL_OBJS)
$OMMakefileName @echo Building library @@ @$(LIB_CMD) $(LIB_FLAGS)
$(TARGET_NAME)$(LIB_EXT) $(OBJS) $(ADDITIONAL_OBJS) clean: cleanall: clean @echo Cleanup
$(RM) $OMFileObjPath $(RM) $(TARGET_NAME)$(LIB_EXT) $(RM) $(TARGET_NAME)$(EXE_EXT)
$OMCleanOBJS $(CLEAN_OBJ_DIR)

```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the CPP\_CG::<Environment>::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

(Default = Blank)

## **NullValue**

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## **ObjCleanCommand**

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default = \$(RM) \$OMFileObjPath*

## **ObjectName**

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = Empty string*

## **ObjectsDirectory**

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = Empty string*

## **ObjExtension**

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*Default = .o*

## **OSFileSystemCaseSensitive**

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Cleared*

## **OSVersion**



Used for the various Wind River environments, the OSVersion property represents the version number of the product. The value of the property is used in the batch files that build applications and framework libraries for these environments. In these batch files, the value of the property is provided as a parameter for both the "wrenv" command and the "make" command.

*Default = 6.8*

## ParseErrorDescript

The ParseErrorDescript property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = ([^:]+)[:]( [0-9]+)[:] (error/warning)[:] (.\*)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = ([^:]+)[:]( [0-9]+)[:]*

## ParseMakeError

The ParseMakeError property is used to define a regular expression that represents the format of make process or linker error messages. This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = (.\*)(make)(.\*)(Error)*

## ParseSeverityError

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^:]+)[:]( [0-9]+)[:] (error)*

## ParseSeverityWarning

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^:]+):([0-9]+):([0-9]+) (warning)*

## PathWhiteSpaceHandling

For different operating systems, there are different methods for handling spaces in file paths, for example, enclosing the entire path in quotation marks. The PathWhiteSpaceHandling property allows you to specify the method that should be used for a given environment. The possible values are:

- NoHandling - the path should be left as is, with no special handling for spaces
- SurroundingQuotes - the entire path should be enclosed in quotation marks
- BackslashBeforeSpace - spaces in paths should be preceded by backslashes, as is the practice in VxWorks platforms

*Default = BackslashBeforeSpace*

## ProcessorArchitecture

The ProcessorArchitecture property is used to specify whether the application is being developed for 32-bit or 64-bit target systems. The value of the property is used in a number of places, such as the command used for building the Rhapsody framework, the content of generated makefiles, and the names of generated libraries.

Verify that ProcessorArchitecture is set to the appropriate value for the target systems. The possible values are x86 (for 32-bit) and x64 (for 64-bit).

Once you have set the property to the appropriate value, rebuild the Rhapsody framework, and generate and build your application.

*Default = x86*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Checked*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running Rational Rhapsody, whereas the target is the machine running the application.

To run remotely, the UseRemoteHost property must be set to True. If UseRemoteHost is True and RemoteHost is blank, the current host name is used for the remote host. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = Empty string*

## **ReusableStatechartSwitches**

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts. See the upgrade history on the support site for information on this change.

*Default = -DOM\_REUSABLE\_STATECHART\_IMPLEMENTATION*

## **SpecExtension**

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*Default = .h*

## **UpdateBuildSettingsInIDE**

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## **UseNewBuildOutputWindow**

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the CG::General::ShowLogViewAfterBuild property to True.

*Default = Checked*

## **UseNonZeroStdInputHandle**

The UseNonZeroStdInputHandle property is a Boolean value that specifies whether to use a non-zero standard input handle.

*Default = Checked*

## UseRemoteHost

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Checked*

## UseTemplateName

In the declarations for members of a template class, some compilers require that you use precede the member type with the keyword "typename" for types with dependent names. Since support for this keyword varies between compilers, the UseTemplateName property is used to specify whether the "typename" keyword should be included in the generated code.

*Default = Cleared*

## VxWorks6gnu\_RTP

The VxWorks6gnu\_RTP metaclass contains environment settings (Compiler, framework libraries, and so on).

## AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)/LangCpp/osconfig/VxWorks*

## AdditionalReservedWords

The AdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = Empty string*

## **BSP**

The BSP property specifies the board support package (BSP) for the system. If you need to change the value of the CPU, you can simply reset the value in this property instead of changing the value in the MakeFileContent property.

*Default = PENTIUM*

## **BuildArgumentsInIDE**

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the CPP\_CG::[environment]::BuildCommandInIDE property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## **BuildCommandInIDE**

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## **BuildCommandSet**

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration.

To change this property, use the Configuration window in the browser - do not change it by using the Properties tab in the Features window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link. The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*Default = Debug*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\$OMROOT/etc/Executer.exe" "\""\$OMROOT\etc\vx6make.bat" vxbuild.mak buildLibs 6.5  
\"CPU=\$BSP\" \"TOOL=gnu\" \"TOOL\_FAMILY=gnu\" \"BUILD=\$BuildCommandSet\"  
\"USE\_RTP=TRUE\" \" \"*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## CodeTestSettings

The CodeTestSettings property specifies the compiler command settings. This property supports integration with Applied Microsystems Corporation CodeTest.

*Default = CXX = \$(AMC\_HOME)\bin\ctcxx*

## CompileSwitches

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property.

*Default =*

```
-I$OMDefaultSpecificationDirectory -fno-merge-templates -I$(OMROOT) -I$(OMROOT)/LangCpp  
-I$(OMROOT)/LangCpp/oxf -DVxWorks $(INST_FLAGS) $(INCLUDE_PATH)  
$OMCPPCompileCommandSet -c
```

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The default is as follows:

```
@echo Compiling $OMFileImpPath $(CREATE_OBJ_DIR) @$(CXX) $(C++FLAGS)  
$OMFileCPPCompileSwitches -o $OMFileObjPath $OMFileImpPath
```

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*Default = -O0 -g*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*Default = Empty string*

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

The default is as follows:

```
$OMFileObjPath : $OMFileImpPath $OMFileSpecPath $OMFileDependencies
```

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: `$OMFileObjPath : $OMFileImpPath "*.bmp" "*.ico" "*.rc2"`

## DuplicateLibsListInMakeFile

The DuplicateLibsListInMakeFile property is a Boolean value that specifies whether Rational Rhapsody should duplicate the libraries list in the generated makefile link command. This property supports linkers that are sensitive to library order in the link command.

*Default = Checked*

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction with an IDE such as Eclipse, the EnableDebugIntegrationWithIDE property can be used to specify whether or not the IDE debugger should be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to True, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's exit() function is called automatically. The system function calls the destructors of framework instances in the order it determines. The EndApplicationCode property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an endApplication() function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to endApplication(), it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The EntryPoint property specifies the name of the main program for a given environment.

*Default = main*

If applicable for the metaclass, see also the definition of the EntryPointDeclarationModifier property for more information.

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens has three parameters, each with an integer value:

- TotalNumberOfTokens - The number of tokens in the regular expression



- FileTokenPosition - The position of the file name token in the expression
- LineTokenPosition - The position of the line number token in the expression

*Default = ToTalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2*

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the CPP\_CG::<Environment>::ExeName property plus the value of this property.

*(Default = .vxe)*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the CPP\_CG::<Environment>::ExeExtension property.

*(Default = Blank)*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements. The file inclusions are generated in the makefile.

*Default = \$OMSpecIncludeInElements \$OMImpIncludeInElements*

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile. Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

*Default =*

`$(OMROOT)/LangCpp/lib/vxWebComponents$(CPU)$(RTP_SUFFIX)$(TOOL)$(RHP_LIB_EXT),  
$(OMROOT)/lib/vxWebServices$(CPU)$(RTP_SUFFIX)$(TOOL)$(RHP_LIB_EXT)`

## HasIDEInterface

The HasIDEInterface property is a Boolean value that specifies whether IDE support is enabled. If IDE support is enabled (Checked), the IDEInterfaceDLL property points to an IDE adapter that provides connection to the IDE. If the property is set to Cleared, IDE support is disabled and IDE services are not attempted. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = Cleared*

## IDEInterfaceDLL

The IDEInterfaceDLL property is a string that points to the IDE adapter DLL. You should not have any reason to modify this property. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = Empty string*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment. The default values are as follows:

*Default = .cpp*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*Default = include*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

"\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\vx6make.bat\" \$makefile \$maketarget 6.2 \$BSP gnu"

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\vx6make.bat\" \$makefile \$maketarget 6.2 \$BSP gnu"*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .a*

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

*Default = -g*

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

*Default = Empty string*

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

*Default = \$OMLinkCommandSet -MD -MP*

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the

makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the CPP\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

(Default = .mak)

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration.

```
Default = ##### Target type (Debug/Release) #####
#####
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease
ConfigurationCPPCompileSwitches=$OMReusableStatechartSwitches
$OMConfigurationCPPCompileSwitches #####
##### Definitions and flags #####
##### CPU = $BSP TOOL = gnu RHP_LIB_EXT
= $OMLibExt RTP_SUFFIX = _RTP_ RTP_LIBS = -L$(WIND_USR_LIB_PATH) -lstdc++
INCLUDE_QUALIFIER=-I LIB_CMD=$(AR) LIB_FLAGS=$(ARFLAGS) LINK_CMD=$(CXX)
$(CC_ARCH_SPEC) LINK_FLAGS=$OMConfigurationLinkSwitches $OMCodeTestSettings
##### Context generated
macros ##### $OMContextMacros OBJ_DIR=$OMObjectsDir ifeq ($(OBJ_DIR),)
CREATE_OBJ_DIR= CLEAN_OBJ_DIR= else CREATE_OBJ_DIR= if not exist $(OBJ_DIR) mkdir
$(OBJ_DIR) CLEAN_OBJ_DIR= if exist $(OBJ_DIR) rmdir $(OBJ_DIR) endif
##### Predefined macros
##### $(OBJJS) : $(INST_LIBS) $(OXF_LIBS) ifeq ($(INSTRUMENTATION),Animation)
INST_FLAGS=-DOMANIMATOR -DUSE_IOSTREAM INST_INCLUDES=-I$(OMROOT)/LangCpp/aom
-I$(OMROOT)/LangCpp/tom INST_LIBS=
$(OMROOT)/LangCpp/lib/vxaomanim$(CPU)$(RTP_SUFFIX)$(TOOL)$(RHP_LIB_EXT)
OXF_LIBS=$(OMROOT)/LangCpp/lib/vxoxfinst$(CPU)$(RTP_SUFFIX)$(TOOL)$(RHP_LIB_EXT)
$(OMROOT)/LangCpp/lib/vxomcomappl$(CPU)$(RTP_SUFFIX)$(TOOL)$(RHP_LIB_EXT)
SOCK_LIB=$(RTP_LIBS) else ifeq ($(INSTRUMENTATION),Tracing) INST_FLAGS=-DOMTRACER
```

```

-DUSE_Iostream INST_INCLUDES=-I$(OMROOT)/LangCpp/aom -I$(OMROOT)/LangCpp/tom
INST_LIBS=$(OMROOT)/LangCpp/lib/vxomtrace$(CPU)$(RTP_SUFFIX)$(TOOL)$(RHP_LIB_EXT)
$(OMROOT)/LangCpp/lib/vxaomtrace$(CPU)$(RTP_SUFFIX)$(TOOL)$(RHP_LIB_EXT) OXF_LIBS=
$(OMROOT)/LangCpp/lib/vxoxfinst$(CPU)$(RTP_SUFFIX)$(TOOL)$(RHP_LIB_EXT)
$(OMROOT)/LangCpp/lib/vxomcomappl$(CPU)$(RTP_SUFFIX)$(TOOL)$(RHP_LIB_EXT)
SOCK_LIB=$(RTP_LIBS) else ifeq $(INSTRUMENTATION),None INST_FLAGS= INST_INCLUDES=
INST_LIBS=
OXF_LIBS=$(OMROOT)/LangCpp/lib/vxoxf$(CPU)$(RTP_SUFFIX)$(TOOL)$(RHP_LIB_EXT)
SOCK_LIB=$(RTP_LIBS) else echo 'An invalid Instrumentation $(INSTRUMENTATION) is specified.'
exit endif endif endif #####
##### Context generated dependencies ##### $OMContextDependencies
$OMFileObjPath : $OMMainImplementationFile $(OBJS) @echo Compiling
$OMMainImplementationFile @$(CXX) $(C++FLAGS) $(ConfigurationCPPCompileSwitches) -o
$OMFileObjPath $OMMainImplementationFile
#####
Predefined linking instructions ##### $(TARGET_NAME)$(EXE_EXT): $(OBJS)
$(ADDITIONAL_OBJS) $OMFileObjPath $OMMakefileName $OMModelLibs @echo Linking
$(TARGET_NAME)$(EXE_EXT) @$(LINK_CMD) $(LINK_FLAGS) $(C++FLAGS) $OMFileObjPath
$(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \ $(OXF_LIBS) \ $(INST_LIBS) \
$(SOCK_LIB) -o @ $(TARGET_NAME)$(RHP_LIB_EXT) : $(OBJS) $(ADDITIONAL_OBJS)
$OMMakefileName @echo Building library @@ @$(LIB_CMD) $(LIB_FLAGS)
$(TARGET_NAME)$(RHP_LIB_EXT) $(OBJS) $(ADDITIONAL_OBJS) clean: cleanall: clean @echo
Cleanup $(RM) $OMFileObjPath $(RM) $(TARGET_NAME)$(RHP_LIB_EXT) $(RM)
$(TARGET_NAME)$(EXE_EXT) $OMCleanOBJS $(CLEAN_OBJ_DIR) include
$(WIND_USR)/make/rules.rtp

```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the CPP\_CG::<Environment>::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## ObjCleanCommand

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default = \$(RM) \$OMFileObjPath*

## **ObjectName**

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = Empty string*

## **ObjectsDirectory**

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = Empty string*

## **ObjExtension**

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*Default = .o*

## **OSFileSystemCaseSensitive**

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Cleared*

## **OSVersion**

Used for the various Wind River environments, the OSVersion property represents the version number of the product. The value of the property is used in the batch files that build applications and framework libraries for these environments. In these batch files, the value of the property is provided as a parameter for both the "wrenv" command and the "make" command.

*Default = 6.8*

## **ParseErrorDescript**

The ParseErrorDescript property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = ([^:]+):([0-9]+): (error/warning): (.\*)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = ([^:]+):([0-9]+):*

## ParseMakeError

The ParseMakeError property is used to define a regular expression that represents the format of make process or linker error messages. This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = (.\*)(make)(.)(Error)*

## ParseSeverityError

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^:]+):([0-9]+): (error)*

## ParseSeverityWarning

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^:]+):([0-9]+): (warning)*

## PathWhiteSpaceHandling

For different operating systems, there are different methods for handling spaces in file paths, for example, enclosing the entire path in quotation marks. The PathWhiteSpaceHandling property allows you to specify the method that should be used for a given environment. The possible values are:

- NoHandling - the path should be left as is, with no special handling for spaces
- SurroundingQuotes - the entire path should be enclosed in quotation marks
- BackslashBeforeSpace - spaces in paths should be preceded by backslashes, as is the practice in VxWorks platforms

*Default = BackslashBeforeSpace*

## ProcessorArchitecture

The ProcessorArchitecture property is used to specify whether the application is being developed for 32-bit or 64-bit target systems. The value of the property is used in a number of places, such as the command used for building the Rhapsody framework, the content of generated makefiles, and the names of generated libraries.

Verify that ProcessorArchitecture is set to the appropriate value for the target systems. The possible values are x86 (for 32-bit) and x64 (for 64-bit).

Once you have set the property to the appropriate value, rebuild the Rhapsody framework, and generate and build your application.

*Default = x86*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Checked*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running Rational Rhapsody, whereas the target is the machine running the application. To run remotely, the UseRemoteHost property must be set to True. If UseRemoteHost is True and RemoteHost is blank, the current host name is used for the remote host. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = Empty string*

## ReusableStatechartSwitches

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts. See the upgrade history on the support site for information on this change.

*Default = -DOM\_REUSABLE\_STATECHART\_IMPLEMENTATION*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.



*Default = .h*

## **UpdateBuildSettingsInIDE**

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## **UseNewBuildOutputWindow**

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the CG::General::ShowLogViewAfterBuild property to True.

*Default = Checked*

## **UseNonZeroStdInputHandle**

The UseNonZeroStdInputHandle property is a Boolean value that specifies whether to use a non-zero standard input handle.

*Default = Checked*

## **UseRemoteHost**

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Checked*

## **UseTemplateName**

In the declarations for members of a template class, some compilers require that you use precede the member type with the keyword "typename" for types with dependent names. Since support for this keyword varies between compilers, the UseTemplateName property is used to specify whether the "typename" keyword should be included in the generated code.

*Default = Cleared*

## WorkbenchManaged

The WorkbenchManaged metaclass contains environment settings (Compiler, framework libraries, and so on) for WorkbenchManaged compiler.

### AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)/LangCpp/osconfig/VxWorks*

### AdditionalReservedWords

The AdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = Empty string*

### AutoAttachToIDEDebugger

The AutoAttachToIDEDebugger property is used to specify that you would like the Workbench debugger to be automatically synchronized with the Rational Rhapsody animation. If for some reason you do not want automatic synchronization, for example, if you prefer to manually connect to the relevant target server, then you can set the value of this property to False.

*Default = Checked*

## BSP

The BSP property specifies the board support package (BSP) for the system. If you need to change the value of the CPU, you can simply reset the value in this property instead of changing the value in the MakeFileContent property.

*Default = PENTIUM*

## BuildArgumentsInIDE

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the CPP\_CG::[environment]::BuildCommandInIDE property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandInIDE

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration.

To change this property, use the Configuration window in the browser - do not change it by using the Properties tab in the Features window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link. The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.

- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*Default = Debug*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\vx6make.bat" vxbuild.mak buildLibs 6.5  
\"CPU=\$BSP\" \"TOOL=\$Tool\" \"TOOL\_FAMILY=\$Tool\" \"BUILD=\$BuildCommandSet\" \" \"*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Checked*

## CodeTestSettings

The CodeTestSettings property specifies the compiler command settings. This property supports integration with Applied Microsystems Corporation CodeTest.

*Default = CXX = \$(AMC\_HOME)\bin\ctcxx*

## CompileSwitches

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property.

*Default = Empty MultiLine*

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The default is as follows:

```
@echo Compiling $OMFileImpPath $(CREATE_OBJ_DIR) @$(CXX) $(C++FLAGS)
$OMFileCPPCompileSwitches -o $OMFileObjPath $OMFileImpPath
```

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*Default = -g*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*Default = Empty string*

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

The default is as follows:

```
$OMFileObjPath : $OMFileImpPath $OMFileSpecPath $OMFileDependencies
```

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: `$OMFileObjPath : $OMFileImpPath "*.bmp" "*.ico" "*.rc2"`

## DuplicateLibsListInMakeFile

The DuplicateLibsListInMakeFile property is a Boolean value that specifies whether Rational Rhapsody should duplicate the libraries list in the generated makefile link command. This property supports linkers that are sensitive to library order in the link command.

*Default = Checked*

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction with an IDE such as Eclipse, the EnableDebugIntegrationWithIDE property can be used to specify whether or not the IDE debugger should be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to True, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's `exit()` function is called automatically. The system function calls the destructors of framework instances in the order it determines. The `EndApplicationCode` property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an `endApplication()` function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to `endApplication()`, it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The `EntryPoint` property specifies the name of the main program for a given environment.

*Default = vxmain*

If applicable for the metaclass, see also the definition of the `EntryPointDeclarationModifier` property for more information.

## ErrorMessageTokensFormat

The `ErrorMessageTokensFormat`, working with the `ParseErrorMessage` property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. `ErrorMessageTokensFormat` defines the number and location of tokens within the regular expression defined by the `ParseErrorMessage` property. `ErrorMessageTokens` has three parameters, each with an integer value:

- `TotalNumberOfTokens` - The number of tokens in the regular expression
- `FileTokenPosition` - The position of the file name token in the expression
- `LineTokenPosition` - The position of the line number token in the expression

*Default = TotalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2*

## ExeExtension

The `ExeExtension` property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the `CPP_CG::<Environment>::ExeName` property plus the value of this property.

*(Default = .out)*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the CPP\_CG::<Environment>::ExeExtension property.

*(Default = Blank)*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements. The file inclusions are generated in the makefile.

*Default = \$OMSpecIncludeInElements \$OMImpIncludeInElements*

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile. Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

*Default =*

*\$(OMROOT)/LangCpp/lib/vxWebComponents\$(CPU)\$(TOOL)\$(LIB\_EXT),  
\$(OMROOT)/lib/vxWebServices\$(CPU)\$(TOOL)\$(LIB\_EXT)*

## HasIDEInterface

The HasIDEInterface property is a Boolean value that specifies whether IDE support is enabled. If IDE support is enabled (Checked), the IDEInterfaceDLL property points to an IDE adapter that provides connection to the IDE. If the property is set to Cleared, IDE support is disabled and IDE services are not attempted. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = Checked*

## IDEInterfaceDLL

The IDEInterfaceDLL property is a string that points to the IDE adapter DLL. You should not have any reason to modify this property. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = \$OMROOT/DLLs/WorkbenchDebuggerIDE.dll*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment. The default values are as follows:

*Default = .cpp*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*Default = include*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\vx6make.bat\" Makefile \$maketarget"*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.



*Default = Cleared*

## **LibExtension**

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .a*

## **LinkDebug**

The LinkDebug property specifies the special link switches used to link in debug mode.

*Default = -X -r*

## **LinkRelease**

The LinkRelease property specifies the special link switches used to link in release mode.

*Default = -X -r*

## **LinkSwitches**

The LinkSwitches property specifies the standard link switches used to link in any mode.

*Default = \$OMLinkCommandSet*

## **MakeExtension**

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the CPP\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*Default = .makefile*

## **MakeFileContent**

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type

- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration.

```
Default = OMROOT=$OMRoot INSTRUMENTATION=$OMInstrumentation LIBS+=$OMLibs
LIB_EXT=.a TARGET_TYPE=$OMTargetType ConfigurationCPPCompileSwitches =
$OMReusableStatechartSwitches $OMConfigurationCPPCompileSwitches ifeq ($(TOOL),diab)
ConfigurationCPPCompileSwitches += $DiabCompileSwitches else ifeq ($(TOOL),gnu)
ConfigurationCPPCompileSwitches += $GNUCompileSwitches endif endif INCLUDE_QUALIFIER=-I
INCLUDE_PATH=$OMIncludePath ifeq ($(INSTRUMENTATION),Animation )
INST_FLAGS=-DOMANIMATOR -DUSE_Iostream INST_INCLUDES=-I$(OMROOT)/LangCpp/aom
-I$(OMROOT)/LangCpp/tom INST_LIBS=
$(OMROOT)/LangCpp/lib/vxaomanim$(CPU)$(TOOL)$(LIB_EXT)
OXF_LIBS=$(OMROOT)/LangCpp/lib/vxoxfinst$(CPU)$(TOOL)$(LIB_EXT)
$(OMROOT)/LangCpp/lib/vxomcomappl$(CPU)$(TOOL)$(LIB_EXT) SOCK_LIB= else ifeq
$(INSTRUMENTATION),Tracing ) INST_FLAGS=-DOMTRACER -DUSE_Iostream
INST_INCLUDES=-I$(OMROOT)/LangCpp/aom -I$(OMROOT)/LangCpp/tom
INST_LIBS=$(OMROOT)/LangCpp/lib/vxtomtrace$(CPU)$(TOOL)$(LIB_EXT)
$(OMROOT)/LangCpp/lib/vxaomtrace$(CPU)$(TOOL)$(LIB_EXT) OXF_LIBS=
$(OMROOT)/LangCpp/lib/vxoxfinst$(CPU)$(TOOL)$(LIB_EXT)
$(OMROOT)/LangCpp/lib/vxomcomappl$(CPU)$(TOOL)$(LIB_EXT) SOCK_LIB= else ifeq
$(INSTRUMENTATION),None ) INST_FLAGS= INST_INCLUDES= INST_LIBS=
OXF_LIBS=$(OMROOT)/LangCpp/lib/vxoxf$(CPU)$(TOOL)$(LIB_EXT) SOCK_LIB= endif endif endif
ADDED_INCLUDES+=$(INCLUDE_PATH) $(INST_INCLUDES) -I$AdaptorSearchPath
-I$OMDefaultSpecificationDirectory -I$(OMROOT) -I$(OMROOT)/LangCpp
-I$(OMROOT)/LangCpp/oxf ADDED_C++FLAGS+=$(INST_FLAGS)
$(ConfigurationCPPCompileSwitches) ifeq ($(TARGET_TYPE),Executable) ADDED_LIBS+=$(LIBS)
$(OXF_LIBS) $(INST_LIBS) $(SOCK_LIB) SUB_OBJECTS+=$(LIBS) $(OXF_LIBS) $(INST_LIBS)
$(SOCK_LIB) endif
```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the CPP\_CG::<Environment>::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## ObjCleanCommand

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default = \$(RM) \$OMFileObjPath*

## ObjectName

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = Empty string*

## ObjectsDirectory

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = Empty string*

## ObjExtension

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*Default = .o*

## OSFileSystemCaseSensitive

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Cleared*

## OSVersion

Used for the various Wind River environments, the OSVersion property represents the version number of the product. The value of the property is used in the batch files that build applications and framework libraries for these environments. In these batch files, the value of the property is provided as a parameter for both the "wrenv" command and the "make" command.

*Default = 6.8*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = [""]([^\:]+)[""][,][ ]line ([0-9]+)[\:]*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation.

*Default = /*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Checked*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running Rational Rhapsody, whereas the target is the machine running the application. To run remotely, the UseRemoteHost property must be set to True. If UseRemoteHost is True and RemoteHost is blank, the current host name is used for the remote host. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = Empty string*

## ReusableStatechartSwitches

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts. See the upgrade history on the support site for information on this change.

*Default = -DOM\_REUSABLE\_STATECHART\_IMPLEMENTATION*

## **SpecExtension**

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*Default = .h*

## **UnixPathNameForOMROOT**

The UnixPathNameForOMROOT property specifies whether the makefile must include UNIXstyle path names.

The pRISM compilers do not tolerate DOS-style path conventions. If you do not set this property correctly, there might be many compilation problems.

*Default = Checked*

## **UpdateBuildSettingsInIDE**

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## **UseNonZeroStdInputHandle**

The UseNonZeroStdInputHandle property is a Boolean value that specifies whether to use a non-zero standard input handle.

*Default = Checked*

## **UseRemoteHost**

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Checked*

## **UseTemplateTypename**

In the declarations for members of a template class, some compilers require that you use precede the

member type with the keyword "typename" for types with dependent names. Since support for this keyword varies between compilers, the UseTemplateTypename property is used to specify whether the "typename" keyword should be included in the generated code.

*Default = Cleared*

## WorkbenchManaged653

The WorkbenchManaged653 metaclass contains environment settings (compiler, framework libraries, and so on) for ARINC 653 development.

### AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)/LangCpp/osconfig/Vx653*

### AdditionalReservedWords

The AdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = Blank*

## API

The API property is used to specify the ARINC 653 API that should be used - POSIX or APEX. Currently, for OXF-based projects, Rational Rhapsody supports only the POSIX API. For SXF-based projects, Rational Rhapsody supports only the APEX API.

*Default = posix for OXF-based projects, apex for SXF-based projects*

## **AutoAttachToIDEDebugger**

The AutoAttachToIDEDebugger property is used to specify that you would like the Workbench debugger to be automatically synchronized with the Rational Rhapsody animation. If for some reason you do not want automatic synchronization, for example, if you prefer to manually connect to the relevant target server, then you can set the value of this property to False.

*Default = True*

## **BSP**

The BSP property specifies the board support package (BSP) for the system. If you need to change the value of the CPU, you can simply reset the value in this property instead of changing the value in the MakeFileContent property.

*Default = SIMNT*

## **BuildArgumentsInIDE**

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the [lang]\_CG::[environment]::BuildCommandInIDE property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## **BuildCommandInIDE**

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## **BuildCommandSet**

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration. To change this property, use the Configuration window in the browser - do not change it by using the Properties window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link.

*Default = Debug*

## **buildFrameworkCommand**

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\"\$OMROOT/etc/Executer.exe\" \"\\\"\\\"\$OMROOT\\\"\\\"etc\\vx6make.bat vx653build.mak*

## **BuildInIDE**

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = True*

## **CodeTestSettings**

The CodeTestSettings property specifies the compiler command settings. This property supports integration with Applied Microsystems Corporation CodeTest.

*Default = Blank*

## **CompileSwitches**

The CompileSwitches property specifies the compiler switches.

*Default = Blank*

## **CPPCompileCommand**



The `CPPCompileCommand` property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = @echo Compiling \$OMFileImpPath \$(CREATE\_OBJ\_DIR) @\$(CXX) \$(C++FLAGS)  
\$OMFileCPPCompileSwitches -o \$OMFileObjPath \$OMFileImpPath*

## CPPCompileDebug

The `CPPCompileDebug` property modifies the makefile compile command with switches for building a debug version of the component.

*Default = -g*

## CPPCompileRelease

The `CPPCompileRelease` property specifies additional compilation flags for a configuration set to Release mode.

*Default = Blank*

## DependencyRule

The `DependencyRule` property specifies how file dependencies for a configuration are generated in the makefile.

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: `$OMFileObjPath : $OMFileImpPath "*.bmp" "*.ico" "*.rc2"`

*Default = \$OMFileObjPath : \$OMFileImpPath \$OMFileSpecPath \$OMFileDependencies*

## DuplicateLibsListInMakeFile

The `DuplicateLibsListInMakeFile` property is a Boolean value that specifies whether Rational Rhapsody should duplicate the libraries list in the generated makefile link command. This property supports linkers that are sensitive to library order in the link command.

*Default = True*

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction an IDE such as Eclipse, you can use the `EnableDebugIntegrationWithIDE` property to specify whether or not the IDE debugger is to be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to Checked, the IDE debugger is used.

*Default = False*

## EndApplicationCode

When the main function ends, the system's `exit()` function is called automatically. The system function calls the destructors of framework instances in the order it determines. The `EndApplicationCode` property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an `endApplication()` function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to `endApplication()`, it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The `EntryPoint` property specifies the name of the main program for a given environment.

*Default = void*

## EntryPointDeclarationModifier

The `EntryPointDeclarationModifier` property specifies a modifier for the entry point declaration. This property allows generation of the `main()` function in the specified syntax.

To modify the `main()` signature implemented in the OSE adapter, do the following:

- Add the `EntryPointDeclarationModifier` property to your environment properties and set it to the main return value and name. For example: "int main"
- Set the `EntryPoint` property to the main arguments. For example: "int a, long b, char\*\*"
- Generate the code.

You get the following `main()` declaration:

```
int main(int a, long b, char** c) { ... }
```

*Default = extern \"C\" void vxmain*

## ErrorMessageTokensFormat

The `ErrorMessageTokensFormat`, working with the `ParseErrorMessage` property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. The `ErrorMessageTokensFormat` property defines the number

and location of tokens within the regular expression defined by the ParseErrorMessage property. The ErrorMessageTokens property has three parameters, each with an integer value:

- TotalNumberOfTokens - The number of tokens in the regular expression
- FileTokenPosition - The position of the file name token in the expression
- LineTokenPosition - The position of the line number token in the expression

*Default = TotalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2*

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the C\_CG::<Environment>::ExeName property plus the value of the ExeExtension property.

*Default = .sm*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the [lang]\_CG::[environment]::ExeExtension property.

*Default = Blank*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements. The file inclusions are generated in the makefile.

*Default = \$OMSpecIncludeInElements \$OMImpIncludeInElements*

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile.

Note: If you select Release Build Set, these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

*Default = Blank*

## GNUCompileSwitches

The GNUCompileSwitches property is used to specify switches that are relevant only for the GNU compiler. The ability to specify GNU-only switches is needed for versions of VxWorks that allow you to use either the GNU or diab compiler. You can see the value of this property included in the value of the MakeFileContent property.

*Default = -DVxWorks \$(CPPCompileDebug) -D\"USER\_APPL\_INIT=vxmain()\"*

## HasIDEInterface

The HasIDEInterface property is a Boolean value that specifies whether IDE support is enabled.

If the value of this property is set to True, the IDEInterfaceDLL property points to an IDE adapter that provides connection to the IDE. If the property is set to False, IDE support is disabled and IDE services are not attempted. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = True*

## IDEInterfaceDLL

The IDEInterfaceDLL property is a string that points to the IDE adapter DLL. You should not have any reason to modify this property. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = \$OMROOT/DLLs/WorkbenchDebuggerIDE.dll*

## IgnoreSwitches

The IgnoreSwitches property is used to suppress certain warning messages. It is used primarily for suppressing warning messages connected to the diab compiler.

*Default = Blank*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment.

*Default = .cpp*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*Default = include*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, include the name of the property preceded by \$. As shown in the following example (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

*Default = Blank*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = False*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .a*

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

*Default = -X -r*

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

*Default = -X -r*

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

*Default = \$OMLinkCommandSet*

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the [lang]\_CG::[environment]::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*Default = .makefile*

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration. For example, the default content of the target type section of a C++ makefile for the Microsoft environment is as follows: ##### Target type (Debug/Release)

```
#####  
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease  
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease BuildSet=$OMBuildSet  
SUBSYSTEM=$OMSubSystem COM=$OMCOM RPFrameWorkDll=$OMRPFrameWorkDll
```

```
ConfigurationCPPCompileSwitches= $OMReusableStatechartSwitches
$OMConfigurationCPPCompileSwitches !IF "$ (RPFrameWorkDll)" == "True"
ConfigurationCPPCompileSwitches= $(ConfigurationCPPCompileSwitches) /D "FRAMEWORK_DLL"
!ENDIF !IF "$ (COM)" == "True" SUBSYSTEM=/SUBSYSTEM:windows !ENDIF
```

**Compilation Flags** The compilation flags section of the makefile contains the default compilation flags stored in the CompileSwitches property. For example, the default content of the compilation flags section of a C++ makefile for the Microsoft environment is as follows: ##### Compilation flags #####  
INCLUDE\_QUALIFIER=/I LIB\_PREFIX=MS

**Commands Definitions** The commands definition section of the makefile specifies programs to execute from the makefile. For example, the default commands definition section of a C++ makefile for the Microsoft environment is as follows: ##### Commands definition #####  
##### RMDIR = rmdir LIB\_CMD=link.exe -lib  
LINK\_CMD=link.exe LIB\_FLAGS=\$OMConfigurationLinkSwitches  
LINK\_FLAGS=\$OMConfigurationLinkSwitches \$(SUBSYSTEM) / MACHINE:I386

**Generated Macros** The generated macros section of the makefile contains a variable that expands to the Rational Rhapsody-generated macros in the makefile. For example: ##### Generated macros #####  
##### \$OMContextMacros  
OBJ\_DIR=\$OMObjectsDir !IF "\$ (OBJ\_DIR)" != "" CREATE\_OBJ\_DIR=if not exist \$(OBJ\_DIR) mkdir  
\$(OBJ\_DIR) CLEAN\_OBJ\_DIR= if exist \$(OBJ\_DIR) \$(RMDIR) \$(OBJ\_DIR) !ELSE  
CREATE\_OBJ\_DIR= CLEAN\_OBJ\_DIR= !ENDIF

The \$OMContextMacros keyword expands several macros in the makefile. Each makefile macro has its own keyword. You can use these keywords separately to customize the makefile. The \$OMContextMacros variable modifies target-specific variables. Replace the \$OMContextMacros line in the MakeFileContent property with the following: FLAGSFILE=\$OMFlagsFile  
RULESFILE=\$OMRulesFile OMROOT=\$OMROOT C\_EXT=\$OMImplExt H\_EXT=\$OMSpecExt  
OBJ\_EXT=\$OMObjExt EXE\_EXT=\$OMExeExt LIB\_EXT=\$OMLibExt  
INSTRUMENTATION=\$OMInstrumentation TIME\_MODEL=\$OMTimeModel  
TARGET\_TYPE=\$OMTargetType TARGET\_NAME=\$OMTargetName \$OMAllDependencyRule  
TARGET\_MAIN=\$OMTargetMain LIBS=\$OMLibs INCLUDE\_PATH=\$OMIncludePath  
ADDITIONAL\_OBJS=\$OMAdditionalObjs OBJS= \$OMObjs

**Predefined Macros** The predefined macros section of the makefile contains other macros than the Rational Rhapsody-generated macros specified in the generated macros section. For example, the default predefined macros section of a C++ makefile for the Microsoft environment is as follows:

```
##### Predefined macros #####
##### $(OBJS) : $(INST_LIBS)
$(OXF_LIBS) LIB_POSTFIX= !IF "$ (BuildSet)" == "Release" LIB_POSTFIX=R !ENDIF !IF
"$(TARGET_TYPE)" == "Executable" LinkDebug=$(LinkDebug) /DEBUG
LinkRelease=$(LinkRelease) /OPT:NOREF !ELSEIF "$(TARGET_TYPE)" == "Library"
LinkDebug=$(LinkDebug) /DEBUGTYPE:CV !ENDIF !IF "$(INSTRUMENTATION)" == "Animation"
INST_FLAGS=/D "OMANIMATOR" INST_INCLUDES=/I $(OMROOT)\LangCpp\aoom /I
$(OMROOT)\LangCpp\tom !IF "$ (RPFrameWorkDll)" == "True" INST_LIBS=
OXF_LIBS=$(OMROOT)\LangCpp\lib$(LIB_PREFIX)oxfanimdll$(LIB_POSTFIX) $(LIB_EXT)
!ELSE INST_LIBS= $(OMROOT)\LangCpp\lib$(LIB_PREFIX)aomanim$(LIB_POSTFIX) $(LIB_EXT)
OXF_LIBS=$(OMROOT)\LangCpp\lib$(LIB_PREFIX)oxfinst$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib$(LIB_PREFIX)omComAppl$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)" == "Tracing" INST_FLAGS=/D
"OMTRACER" INST_INCLUDES=/I $(OMROOT)\LangCpp\aoom /I $(OMROOT)\LangCpp\tom !IF
"$(RPFrameWorkDll)" == "True" INST_LIBS= OXF_LIBS=
```



```
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfracedll$(LIB_POSTFIX)$(LIB_EXT) !ELSE
INST_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)tomtrace$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomtrace$(LIB_POSTFIX)$(LIB_EXT) OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfinst$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX)$(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)" == "None" INST_FLAGS=
INST_INCLUDES= INST_LIBS= !IF "$(RPFramewordDll)" == "True"
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfdll$(LIB_POSTFIX)$(LIB_EXT) !ELSE
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$(LIB_POSTFIX)$(LIB_EXT) !ENDIF
SOCK_LIB= !ELSE !ERROR An invalid Instrumentation $(INSTRUMENTATION) is specified.
!ENDIF
```

**Generated Dependencies** The generated dependencies section of the makefile contains a variable that expands to Rational Rhapsody-generated dependencies and compilation instructions. For example, the generated dependencies section of a C++ makefile for the Microsoft environment is as follows:

```
##### Generated dependencies #####
##### $OMContextDependencies
$OMFileObjPath : $OMMainImplementationFile $(OBJS) $(CPP) $(ConfigurationCPPCompileSwitches)
/Fo"$OMFileObjPath" $OMMainImplementationFile
```

**Linking Instructions** The linking instructions section of the makefile contains the predefined linking instructions. For example, the default linking instructions section of a C++ makefile for the Microsoft environment is as follows:

```
##### Linking instructions #####
#####
$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS) $OMFileObjPath
$OMMakefileName $OMModelLibs @echo Linking $(TARGET_NAME)$(EXE_EXT) $(LINK_CMD)
$OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \ $(OXF_LIBS) \
$(SOCK_LIB) \ $(LINK_FLAGS) /out:$(TARGET_NAME)$(EXE_EXT)
$(TARGET_NAME)$(LIB_EXT) : $(OBJS) $(ADDITIONAL_OBJS) $OMMakefileName @echo
Building library $@ $(LIB_CMD) $(LIB_FLAGS) /out:$(TARGET_NAME)$(LIB_EXT) $(OBJS)
$(ADDITIONAL_OBJS) clean: @echo Cleanup $OMCleanOBJS if exist $OMFileObjPath erase
$OMFileObjPath if exist *$(OBJ_EXT) erase *$(OBJ_EXT) if exist $(TARGET_NAME).pdb erase
$(TARGET_NAME).pdb if exist $(TARGET_NAME)$(LIB_EXT) erase
$(TARGET_NAME)$(LIB_EXT) if exist $(TARGET_NAME).ilk erase $(TARGET_NAME).ilk if exist
$(TARGET_NAME)$(EXE_EXT) erase $(TARGET_NAME)$(EXE_EXT) $(CLEAN_OBJ_DIR)
```

```
Default = OMROOT=$OMRoot INSTRUMENTATION=$OMInstrumentation ADD_LIBS=$OMLibs
ADD_VTHREADS_LIBS=-L$(WIND_GNU_PATH)/target/i586-wrs-vxworksae/lib/mvthreads -lgcc
-lstdc++ -L$(WIND_BASE)/target/vThreads/lib -l$(CPU)gnumv LIB_EXT=.a
TARGET_TYPE=$OMTargetType ConfigurationCPPCompileSwitches =
$OMReusableStatechartSwitches $OMConfigurationCPPCompileSwitches
ConfigurationCPPCompileSwitches += $GNUCompileSwitches INCLUDE_QUALIFIER=-I
INCLUDE_PATH=$OMIncludePath $(INCLUDE_QUALIFIER)$OMConfigurationPath ifeq
$(INSTRUMENTATION),Animation ) INST_FLAGS=-DOMANIMATOR -DUSE_Iostream
INST_INCLUDES=-I$(OMROOT)/LangCpp/aom -I$(OMROOT)/LangCpp/tom INST_LIBS=
$(OMROOT)/LangCpp/lib/vx653aomanim$(CPU)$(API)$(LIB_EXT)
OXF_LIBS=$(OMROOT)/LangCpp/lib/vx653oxfinst$(CPU)$(API)$(LIB_EXT)
$(OMROOT)/LangCpp/lib/vx653omcomappl$(CPU)$(API)$(LIB_EXT) SOCK_LIB= else ifeq
$(INSTRUMENTATION),Tracing ) INST_FLAGS=-DOMTRACER -DUSE_Iostream
INST_INCLUDES=-I$(OMROOT)/LangCpp/aom -I$(OMROOT)/LangCpp/tom
INST_LIBS=$(OMROOT)/LangCpp/lib/vx653tomtrace$(CPU)$(API)$(LIB_EXT)
$(OMROOT)/LangCpp/lib/vx653aomtrace$(CPU)$(API)$(LIB_EXT) OXF_LIBS=
$(OMROOT)/LangCpp/lib/vx653oxfinst$(CPU)$(API)$(LIB_EXT)
$(OMROOT)/LangCpp/lib/vx653omcomappl$(CPU)$(API)$(LIB_EXT) SOCK_LIB= else ifeq
```



```

( $\$(INSTRUMENTATION),None$  )  $INST\_FLAGS=$   $INST\_INCLUDES=$   $INST\_LIBS=$ 
 $OXF\_LIBS=\$(OMROOT)/LangCpp/lib/vx653oxf\$(CPU)\$(API)\$(LIB\_EXT)$   $SOCK\_LIB=$   $endif$   $endif$   $endif$ 
 $ADDED\_INCLUDES+=\$(INCLUDE\_PATH) \$(INST\_INCLUDES) -I\$(AdaptorSearchPath$ 
 $-I\$(OMDefaultSpecificationDirectory) -I\$(OMROOT) -I\$(OMROOT)/LangCpp$ 
 $-I\$(OMROOT)/LangCpp/oxf$   $ADDED\_C++FLAGS+=\$(INST\_FLAGS)$ 
 $\$(ConfigurationCPPCompileSwitches)$   $ifeq$  ( $\$(TARGET\_TYPE),Executable$ )  $ADDED\_LIBS+=\$(LIBS)$ 
 $\$(ADD\_LIBS) \$(INST\_LIBS) \$(OXF\_LIBS) \$(INST\_LIBS) \$(SOCK\_LIB) \$(ADD\_VTHREADS\_LIBS)$ 
 $SUB\_OBJECTS+=\$(LIBS) \$(ADD\_LIBS) \$(INST\_LIBS) \$(OXF\_LIBS) \$(INST\_LIBS) \$(SOCK\_LIB)$ 
 $\$(ADD\_VTHREADS\_LIBS)$   $endif$ 

```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the [lang]\_CG::[environment]::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*Default = Blank*

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## ObjCleanCommand

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default =  $\$(RM)$   $\$(OMFileObjPath)$*

## ObjectName

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = Blank*

## ObjectsDirectory

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the

generated makefile.

*Default = Blank*

## ObjExtension

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*Default = .o*

## OSFileSystemCaseSensitive

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = False*

## OSVersion

Used for the various Wind River environments, the OSVersion property represents the version number of the product. The value of the property is used in the batch files that build applications and framework libraries for these environments. In these batch files, the value of the property is provided as a parameter for both the "wrenv" command and the "make" command.

*Default = 2.3*

## ParseErrorDescript

The ParseErrorDescript property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = ([^:]+)[:]/([0-9]+)[:] (error/warning)[:] (.\*)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning)

Along with the ErrorMessageTokensFormat property, the ParseErrorMessage property specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = ([^:]+)[:]/([0-9]+)[:]*

## ParseMakeError

The ParseMakeError property is used to define a regular expression that represents the format of make process or linker error messages. This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = (.\*)(make)(.)(Error)*

## ParseSeverityError

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^:]+):([0-9]+): (error)*

## ParseSeverityWarning

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^:]+):([0-9]+): (warning)*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation.

*Default = /*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = True*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running the product, whereas the target is the machine running the application. To run remotely, the UseRemoteHost property must be set to Checked.

If the UseRemoteHost property is set to Checked and the RemoteHost property is blank, the current host name is used for the remote host. You can use this as a workaround if you have problems running

animated applications on Windows 95.

You can leave the RemoteHost property blank if both the application and Rational Rhapsody are running on the same machine.

*Default = Blank*

## ReusableStatechartSwitches

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts.

*Default = -DOM\_REUSABLE\_STATECHART\_IMPLEMENTATION*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*Default = .h*

## TargetArchitecture

The property TargetArchitecture can be used to specify the architecture of the target environment for which you are developing your application.

*Default = i586-wrs-vxworksae*

## Tool

The Tool property is used to specify the compiler to use. For certain versions of VxWorks, you can use the GNU or diab compiler. The value of this property is passed to the command that makes the makefile, and to the command that builds the Rational Rhapsody framework.

*Default = gnu*

## UnixPathNameForOMROOT

The UnixPathNameForOMROOT property specifies whether the makefile must include UNIXstyle path names.

The pRISM compilers do not tolerate DOS-style path conventions. If you do not set this property correctly, there may be many compilation problems.

*Default = True*

## UpdateBuildSettingsInIDE

The `UpdateBuildSettingsInIDE` property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to `True`, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = True*

## UseNewBuildOutputWindow

The `UseNewBuildOutputWindow` property determines which tab is brought to the front of the Output window after the completion of a build action. If set to `True`, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the `CG::General::ShowLogViewAfterBuild` property to `True`.

*Default = True*

## UseNonZeroStdInputHandle

The `UseNonZeroStdInputHandle` property is a Boolean value that specifies whether to use a non-zero standard input handle.

*Default = True*

## UseRemoteHost

The `UseRemoteHost` property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = True*

## UseTemplateName

In the declarations for members of a template class, some compilers require that you use precede the member type with the keyword `"typename"` for types with dependent names. Since support for this keyword varies between compilers, the `UseTemplateName` property is used to specify whether the `"typename"` keyword should be included in the generated code.

*Default = False*

## WorkbenchManaged\_RTP

The WorkbenchManaged\_RTP metaclass contains environment settings (Compiler, framework libraries, and so on) for WorkbenchManaged\_RTP compiler.

### AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)/LangCpp/osconfig/VxWorks*

### AdditionalReservedWords

The AdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = Empty string*

### AutoAttachToIDEDebugger

The AutoAttachToIDEDebugger property is used to specify that you would like the Workbench debugger to be automatically synchronized with the Rational Rhapsody animation. If for some reason you do not want automatic synchronization, for example, if you prefer to manually connect to the relevant target server, then you can set the value of this property to False.

*Default = Checked*

## BSP

The BSP property specifies the board support package (BSP) for the system. If you need to change the value of the CPU, you can simply reset the value in this property instead of changing the value in the MakeFileContent property.

*Default = PENTIUM*

## BuildArgumentsInIDE

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the CPP\_CG::[environment]::BuildCommandInIDE property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandInIDE

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration.

To change this property, use the Configuration window in the browser - do not change it by using the Properties tab in the Features window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link.

The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag

(:cx\_option=exceptions).

- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*Default = Debug*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\$OMROOT/etc/Executer.exe" "\""\$OMROOT\etc\vx6make.bat" vxbuild.mak buildLibs 6.5  
\"CPU=\$BSP\" \"TOOL=\$Tool\" \"TOOL\_FAMILY=\$Tool\" \"BUILD=\$BuildCommandSet\"  
\"USE\_RTP=TRUE\" \" \"*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Checked*

## CodeTestSettings

The CodeTestSettings property specifies the compiler command settings. This property supports integration with Applied Microsystems Corporation CodeTest.

*Default = CXX = \$(AMC\_HOME)\bin\ctcxx*

## CompileSwitches

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property.

*Default = Empty MultiLine*

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general,



this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The default is as follows:

```
@echo Compiling $OMFileImpPath $(CREATE_OBJ_DIR) @$(CXX) $(C++FLAGS)
$OMFileCPPCompileSwitches -o $OMFileObjPath $OMFileImpPath
```

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*Default = -g*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*Default = Empty string*

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

The default is as follows:

```
$OMFileObjPath : $OMFileImpPath $OMFileSpecPath $OMFileDependencies
```

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: `$OMFileObjPath : $OMFileImpPath "*.bmp" "*.ico" "*.rc2"`

## DuplicateLibsListInMakeFile

The DuplicateLibsListInMakeFile property is a Boolean value that specifies whether Rational Rhapsody should duplicate the libraries list in the generated makefile link command. This property supports linkers that are sensitive to library order in the link command.

*Default = Checked*

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction with an IDE such as Eclipse, the

EnableDebugIntegrationWithIDE property can be used to specify whether or not the IDE debugger should be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to True, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's exit() function is called automatically. The system function calls the destructors of framework instances in the order it determines. The EndApplicationCode property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an endApplication() function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to endApplication(), it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The EntryPoint property specifies the name of the main program for a given environment.

*Default = main*

If applicable for the metaclass, see also the definition of the EntryPointDeclarationModifier property for more information.

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens has three parameters, each with an integer value:

- TotalNumberOfTokens - The number of tokens in the regular expression
- FileTokenPosition - The position of the file name token in the expression
- LineTokenPosition - The position of the line number token in the expression

*Default = ToTalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2*

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable

program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the `CPP_CG::<Environment>::ExeName` property plus the value of this property.

*(Default = .vxe)*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the `ExeName` property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the `CPP_CG::<Environment>::ExeExtension` property.

*(Default = Blank)*

## FileDependencies

The `FileDependencies` property specifies which framework specification files and implementation files should be included in model elements. The file inclusions are generated in the makefile.

*Default = \$OMSpecIncludeInElements \$OMImpIncludeInElements*

## GetConnectedRuntimeLibraries

The `GetConnectedRuntimeLibraries` property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile. Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

*Default =*

`$(OMROOT)/LangCpp/lib/vxWebComponents$(CPU)$(RTP_SUFFIX)$(TOOL)$(LIB_EXT),  
$(OMROOT)/lib/vxWebServices$(CPU)$(RTP_SUFFIX)$(TOOL)$(LIB_EXT)`

## HasIDEInterface

The `HasIDEInterface` property is a Boolean value that specifies whether IDE support is enabled. If IDE support is enabled (Checked), the `IDEInterfaceDLL` property points to an IDE adapter that provides connection to the IDE. If the property is set to Cleared, IDE support is disabled and IDE services are not

attempted. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = Checked*

## IDEInterfaceDLL

The IDEInterfaceDLL property is a string that points to the IDE adapter DLL. You should not have any reason to modify this property. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = \$OMROOT/DLLs/WorkbenchDebuggerIDE.dll*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment.

*Default = .cpp*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*Default = include*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\vx6make.bat\" Makefile \$maketarget"*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is

Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .a*

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

*Default = -X -r*

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

*Default = -X -r*

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

*Default = \$OMLinkCommandSet*

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the CPP\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*(Default = .makefile)*

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration.

```
Default = OMROOT=$OMRoot INSTRUMENTATION=$OMInstrumentation LIBS+= $OMLibs
RTP_SUFFIX = _RTP_ LIB_EXT=.a TARGET_TYPE=$OMTargetType
ConfigurationCPPCompileSwitches = $OMReusableStatechartSwitches
$OMConfigurationCPPCompileSwitches ifeq ($(TOOL),diab) ConfigurationCPPCompileSwitches +=
$DiabCompileSwitches else ifeq ($(TOOL),gnu) ConfigurationCPPCompileSwitches +=
$GNUCompileSwitches endif endif INCLUDE_QUALIFIER=-I INCLUDE_PATH=$OMIncludePath ifeq
$(INSTRUMENTATION),Animation ) INST_FLAGS=-DOMANIMATOR -DUSE_Iostream
INST_INCLUDES=-I$(OMROOT)/LangCpp/aom -I$(OMROOT)/LangCpp/tom INST_LIBS=
$(OMROOT)/LangCpp/lib/vxaomanim$(CPU)$(RTP_SUFFIX)$(TOOL)$(LIB_EXT)
OXF_LIBS=$(OMROOT)/LangCpp/lib/vxoxfinst$(CPU)$(RTP_SUFFIX)$(TOOL)$(LIB_EXT)
$(OMROOT)/LangCpp/lib/vxomcomappl$(CPU)$(RTP_SUFFIX)$(TOOL)$(LIB_EXT) SOCK_LIB= else
ifeq $(INSTRUMENTATION),Tracing ) INST_FLAGS=-DOMTRACER -DUSE_Iostream
INST_INCLUDES=-I$(OMROOT)/LangCpp/aom -I$(OMROOT)/LangCpp/tom
INST_LIBS=$(OMROOT)/LangCpp/lib/vxtomtrace$(CPU)$(RTP_SUFFIX)$(TOOL)$(LIB_EXT)
$(OMROOT)/LangCpp/lib/vxaomtrace$(CPU)$(RTP_SUFFIX)$(TOOL)$(LIB_EXT) OXF_LIBS=
$(OMROOT)/LangCpp/lib/vxoxfinst$(CPU)$(RTP_SUFFIX)$(TOOL)$(LIB_EXT)
$(OMROOT)/LangCpp/lib/vxomcomappl$(CPU)$(RTP_SUFFIX)$(TOOL)$(LIB_EXT) SOCK_LIB= else
ifeq $(INSTRUMENTATION),None ) INST_FLAGS= INST_INCLUDES= INST_LIBS=
OXF_LIBS=$(OMROOT)/LangCpp/lib/vxoxf$(CPU)$(RTP_SUFFIX)$(TOOL)$(LIB_EXT) SOCK_LIB=
endif endif endif ADDED_INCLUDES+= $(INCLUDE_PATH) $(INST_INCLUDES)
-I$AdaptorSearchPath -I$OMDefaultSpecificationDirectory -I$(OMROOT) -I$(OMROOT)/LangCpp
-I$(OMROOT)/LangCpp/oxf ADDED_C++FLAGS+= $(INST_FLAGS)
$(ConfigurationCPPCompileSwitches) ifeq $(TARGET_TYPE),Executable ADDED_LIBS+= $(LIBS)
$(OXF_LIBS) $(INST_LIBS) $(SOCK_LIB) SUB_OBJECTS+= $(LIBS) $(OXF_LIBS) $(INST_LIBS)
$(SOCK_LIB) endif
```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use

for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the `CPP_CG::<Environment>::MakeExtension` property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## **NullValue**

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## **ObjCleanCommand**

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default = \$(RM) \$OMFileObjPath*

## **ObjectName**

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = Empty string*

## **ObjectsDirectory**

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = Empty string*

## **ObjExtension**

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*Default = .o*

## **OSFileSystemCaseSensitive**

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Cleared*

## OSVersion

Used for the various Wind River environments, the OSVersion property represents the version number of the product. The value of the property is used in the batch files that build applications and framework libraries for these environments. In these batch files, the value of the property is provided as a parameter for both the "wrenv" command and the "make" command.

*Default = 6.8*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = [""]([^\:]+)[""][,][ ]line ([0-9]+)[:]*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation.

*Default = /*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Checked*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running Rational Rhapsody, whereas the target is the machine running the application. To run remotely, the UseRemoteHost property must be set to True. If UseRemoteHost is True and RemoteHost is blank, the current host name is used for the remote host. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.



*Default = Empty string*

## **ReusableStatechartSwitches**

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts. See the upgrade history on the support site for information on this change.

*Default = -DOM\_REUSABLE\_STATECHART\_IMPLEMENTATION*

## **SpecExtension**

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*Default = .h*

## **UnixPathNameForOMROOT**

The UnixPathNameForOMROOT property specifies whether the makefile must include UNIXstyle path names.

The pRISM compilers do not tolerate DOS-style path conventions. If you do not set this property correctly, there might be many compilation problems.

*Default = Checked*

## **UpdateBuildSettingsInIDE**

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## **UseNonZeroStdInputHandle**

The UseNonZeroStdInputHandle property is a Boolean value that specifies whether to use a non-zero standard input handle.

*Default = Checked*

## **UseRemoteHost**

The UseRemoteHost property specifies whether you intend to run an application remotely over a network

by default in a given environment.

*Default = Checked*

## **UseTemplateName**

In the declarations for members of a template class, some compilers require that you use precede the member type with the keyword "typename" for types with dependent names. Since support for this keyword varies between compilers, the UseTemplateName property is used to specify whether the "typename" keyword should be included in the generated code.

*Default = Cleared*

# CPP\_ReverseEngineering

---

In addition to the ReverseEngineering subject, Rational Rhapsody provides language-specific subjects to control how Rational Rhapsody imports legacy code. Most of the properties are identical for each language. In general, most of the reverse engineering properties have graphical representation in the Reverse Engineering Advanced Options window. You should change the options by using the Reverse Engineering Advanced Options window instead of the corresponding properties.

## Filtering

The Filtering metaclass contains properties that control which items are analyzed during the reverse engineering operation.

### AnalyzeGlobalFunctions

The AnalyzeGlobalFunctions property specifies whether to analyze global functions.

*Default = Checked*

### AnalyzeGlobalTypes

The AnalyzeGlobalTypes property specifies whether to analyze global types.

*Default = Checked*

### AnalyzeGlobalVariables

The AnalyzeGlobalVariables property specifies whether to analyze global variables.

*Default = Checked*

### CreateReferenceClasses

The CreateReferenceClasses property specifies whether to create external classes for undefined classes that result from forward declarations and inheritance.

By default, reference classes are created. If the incomplete class cannot be resolved, the tool deletes the incomplete class if this property is set to Cleared. In some cases, the class cannot be deleted (for example, a class referenced by a typedef type).

*Default = Checked*

## IncludeInheritanceInReference

The IncludeInheritanceInReference property specifies whether to include inheritance information in reference classes.

*Default = Cleared*

## ReferenceClasses

The ReferenceClasses property specifies which classes to model as reference classes. Reference classes are classes that can be mentioned in the final design as placeholders without having to specify their internal details. For example, you can include the MFC classes as reference classes, without having to specify any of their members or relations.

They would simply be modeled as terminals for context, to show that they are acting as superclasses or peers to other classes.

*Default = empty string*

## ReferenceDirectories

The ReferenceDirectories property specifies which directories (and subdirectories) contain reference classes.

*Default = empty string*

## ImplementationTrait

The ImplementationTrait metaclass contains properties that determine the implementation traits used during the reverse engineering operation.

## AnalyzeIncludeFiles

The AnalyzeIncludeFiles property specifies which, if any, include files should be analyzed during reverse engineering. The possible values are as follows:

- AllIncludes - Analyze all include files.
- IgnoreIncludes - Ignore all include files.
- OnlyFromSelected - Analyze the specified include files only.
- OnlyLogicalHeader - Analyze the logical header files only.

*Default = OnlyFromSelected*

## AutomaticIncludePath

When Rational Rhapsody reverse engineers a file, there might be cases where the file references a header file but the path in the include directive is not clear enough for Rational Rhapsody to find the file. If you set the value of the AutomaticIncludePath property to Checked, then in such cases, Rational Rhapsody searches the list of files to be reverse engineered to see if the list contains a header file with that name. If there is such a file, Rational Rhapsody uses the full path that was provided for that header file, assuming that this is the header file that was being referenced in the original file.

Rational Rhapsody performs this search for ambiguous header files when it does macro collection. This means that if the value of the CPP\_ReverseEngineering::ImplementationTrait::CollectMode property is set to None, then Rational Rhapsody does not search for ambiguous header files even if the value of the AutomaticIncludePath property is set to Checked.

*Default = Checked*

## CreateBlackDiamondAssociations

The CreateBlackDiamondAssociations property specifies how the reverse engineering feature should handle composition relationships. If the value of the property is set to False, then Rational Rhapsody creates parts. If the value of the property is set to Checked, Rational Rhapsody creates composition associations (black diamond).

*Default = Cleared*

## CollectMode

The CollectMode property allows Rational Rhapsody to collect macros. The possible values are as follows:

- None - Macros are not collected from include files that are not on the reverse engineering list.
- Once - Macros are collected only if the model does not yet include a controlled file of collected macros.
- Always - Macros are collected each time reverse engineering is carried out. The controlled file that stores the macros are replaced each time.

*Default = Once*

## ComponentFileType

The possible values are "SpecificationOrImplementation" or "Logical."

*Default = SpecificationOrImplementation*

## CreateDependencies

The CreateDependencies property allows you to specify how the Reverse Engineering feature should handle the creation of dependency elements in the model from code constructs such as #includes, forward declarations, friends, and namespace usage.

The default value for this property represents the most advanced handling option available in Rational Rhapsody, while some of the other options represent various handling options that were introduced in earlier versions of Rational Rhapsody.

The possible values for this property are:

- None - Dependencies are not created to represent code constructs such as #includes and forward declarations.
- DependenciesOnly - Dependencies are created in the model only when the code represents a dependency relationship between two classes.
- PackageOnly - Actual dependencies are created in the model only when the code represents a dependency relationship between two classes. For other #include statements, the relevant information is stored in the SpecIncludes and ImpIncludes properties so that the original code can be regenerated.
- ComponentOnly - Dependencies are created for all code constructs that represent dependency relationships, however the dependency is created between the component files that contain the elements rather than between the elements themselves.
- PackageAndComponent - Dependencies are created for all code constructs that represent dependency relationships. For each such relationship, a dependency is created between the component files containing the relevant elements, and where possible a dependency is also created between the relevant elements themselves. This means that for some #includes two dependencies are created in the model.
- SmartPackageAndComponent - Dependencies are created for all code constructs that represent dependency relationships. Where possible, the dependency is created between the relevant elements. Where this is not possible, a dependency is created between the component files containing the elements. This means that only a single dependency is created for any single #include statement.

*Default = SmartPackageAndComponent*

## CreateFilesIn

The CreateFilesIn property is a placeholder for the reverse engineering option Create File-s In option. You should not set this value directly. The C++ default value is None.

## DataTypesLibrary

The Mapping tab of the Reverse Engineering Options window allows you to specify a list of types that should be modeled as "Language" types. You can add individual types to the list or groups of types that you have previously defined as data types for a specific library.

If you select the option of adding a library, you are presented with a drop-down list of libraries to choose from. The libraries on this list are taken from the value of the DataTypesLibrary property. You can add a number of libraries to the drop-down list by using a comma-separated list of names as the value for this property.

When you select a library from the drop-down list, all of the types that were defined for that library are added to the list of types.

You define types for a library by carrying out the following steps:

- In the relevant .prp file, under the CPP\_ReverseEngineering subject, add a metaclass with the name of the library (use the same name you used in the value of the DataTypesLibrary property).
- Under the new metaclass, add a property called DataTypes.
- For the value of the DataTypes property that you added, enter a comma-separated list of the types that you want to include for that library.
- Now, if you select the library from the drop-down list displayed on the Mapping tab, the types you defined with the DataTypes property is automatically added to the list of types that should be modeled as "Language" types.

*Default = MFC*

## ImportAsExternal

The ImportAsExternal property specifies whether the elements contained in the files you are reverse engineering should be brought into the model as "external" elements. This means that code is not generated for these elements during code generation.

This property corresponds to the Import as External check box on the Mapping tab of the Reverse Engineering Options window.

*Default = Cleared*

## ImportDefineAsType

The ImportDefineAsType property is a Boolean value that specifies how to import a #define. Note that models created before Version 5.2 automatically have this property overridden (set to True) when the model is loaded. The possible values are as follows:

- True - Import a #define as a user type.
- False - Import a #define as a constant variable, constant function, or type according to the following policy:
- If the #define has parameters, Rational Rhapsody creates a constant function. This applies to Rational Rhapsody Developer for C only.
- If the #define does not have parameters and its value includes only one line, Rational Rhapsody creates a constant variable. In Rational Rhapsody Developer for C++, the CG::Attribute::ConstantVariableAsDefine property is set to True.
- If the #define was not imported as a variable or function, Rational Rhapsody creates a type.

*Default = False*

## ImportGlobalAsPrivate

The ImportGlobalAsPrivate property allows you to import C functions as public or private.

The possible values are as follows:

- Never - Import globals (functions) as public. The declaration remains in the specification file.
- InImplementation - Global functions are imported as private. Both the declaration and the implementation of the function are imported into the implementation (.c) file.
- StaticInImplementation - Globals are imported as private in the implementation (.c) file and the functions are marked as static. (same as "InImplementation" but the keyword "static" is added to the declaration and implementation of the function).

## ImportPreprocessorDirectives

Before running reverse engineering or roundtrip operations, set the ImportPreprocessorDirectives property to preserve the order of preprocessor directives during code generation. This property is only used if the CPP\_Roundtrip:General:RoundtripScheme property is set to the "Respect" scheme.

However, the order of #include and #define is always preserved regardless of the setting of the ImportPreprocessorDirectives property.

*Default = Checked*

## ImportStructAsClass

The ImportStructAsClass property is a Boolean value specifies how structs in external code are imported during reverse engineering. The possible values are as follows:

- Checked - structs are imported as classes.
- Cleared - structs are imported as types of kind Structure.

*Default = Cleared*

## LocalizeRespectInformation

When reverse engineering code in Respect mode, Rational Rhapsody stores information such as the order of code elements so that when code is regenerated from the model, the code resembles as much as possible the original code.

When the LocalizeRespectInformation property is set to Checked, the software stores this information as SourceArtifact elements below the relevant class. (These elements are not visible by default, but you can see them in the model if you set the value of the ShowSourceArtifacts property to True.)

If the value of the LocalizeRespectInformation property is set to Cleared, then Rational Rhapsody stores this "respect" information as File elements under the relevant Component.

*Default = Checked*

## MacroExpansion

Early versions of Rational Rhapsody were not capable of importing macros in code such that they would be regenerated as macros. Rather, the code represented by the macro was stored in the model, and when the code was regenerated, the macro calls would be replaced with the relevant code.



Now, by default, Rational Rhapsody imports macros such that when the code is regenerated, the macro definition and macro calls are generated as they appeared in the original code that was reverse engineered.

If you would like the previous Rational Rhapsody behavior, that is, replacement of macro calls with the actual macro code, you can set the MacroExpansion property to Checked.

Note that the `CPP_ReverseEngineering::Parser::ForceExpansionMacros` property allows you to specify that individual macros should be expanded during reverse engineering even if the value of the MacroExpansion property is set to False.

*Default = Cleared*

## MapGlobalsToComponentFiles

The MapGlobalsToComponentFiles property allows you to specify whether Rational Rhapsody should map global variables, functions, and types to component files, reflecting the original file location of these elements in the files that were reverse engineered. The property can take any of the following values:

- True - All global variables, functions, and types should be mapped to component files
- OnExternal - Global variables, functions, and types should be mapped to component files only if the user selected the reverse engineering option "Import as External"
- TypesOnly - Global types should be mapped to component files, but not global variables and functions
- TypesOnExternal - Only global types should be mapped to component files, and this should only be done if the user selected the reverse engineering option "Import as External"
- False - Global variables, functions, and types should not be mapped to component files

*Default = True*

## MapToPackage

The MapToPackage property allows you to specify how the code elements you are reverse engineering should be divided into packages.

The property represents the options that appear in the Map to Package section of the Mapping tab in the Reverse Engineering Options window.

When the value of the property is set to Directory, a separate package is created for each subdirectory in the directory you have chosen to reverse engineer. The elements found in the files in each subdirectory is added to the package that corresponds to that subdirectory.

If you set the value of this property to User, then Rational Rhapsody places all of the reverse engineered elements into a single package in the model. The name of the package is taken from the property `CPP_ReverseEngineering::ImplementationTrait::UserPackage`.

*Default = Directory*

## ModelStyle

The `ModelStyle` property determines how model elements are opened in the browser after reverse engineering - by using a file-based functional approach or by using an object-oriented approach based on classes (the corresponding property values are `Functional` and `ObjectBased`).

This property corresponds to the Modeling Policy radio buttons on the Mapping tab of the Reverse Engineering Options window.

Note that for C++ and Java, the file-based approach can only be used for visualization purposes. Rational Rhapsody does not generate code from the model for elements imported that uses the `Functional` option. (On the Reverse Engineering Options window, you can only select the File radio button if you first select the Visualization Only option.)

*Default = Functional in RiC, ObjectBased in RiC++ and RiJ*

## PackageForExternals

If the value of the `UsePackageForExternals` property is set to `True`, the Rational Rhapsody reverse engineering feature puts all external elements in a separate package. You can control the name of this package by changing the value of the `PackageForExternals` property.

*Default = Externals*

## PreCommentSensibility

During reverse engineering, a comment that comes immediately before the code for an element is considered a comment for that element, and the comment text is brought into Rational Rhapsody as the description for that element.

The `PreCommentSensibility` property is used to specify the maximum number of lines by which a comment can precede the code for an element and still be considered a comment for that element. Any comment that precedes an element by more than the number of lines specified is considered a global comment.

A value of 1 means that a comment must appear on the line prior to the code for an element to be considered a comment for that element.

*Default = 2*

## ReflectDataMembers

The `ReflectDataMembers` property determines how the visibility of attributes is brought into the model when code is reverse engineered. The property affects both the visibility of the attribute in the regenerated code and the generation of get and set operations for the attribute. The property can take any of the following values:

- `None` - The visibility used for attributes is the same as that specified in the code that was reverse engineered. However, Rational Rhapsody generates public get/set operations for the attributes regardless of the visibility specified.
- `VisibilityOnly` - The visibility used for attributes is the same as that specified in the code that was

reverse engineered. In addition, Rational Rhapsody generates get/set operations for the attribute with the same visibility. For example, if the attribute visibility in the original code was private, the visibility is private in the regenerated code and the code also includes private get/set operations for the attribute.

- **VisibilityAndHelpers** - The visibility used for attributes is the same as that specified in the code that was reverse engineered. Rational Rhapsody does not generate get/set operations for the attribute if the original code did not contain such operations.

Note that when the property is set to **VisibilityAndHelpers**, get/set operations are not generated for attributes and Rational Rhapsody does not generate any of its automatically-generated operations, such as default constructors.

*Default = VisibilityAndHelpers*

## RespectCodeLayout

The **RespectCodeLayout** property determines to what degree Rational Rhapsody attempts to save information about the code that is reverse engineered so that it is possible to match the original code when code is later regenerated from the model. The saved information includes:

- order of #includes and other code elements
- handling of preprocessor directives such as #ifdefs
- keeping macro calls as they were rather than expanding the macro in the regenerated code
- handling of global comments

The property can take any of the following values:

- **None** - Rational Rhapsody does not save information about the order of elements in the code that is imported, nor does it save the information necessary to regenerate all elements back to the files from which they were originally imported.
- **Mapping** - Rational Rhapsody saves the partial information so that it can regenerate all elements back to the files from which they were originally imported.
- **Ordering** - Rational Rhapsody saves all of the information it can so that the regenerated code matches the original code as much as possible. See the examples listed above.

Note that even if the value of this property is set to **Ordering**, Rational Rhapsody only attempts to match the regenerated code to the original code if the property **CPP\_CG::Configuration::CodeGeneratorTool** is set to **Advanced**, which is the default value for that property.

*Default = Ordering*

## RootDirectory

This property specifies the root directory for reverse engineering. This root directory might contain all the folders that should become package during the reverse engineering process. Rational Rhapsody builds the package hierarchy according to the folder tree from the specified path.

*Default = empty string*

## RoundtripArgumentAsExistingType

Prior to version 8.0.3 of Rational Rhapsody, operation arguments whose type was a type defined in the model were imported into the model as "on-the-fly" types. Beginning in version 8.0.3, for CPP and Java code, if the type of the argument is a class defined in the model, the argument type is identified correctly when imported into the model. To preserve the previous code generation behavior for pre-8.0.3 models, the RoundtripArgumentAsExistingType property was added to the backward compatibility settings for C++ and Java with a value of False.

## RoundtripArgumentTypeAsType

Prior to version 8.0.3 of Rational Rhapsody, operation arguments whose type was a type defined in the model were imported into the model as "on-the-fly" types. Beginning in version 8.0.3, for C++ and Java code, if the type of the argument was a class defined in the model, the argument type was identified correctly when imported into the model. However, this was not the case for "types" such as structs. Beginning in version 8.1, for C++, also for arguments that are "types", the argument type is identified correctly when imported into the model.

To preserve the previous code generation behavior for pre-8.1 models, the RoundtripArgumentTypeAsType property was added to the backward compatibility settings for C++ with a value of False.

## RoundtripOperationReturnTypeAsExistingType

Prior to version 8.0.5 of Rational Rhapsody, operation return types which were types defined in the model were imported into the model as "on-the-fly" types. Beginning in version 8.0.5, for CPP and Java code, if the return type is a class defined in the model, the return type is identified correctly when imported into the model. To preserve the previous code generation behavior for pre-8.0.5 models, the RoundtripOperationReturnTypeAsExistingType property was added to the backward compatibility settings for C++ and Java with a value of False.

## RoundtripOperationReturnTypeAsType

Prior to version 8.0.5 of Rational Rhapsody, operation return types that were types defined in the model were imported into the model as "on-the-fly" types. Beginning in version 8.0.5, for C++ and Java code, if the return type was a class defined in the model, the return type was identified correctly when imported into the model. However, this was not the case for "types" such as structs. Beginning in version 8.1, also for return types that are "types", the return type is identified correctly when imported into the model.

To preserve the previous code generation behavior for pre-8.1 models, the RoundtripOperationReturnTypeAsType property was added to the backward compatibility settings for C++ with a value of False.

## UseCalculatedRootDirectory

This property controls the use of the `<lang>_ReverseEngineering::Implementation::RootDirectory` property.

The possible values are:

- Never - Do not calculate the root directory.
- Always - Calculate the root directory and override the RootDirectory property.
- Auto - Ask the user if they want to override the value in the RootDirectory property if it is different from the calculated root directory. If the RootDirectory property is empty, Rational Rhapsody uses the calculated value without asking.

*Default = Auto*

## UsePackageForExternals

When Rational Rhapsody generates code, it does not regenerate code for elements that have been brought in as "external" elements. By default, the reverse engineering feature puts all external elements in a separate package in the model. You can change this behavior by changing the value of the UsePackageForExternals property. When a separate package is used, the name of the package is taken from the value of the PackageForExternals property.

*Default = Checked*

## UserDataTypes

The UserDataTypes specifies classes to be modeled as data types. This property corresponds to types entered in the Add Type window.

*Default = empty string*

## UserPackage

When reverse engineering files, Rational Rhapsody allows you the option of having packages created for each subdirectory or having all of the reverse-engineered elements placed in a single package. This option is controlled by the property `CPP_ReverseEngineering::ImplementationTrait::MapToPackage`.

When MapToPackage is set to "User", you can use the UserPackage property to provide the name that you would like Rational Rhapsody to use for the single package that contains all of the reverse-engineered elements.

You can specify a nested package by using the following syntax: `package1::package2`

If the model already contains a package with the specified name, the reverse-engineered elements are put in that package. If not, Rational Rhapsody creates the package.

This property corresponds to the text field provided for the package name in the Map to Package section of the Mapping tab in the Reverse Engineering Options window.

*Default = ReverseEngineering*

## VisualizationUpdate

The VisualizationUpdate property instructs Rational Rhapsody to use the code-centric approach during reverse engineering and roundtripping. This approach assumes that the code serves as the blueprint for the software, and that the visual modeling capabilities of Rational Rhapsody are being used primarily to visualize the code.

In general, in code-centric mode, Rhapsody allows more drastic code changes to be brought into the model, relative to the changes that are imported when using the model-centric mode.

*Default = Checked (in code-centric settings)*

## Main

The metaclass Main contains properties that define the file extensions used for filtering files in the reverse engineering file selection window, as well as properties that enable jumping to problematic lines of code by double-clicking messages in the Output window.

## ErrorMessageTokensFormat

When errors are encountered during reverse engineering, they are displayed in the Rational Rhapsody Output window. If you double-click the error message, you are taken to the problematic line in the relevant source file.

This ability is made possible by the values provided for the ParseErrorMessage and ErrorMessageTokensFormat properties.

The value of the ParseErrorMessage property is a regular expression that extracts the relevant filename and line number information from the Rational Rhapsody-generated error message. The value of the ErrorMessageTokensFormat property is then used to interpret the information that was extracted from the error message.

The value of the ErrorMessageTokensFormat property consists of a comma-separated list of keyword-value pairs representing the number of tokens contained in the extracted information, which token represents the filename, and which token represents the line number.

Users should not change the value of this property.

*Default = ToTalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2*

## ImplementationExtension

The ImplementationExtension property specifies the file extensions used to filter the list of files displayed in the Reverse Engineering window.

If you want the Reverse Engineering window to also display files with no file extension, add the string "NoExtension" to the list of extensions, for example: c,cpp,cxx,cc,NoExtension.

*Default = c,cpp,cxx,cc*

## MakefileExtension

This property specifies the list of file extensions that are displayed and analyzed in the Reverse Engineering window when "Makefiles" is selected.

If you want the Reverse Engineering window to also display files with no file extension, add the string "NoExtension" to the list of extensions, for example: mak,mk,makefile,gpj,NoExtension.

Note that if you want Rational Rhapsody to import a makefile that is non-standard in terms of its content, you have to set the value of the CPP\_ReverseEngineering::MakefileImport::MakefileType property to UserDefined, and set the values of the properties under the CPP\_ReverseEngineering::MakefileUserDefined metaclass.

*Default = mak,mk,makefile,gpj*

## ParseErrorMessage

When errors are encountered during reverse engineering, they are displayed in the Rational Rhapsody Output window. If you double-click the error message, you are taken to the problematic line in the relevant source file.

This ability is made possible by the values provided for the ParseErrorMessage and ErrorMessageTokensFormat properties.

The value of the ParseErrorMessage property is a regular expression that extracts the relevant filename and line number information from the Rational Rhapsody-generated error message. The value of the ErrorMessageTokensFormat property is then used to interpret the information that was extracted from the error message.

Users should not change the value of this property.

*Default = "([a-zA-Z\_]+[:0-9a-zA-Z\_\\.\\])\*"[:][ ]\*LINE[ ]\*([0-9]+)*

## SpecificationExtension

The SpecificationExtension property is used to specify the filename extensions that should be used to filter files in the Reverse Engineering window. This property is used in conjunction with the

ImplementationExtension property.

You can specify a number of extensions. They should be entered as a comma-separated list.

If you want the Reverse Engineering window to also display files with no file extension, add the string "NoExtension" to the list of extensions, for example: h,hpp,hxx,inl,NoExtension.

*Default = h,hpp,hxx,inl*

## **useCodeCentricAbsolutePath**

For models that use code-centric mode, you can specify whether paths to the code files should be saved as absolute or relative paths. This setting is controlled by the property useCodeCentricAbsolutePath. Note that prior to release 8.1.2, all paths to the code files were saved as absolute paths.

*Default = False*

## **UseCodeCentricSettings**

The UseCodeCentricSettings property specifies whether the visualization result of running reverse engineering is in code-centric mode in Rational Rhapsody.

If this property is checked, the code-centric settings are added to the model during reverse engineering (if they do not already exist) and a dependency with the applied profile stereotype is added between the active component to the code-centric settings.

*Default = Cleared*

## **MFC**

The MFC metaclass contains a property that affects the MFC type library.

## **DataTypes**

The DataTypes property specifies classes to be modeled as MFC data types. There is only one predefined library (MFC) that contains only one class (CString). You can, however, expand this short list of classes by the addition of classes in this property or the creation of new libraries in the property files factory.prpfactory and site.prpsite.

*Default = CString*

## **MSVC60**



The MSVC60 metaclass contains properties used to control the Microsoft Visual C++ environment.

## Defined

The Defined property specifies symbols that are defined for the Microsoft Visual C++ version 6.0 (MSVC60) preprocessor. These symbols are automatically filled into the Name list of the Preprocessing tab of the Reverse Engineering Options window when you select Add > Dialect: MSVC60.

The default value is as follows: `__STDC__, __STDC_VERSION__, __cplusplus, __DATE__, __TIME__, __WIN32__, __cdecl__, __cdecl__, __int64=int, __stdcall, __export, __export, __AFX_PORTABLE__, __M_IX86=500, __declspec, __MSC_VER=1200, __inline=inline, __far__, __near__, __far__, __near__, __pascal, __pascal, __asm__, __finally=catch, __based, __inline=inline, __single_inheritance, __cdecl__, __int8=int, __stdcall, __declspec, __int16=int, __int32=int, __try=try, __int64=int, __virtual_inheritance, __except=catch, __leave=catch, __fastcall, __multiple_inheritance)`

## IncludePath

The IncludePath property specifies necessary include paths for the Microsoft Visual C++ preprocessor. It is possible to specify the path to the site installation of the compiler as part of the site.prp, thus doing it only once and not for every project.

*Default = empty string*

## Undefined

The Undefined property specifies symbols that must be undefined for the Microsoft Visual C++ preprocessor.

*Default = empty string*

## Parser

The metaclass Parser contains properties that can be used to modify the way the parser handles code during reverse engineering.

## AdditionalKeywords

The AdditionalKeywords property can be used to list non-standard keywords that might appear in the code that you reverse engineer. This allows Rational Rhapsody to parse this code correctly during reverse engineering.

The value of this property should be a comma-separated list of the additional keywords you want to

include.

Note that keywords with parameters are not supported, nor are keywords that consist of more than one word.

This property corresponds to the keywords listed on the Preprocessing tab of the Reverse Engineering Options window. Note that when you add additional keywords by using the controls on the Preprocessing tab, these keywords are included in the value of the AdditionalKeywords property at the level of the active configuration.

*Default = far,near*

## Defined

The Defined property specifies symbols and macros to be defined by using #define. For example, you can enter the following to define name> as text with the appropriate intermediate character: /D name{=|#}text

*Default = empty string*

## Dialects

The Dialects property specifies which symbols are added to the Preprocessing tab of the Reverse Engineering window box when that dialect is selected. The default value is MSVC60, which is itself defined by a metaclass of the same name under the CPP\_ReverseEngineering subject. This dialect specifies the symbols that are defined for the Microsoft Visual C++ environment. You can define your own dialect (in the site.prp file) and select it in the Dialects property. The default value for C is an empty string; the default value for C++ is MSVC60.

## ForceExpansionMacros

By default, Rational Rhapsody reverse engineers macros such that when the code is regenerated, the macro definition and macro calls are generated as they appeared in the original code that was reverse engineered. (This behavior can be controlled with the CPP\_ReverseEngineering::ImplementationTrait::MacroExpansion property.)

In some cases, you might find that you are not satisfied with the way that Rational Rhapsody imports the macro. For such situations, you can use the ForceExpansionMacros property to list specific macros that should be expanded during reverse engineering even if the value of the MacroExpansion property is set to False.

The value of this property should be a comma-separated list of the macros that you would like Rational Rhapsody to expand during reverse engineering.

*Default = Blank*

## IncludePath

The Preprocessing tab of the Reverse Engineering Options window allows you to specify an include path

(classpath for Java) for the parser to use. The `In property` `includePath` represents this path.

For the value of this property, you can enter a comma-separated list of directories. Note that you have to specify subdirectories individually.

The directories you list here is combined with the directories specified in `#include` statements in order to find the necessary files. For example, if you have `c:\d1\d2\d3\file.h`, you can enter `c:\d1\d2` as the value of this property and then use `d3\file.h` in the `#include` statement.

You should take into account that the value of this property also determines the structure of the source file directory when code is generated from the model. So, in the above example, the top-level directory created is `d3`.

*Default = Blank*

## Undefined

The Undefined property specifies symbols and macros to be undefined by using `#undef`.

*Default = empty string*

## Promotions

The metaclass Promotion contains a number of properties used to specify whether Rational Rhapsody should add various advanced modeling constructs to your model based on relationships/patterns uncovered during reverse engineering.

### EnableAttributeToRelation

The `EnableAttributeToRelation` property is used to specify whether Rational Rhapsody should add Associations to the model for attributes whose type is another class in the model.

For example, if you have two classes, A and B, and B contains an attribute of type A, Rational Rhapsody adds an Association to the model reflecting this relationship.

*Default = Checked*

### EnableFunctionToObjectBasedOperation

The `EnableFunctionToObjectBasedOperation` property specifies whether object-based promotion is enabled during reverse engineering. Object-based promotion “promotes” a global function to comply with the pattern specified in the `C_CG::Operation::PublicName` and `C_CG::Operation::ProtectedName` properties to be an operation of the class (`object_type`) defined in the `me` parameter for the function.

*Default = Cleared*

## EnableResolveIncompleteClasses

Sometimes, during reverse engineering, Rational Rhapsody is not able to find the base class for a given class. The EnableResolveIncompleteClasses property is used to specify that if Rational Rhapsody finds a class with the same name as the base class in a different location, it should assume that this class is the missing base class.

*Default = Checked*

## EnableTypeToTemplateInstantiation

During reverse engineering, when Rational Rhapsody encounters a typedef declaration that contains a template instantiation, the product by default creates a template instantiation in the model.

The EnableTypeToTemplateInstantiation property allows you to change this behavior. If you set the value of this property to False, then during reverse engineering Rational Rhapsody creates a Language type in the model rather than a template instantiation.

*Default = Checked*

## Update

The Update metaclass contains properties used to control various aspects of the Rational Rhapsody behavior during and after reverse engineering.

## CreateFlowcharts

Use this property to specify whether or not Rational Rhapsody should automatically create flowcharts for operations during reverse engineering of code.

Set this property before you start the reverse engineering process.

Use this property in conjunction with the FlowchartCreationCriterion, FlowchartMinLOC, FlowchartMaxLOC, FlowchartMinControlStructures and FlowchartMaxControlStructures properties so that flowcharts are created only for operations that are within a given range in terms of lines of code or in terms of the number of control structures in the operation.

*Default = Cleared*

## FlowchartActionFixedWidth

The FlowchartActionFixedWidth property can be used to control the width of action elements in flowcharts that are created automatically during reverse engineering or when you manually select the Populate Flowchart option.

When set to True, a constant width is used, regardless of the length of the code lines that are to be displayed inside.

When set to False, the width of the graphic element representing the action is expanded so that the contained lines of code are not split.

Note that when the property is set to False, the degree to which action elements will be expanded is limited by the value of the property FlowchartActionMaxWidthSize. You can modify the value of that property if necessary.

*Default = True*

## **FlowchartActionMaxWidthSize**

When the value of the property FlowchartActionFixedWidth is set to False, the width of the graphic element representing an action is expanded so that the contained lines of code are not split. However, the degree to which action elements will be expanded is limited by the value of the property FlowchartActionMaxWidthSize.

The value of FlowchartActionMaxWidthSize should be the maximum number of pixels you want to allow for the width of the text portion of the box.

*Default = 500*

## **FlowchartCreationCriterion**

If you have set the CreateFlowcharts property to have Rational Rhapsody create flowcharts during reverse engineering, you can use the FlowchartCreationCriterion property to select the criterion that should be used to decide what operations Rational Rhapsody should create flowcharts for.

The property can take the following values:

- **Control Structures** - the decision whether or not to generate a flowchart for an operation is based on the number of control structures in the operation. When this option is selected, the minimum and maximum number of control structures used to define the inclusion criterion are taken from the FlowchartMinControlStructures and FlowchartMaxControlStructures properties.
- **LOC** - the decision whether or not to generate a flowchart for an operation is based on the number of lines of code in the operation. When this option is selected, the minimum and maximum lines of code used to define the inclusion criterion are taken from the FlowchartMinLOC and FlowchartMaxLOC properties.

*Default = LOC*

## **FlowchartMaxControlStructures**

If you have set the CreateFlowcharts property to have Rational Rhapsody create flowcharts during reverse engineering, and you have set the value of the FlowchartCreationCriterion property to Control Structures, then you can use the FlowchartMaxControlStructures property to specify the maximum number of control structures that an operation can have, above which Rational Rhapsody does not create a flowchart for it.

*Default = 10*

## **FlowchartMaxLOC**

If you have set the CreateFlowcharts property to have Rational Rhapsody create flowcharts during reverse engineering, and you have set the value of the FlowchartCreationCriterion property to LOC, then you can use the FlowchartMaxLOC property to specify the maximum number of lines of code that an operation can have, above which Rational Rhapsody does not create a flowchart for it.

*Default = 100*

## **FlowchartMinControlStructures**

If you have set the CreateFlowcharts property to have Rational Rhapsody create flowcharts during reverse engineering, and you have set the value of the FlowchartCreationCriterion property to Control Structures, then you can use the FlowchartMinControlStructures property specify the minimum number of control structures that an operation must have in order to have Rational Rhapsody create a flowchart for it.

*Default = 2*

## **FlowchartMinLOC**

If you have set the CreateFlowcharts property to have Rational Rhapsody create flowcharts during reverse engineering, and you have set the value of the FlowchartCreationCriterion property to LOC, then you can use the FlowchartMinLOC property to specify the minimum number of lines of code that an operation must have in order to have Rational Rhapsody create a flowchart for it.

*Default = 10*

# CPP\_Roundtrip

---

The CPP\_Roundtrip subject contains metaclasses that contain properties that affect roundtripping.

## General

The General metaclass contains properties that control how changes to code are roundtripped in Rational Rhapsody.

### AnalyzePeripheralIncludes

When the AnalyzePeripheralIncludes property is set to True, Rhapsody uses information from files specified as #includes in the current file in order to correctly interpret #ifdef and other preprocessor directives.

*Default = True*

### CancelIfReadOnly

Set this property to Checked in order to cancel Roundtripping if there are any Read Only (Checked-In) units that are related to one of the files to be roundtripped.

By default, Roundtrip changes only Read/Write units and provides warnings in the Output window regarding any attempted changes to Read Only units.

*Default = name or rename*

### CreateFileAsUnit

When a File is created in a model, the value of the General::Model::FileIsSavedUnit property determines whether or not it is saved as a unit (that is, as a separate file in the file system).

The CreateFileAsUnit property provides you with a certain degree of flexibility during roundtripping so that a File created during roundtripping can be saved as a unit even if the value of the FileIsSavedUnit property is set to False.

The possible values for this property are:

- Default - The decision whether or not to save the newly-created File as a unit is based on the value of the FileIsSavedUnit property.
- AsModel - The decision whether or not to save the newly-created File as a unit depends on the class in the model that it represents. If the class is currently saved as a unit, then the File created during roundtripping is also saved as a unit. If the class is not a unit, then the File created is also not saved as a unit.

*(Default = AsModel)*

## CreateFolderAsUnit

When a Folder is created in a model, the value of the General::Model::FolderIsSavedUnit property determines whether or not it is saved as a unit (that is, as a separate file in the file system).

The property CreateFolderAsUnit provides you with a certain degree of flexibility during roundtripping so that a Folder created during roundtripping can be saved as a unit even if the value of the FolderIsSavedUnit property is set to False.

The possible values for this property are:

- Default - The decision whether or not to save the newly-created Folder as a unit is based on the value of the FolderIsSavedUnit property.
- AsModel - The decision whether or not to save the newly-created Folder as a unit depends on the package in the model that it represents. If the package is currently saved as a unit, then the Folder created during roundtripping is also saved as a unit. If the package is not a unit, then the Folder created is also not saved as a unit.

*(Default = AsModel)*

## NotifyOnInvalidatedModel

The NotifyOroundnInvalidatedModel property is a Boolean value that determines whether a warning window is displayed during a round trip. This warning is displayed when information might get lost because the model was changed between the last code generation and the round trip operation.

*(Default = Checked)*

## ParserErrors

The ParserErrors property specifies the roundtripping behavior when a parser error is encountered. The possible values are as follows:

- Abort - Stop the round trip whenever there is a parser error in the code. No changes is applied to the model.
- AskUser - When Rational Rhapsody encounters an error, the program displays a message asking what you want to do.
- AbortOnCritical - Stop roundtripping if any critical parser errors are encountered in the code.
- Ignore - Continue roundtripping and skip any parser errors that are encountered.

*(Default = AskUser)*

## PredefineIncludes

The PredefineIncludes property specifies the predefined include path for roundtripping.



(Default = empty string)

## PredefineMacros

The PredefineMacros property specifies the predefined macros for roundtripping. The default value is as follows:

```
DECLARE_META(class_0\,animClass_0), DECLARE_REACTIVE_META(class_0\,animClass_0),
OMINIT_SUPERCLASS(class_0Super\,animClass_0Super),
OMREGISTER_CLASS\,DECLARE_META_T(class_0\, ttype\,animClass_0),
DECLARE_REACTIVE_META_T(class_0\, ttype\,animClass_0),
DECLARE_META_SUBCLASS_T(class_0\, ttype\,animClass_0),
DECLARE_REACTIVE_META_SUBCLASS_T(class_0\, ttype\,animClass_0),
DECLARE_MEMORY_ALLOCATOR(CLASSNAME\,INITNUM),
IMPLEMENT_META(class_0\,Default\,FALSE),
IMPLEMENT_META_S(class_0\,FALSE\,class_1\,animClass_1\,animClass_0),
IMPLEMENT_META_M(class_0\, FALSE\, class_0Super\, 2 \,animClass_0),
IMPLEMENT_REACTIVE_META(class_0\,Default\,FALSE),
IMPLEMENT_REACTIVE_META_S(class_0\,FALSE\,class_1\,animClass_1\,animClass_0),
IMPLEMENT_REACTIVE_META_M(class_0\, FALSE\, class_0Super\, 2 \,animClass_0),
IMPLEMENT_REACTIVE_META_SIMPLE(class_0\,Default\,FALSE),
IMPLEMENT_REACTIVE_META_S_SIMPLE(class_0\,FALSE\,class_1\,animClass_1\,animClass_0),
IMPLEMENT_REACTIVE_META_M_SIMPLE(class_0\, FALSE\, class_0Super\, 2 \,animClass_0),
IMPLEMENT_META_T(class_0\, Default\, FALSE\, animClass_0),
IMPLEMENT_META_S_T(class_0\,FALSE\,class_0Super\,animclass_0Super\,animClass_0),
IMPLEMENT_META_M_T(class_0\, FALSE\, class_0Super\, 2 \,animClass_0),
IMPLEMENT_META_T_S_T(tname\,IsSingleton\,SuperClass\,animSuperClass\,animTname),
IMPLEMENT_META_T_S_T_N(tname\,IsSingleton\,NameSpace\,SuperClass\,animSuperClass\,animTname),
IMPLEMENT_META_OBJECT(class_0\,class_type\,Default\,FALSE),
IMPLEMENT_META_S_OBJECT(class_0\,class_type\,FALSE\,class_1\,animClass_1\,animClass_0),
IMPLEMENT_META_M_OBJECT(class_0\,class_type\,FALSE\, class_0Super\, 2 \,animClass_0),
IMPLEMENT_REACTIVE_META_OBJECT(class_0\,class_type\,Default\,FALSE),
IMPLEMENT_REACTIVE_META_S_OBJECT(class_0\,class_type\,FALSE\,class_1\,animClass_1\,animClass_0),
IMPLEMENT_REACTIVE_META_M_OBJECT(class_0\,class_type\,FALSE\, class_0Super\, 2
\,animClass_0),
IMPLEMENT_REACTIVE_META_SIMPLE_OBJECT(class_0\,class_type\,Default\,FALSE),
IMPLEMENT_REACTIVE_META_S_SIMPLE_OBJECT(class_0\,class_type\,FALSE\,class_1\,animClass_1\,animClass_0),
IMPLEMENT_REACTIVE_META_M_SIMPLE_OBJECT(class_0\,class_type\,FALSE\, class_0Super\,
2 \,animClass_0), IMPLEMENT_META_T_OBJECT(class_0\,class_type\, Default\, FALSE\,
animClass_0),
IMPLEMENT_META_S_T_OBJECT(class_0\,class_type\,FALSE\,class_0Super\,animclass_0Super\,animClass_0),
IMPLEMENT_META_M_T_OBJECT(class_0\,class_type\, FALSE\, class_0Super\, 2 \,animClass_0),
IMPLEMENT_MEMORY_ALLOCATOR(CLASSNAME\,INITNUM\,INCREMENTNUM\,ISPROTECTED),
DECLARE_META_PACKAGE(Default), DECLARE_PACKAGE(Default),
IMPLEMENT_META_PACKAGE(Default\,Default), DECLARE_META_EVENT(event_0),
DECLARE_META_SUBEVENT(event_0\,event_0Super\,event_0SuperNamespace),
IMPLEMENT_META_EVENT(event_0\,Default\,event_0), IMPLEMENT_META_EVENT_S(words\,
words\, baseWords), DECLARE_OPERATION_CLASS(mangledName),
DECLARE_META_OP(mangledName), OM_OP_UNSER(type\, name), OP_UNSER(func\, name),
OP_SET_RET_VAL(retVal), OM_OP_SET_RET_VAL(retVal),
IMPLEMENT_META_OP(animatedClassName\, mangledName\, opNameStr\, isStatic\, signatureStr\,
numOfArgs), IMPLEMENT_OP_CALL(mangledName\, userClassName\, call\, retExp),
STATIC_IMPLEMENT_OP_CALL(mangledName\, userClassName\, call\, retExp),
```

OMDefaultThread=0, NULL=0, OMDECLARE\_GUARDED, OM\_DECLARE\_COMPOSITE\_OFFSET

## ReportChanges

The ReportChanges property defines which changes are reported (and displayed) by the roundtrip operation. The possible values are as follows:

- None - No changes are displayed in the output window.
- AddRemove - Only the elements added to, or removed from, the model are displayed in the output window.
- UpdateFailures - Code changes that could not be updated in the model are displayed in the output window, as well as elements added to, or removed from, the model.
- All - All changes to the model are displayed in the output window.

*Default = AddRemove*

## RestrictedMode

The RestrictedMode property is a Boolean value that specifies whether restricted-mode roundtripping is enabled. This property can be modified on the configuration level.

Restricted mode of full roundtrip roundtrips unusual usage of Rational Rhapsody elements, such as a class declaration in a user-defined type. Restricted mode has more limitations, but preserves the model from unexpected changes. The additional limitations for restricted mode are as follows:

- User-defined types cannot be removed or changed on roundtrip because Rational Rhapsody code generation adds the Ignore annotation for a user-defined type declaration.
- Relations cannot be removed or changed on roundtrip.
- New classes are not added to the model.

*(Default = name or rename)*

## RoundtripNewFiles

When working in Eclipse platform integration, use this property to set Roundtrip policy for the files that the user added to the Eclipse project and are connected to the active Rational Rhapsody Eclipse Configuration. The possible values are as follows:

- AskUser - A message is displayed and asks the user whether or not to Roundtrip the new file.
- Always - The new file is roundtripped into a Rational Rhapsody model with no message.
- Never - The new file is never roundtripped into a Rational Rhapsody model without a message.

*(Default = AskUser)*

## RoundtripPreprocessorDirectives

By default, the Rational Rhapsody roundtripping feature takes into account changes made to preprocessor directives. The RoundtripPreprocessorDirectives property can be used to turn off roundtripping for the following types of preprocessor directives:

- `elif`
- `else`
- `endif`
- `error`
- `if`
- `ifdef`
- `ifndef`
- `import`
- `line`
- `pragma`
- `undef`
- `using`

*(Default = Checked)*

## RoundtripScheme

Determines what type of changes can be roundtripped back into the model. The possible values are Basic, Advanced, and Respect.

When set to Basic, only changes to the bodies of operations and actions are roundtripped into the model.

When set to Advanced, roundtripping also takes into account elements that have been added, such as attributes and operations, and can optionally take into account elements that have been modified or removed.

When set to Respect, roundtripping also takes into account the changes that are covered by the Rational Rhapsody code preserving feature, for example, the order of class members or elements like `#include-s`.

(Note that for pre-7.1 models, the possible values are Basic and Full, where Basic roundtrips only changes to the bodies of operations and actions, and Full represents the roundtripping mode currently referred to as Advanced.)

*Default = Respect*

## Update

The Update metaclass contains a property that controls the update process used during roundtripping.

## AcceptChanges

The AcceptChanges property is an enumerated type that specifies which changes are applied to each CG element (attribute, operation, type, class, or package). You can apply separate properties to each type of CG element. The possible values are as follows:

- All - All the changes can be applied to the model element, including deletion.
- Default is 1) Rational Rhapsody does not roundtrip deletions if the updated code results in parser errors. 2) Rational Rhapsody does not roundtrip the deletion of classes.
- NoDelete - All the changes except deletion can be applied to the model element. This setting prevents accidental removal of operations, constructors, attributes, relations, variables, instances, and functions.
- AddOnly - Apply only the addition of an aggregate to the model element. You cannot delete or change elements.
- NoChanges - Do not apply any changes to the model element.

Note that the value of the property is propagated to all the aggregates of an element. Therefore, if a package has the NoChanges value, no elements in that package is changed.

*Default = "Default" (in code-centric settings, default value is All)*

## MergePolicy

The MergePolicy property determines the merge policy Rational Rhapsody uses during roundtripping when comparing the updated code to the saved model. When the value of this property is CodeDriven, Rational Rhapsody imports certain types of code elements that it does not import when working in model-centric mode.

This property differs from the AcceptChanges property in that AcceptChanges deals with changes to model elements (adding, deleting, modifying) while MergePolicy is used as a general indication for Rational Rhapsody that the code, rather than the model, should be given precedence when it comes to merging changes.

*Default = CodeDriven (in code-centric settings)*

## ReplaceRTFDescription

The property ReplaceRTFDescription is used to determine whether element descriptions containing RTF are overwritten with plain text when changes to comments are roundtripped into the model.

When set to False, changes to comments are not brought into the model if the existing element description contains RTF.

When set to True, the updated comment is taken from the code and the RTF in the existing description is replaced with plain text.

Note that in code-centric mode, the updated comment is always brought into the model, regardless of the value of this property.

*Default = False*

The C CG subject contains metaclasses for operating system environments in addition to general metaclasses for code generation.

## Argument

The Argument metaclass contains properties that control how arguments are generated in code.

### DeclarationModifier

The DeclarationModifier property is used to allow Rational Rhapsody to reverse engineer non-standard keywords that appear in argument declarations. Keywords that appear between the argument type and the argument name are stored as the value of this property, and the property is then used during code generation to re-create the original code.

Since this is a code generation property, it can also be used to add non-standard keywords to code even when reverse engineering is not used.

This property is used in conjunction with the PreDeclarationModifier and PostDeclarationModifier properties.

*Default = Blank*

### DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- \$Name - the element name
- \$FullName - the full path of the element (P1::P2::C.a)
- \$Description - the text provided on the Description tab of the Features window for the element
- \$Requirements - There is a global option on the Settings tab of the Features window for configurations

that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the \$Requirements keyword in the DescriptionTemplate property to get such comments generated for specific model elements.

- \$Login - the login name of the user who generated the file
- \$CodeGeneratedDate - the date on which the code was generated
- \$CodeGeneratedTime - the time at which the code was generated
- \$RhapsodyVersion - the version of Rhapsody that was used to generate the file
- For operations: \$Signature - the operation signature, \$Arguments - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: \$Type - the type of the attribute
- For arguments: \$Type - the argument type, \$Direction - the argument direction (In, Out, or InOut)
- For events: \$Arguments - the text provided on the Description tab of the Features window for the event arguments
- For association ends: \$Target - the other end of the association
- For requirements: \$ID - the ID specified for the requirement, \$Specification - the specification text for the requirement
- The values of tags can be included by using the syntax \$tag\_name. For example, if you use a tag called author for your classes, you can use the keyword \$author in the value of DescriptionTemplate in order to have the value of the author tag included in the element description.
- The values of properties can be included by using the syntax \$property\_name. For example, you can use the keyword \$Animate in the value of DescriptionTemplate in order to have the value of the Animate property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as \$Name)
- Tag keywords
- Property keywords

If the value of a keyword is a MultiLine value, then each new line (except the first one) starts with the value of the <lang>\_CG::Configuration::DescriptionBeginLine property, and each line ends with the value of the <lang>\_CG::Configuration::DescriptionEndLine property.

*Default = Blank*

## IsRegister

The IsRegister property can be used to specify that the keyword "register" should be generated in the code for a given argument.

*Default = Cleared*

## IsVolatile

The IsVolatile property allows you to specify that a specific operation argument should be declared as volatile.

*Default = Cleared*

## PostDeclarationModifier

The PostDeclarationModifier property is used to enable Rational Rhapsody to reverse engineer non-standard keywords that appear in argument declarations. Keywords that appear after the argument name are stored as the value of this property, and the property is then used during code generation to re-create the original code.

Since this is a code generation property, it can also be used to add non-standard keywords to code even when reverse engineering is not used.

This property is used in conjunction with the PreDeclarationModifier and DeclarationModifier properties.

*Default = Blank*

## PreDeclarationModifier

The PreDeclarationModifier property is used to enable Rational Rhapsody to reverse engineer non-standard keywords that appear in argument declarations. Keywords that appear before the argument type are stored as the value of this property, and the property is then used during code generation to re-create the original code.

Since this is a code generation property, it can also be used to add non-standard keywords to code even when reverse engineering is not used.

This property is used in conjunction with the DeclarationModifier and PostDeclarationModifier properties.

*Default = Blank*

## PrintName

When an operation argument is not accessed in the operation body, some compilers issue a warning. The PrintName property allows you to avoid such warnings by having the generated code include only the argument type but not the argument name.

If the property is set to True, the argument name is included in the generated code. If the property is set to False, only the argument type is included in the generated code.

*Default = True*

## Attribute



The Attribute metaclass contains properties that control attributes of code generation, such as whether to generate accessor operations.

## AccessorGenerate

The AccessorGenerate property specifies whether to generate accessor operations for attributes. The possible values are:

- Checked - A get() method is generated for the attribute.
- Cleared - A get() method is not generated for the attribute..

Setting this property to Cleared is one way to optimize your code for size.

*Default = Cleared*

## AccessorVisibility

The AccessorVisibility property specifies the access level of the generated accessor for attributes. This property defines the access level of an accessor operation regardless of the visibility of the attribute. The possible values are as follows:

- fromAttribute - Use the access level for the attribute for the accessor.
- public - Set the access level to public for the accessor.
- private - Set the access level to private for the accessor.
- protected - Set the access level to protected for the accessor. This value is not available in Rational Rhapsody Developer for C.

*Default = fromAttribute*

## AttributeInitializationFile

The AttributeInitializationFile property specifies how static const attributes are initialized. In Rational Rhapsody, you can initialize these attributes in the specification file or directly in the initialization file. This property is analogous to the VariableInitializationFile property for global const variables. The possible values are as follows:

- Default - The attribute is initialized in the specification file if the type declaration begins with const. Otherwise, the variable is initialized in the implementation file.
- Implementation - Initialize constant attributes in the implementation file.
- Specification - Initialize constant attributes in the specification file.

*Default = Default*

## AttributePrivateQualifier

The AttributePrivateQualifier property specifies the qualifier that is printed at the beginning of the declaration of the struct member that is generated for a private attribute. If you don't want any qualifier to

be generated, you can set the value of the property to an empty string.

*Default = Blank*

## **AttributePublicQualifier**

The AttributePublicQualifier property specifies the qualifier that is printed at the beginning of the declaration of the struct member that is generated for a public attribute. If you don't want any qualifier to be generated, you can set the value of the property to an empty string.

*Default = Blank*

## **BitField**

Allows you to define a bit field for an attribute. To define a bit field, open the Features window for the relevant attribute and enter the number you want to use for the bit field as the value of the BitField property. For example, if you enter 2 as the value of BitField for an attribute named attribute\_1 of type int, the resulting code is:

```
int attribute_1 : 2;
```

## **ConstantVariableAsDefine**

The ConstantVariableAsDefine property is a Boolean value that determines whether the variable, defined as constant in file or package, is generated by using a #define macro. Otherwise, it is generated by using the const qualifier.

*Default = Checked*

## **DeclarationModifier**

The DeclarationModifier property is used to enable Rational Rhapsody to reverse engineer non-standard keywords that appear in attribute declarations. Keywords that appear between the attribute type and the attribute name are stored as the value of this property, and the property is then used during code generation to re-create the original code.

Since this is a code generation property, it can also be used to add non-standard keywords to code even when reverse engineering is not used.

This property is used in conjunction with the PreDeclarationModifier and PostDeclarationModifier properties.

*Default = Blank*

## **DeclarationPosition**

The DeclarationPosition property controls the declaration order of attributes. The possible values are as follows:

- Default - Similar to the AfterClassRecord setting, with the following difference:
- For static attributes defined in a class with the C\_CG::Attribute::Visibility property set to Public, these attributes are generated after types whose C\_CG::Type::Visibility property is set to Public.
- You should not use this setting for new models.
- BeforeClassRecord - Generate the attribute immediately before the class record.
- AfterClassRecord - Generate the attribute immediately after the class record.
- StartOfDeclaration - Generate the attribute immediately after the start of the section (private or public part of the specification, or package body).
- EndOfDeclaration - Generate the attribute immediately before the end of the section (private or public part of the specification, or package body).

*Default = Default*

## DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- \$Name - the element name
- \$FullName - the full path of the element (P1::P2::C.a)
- \$Description - the text provided on the Description tab of the Features window for the element
- \$Requirements - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the \$Requirements keyword in the DescriptionTemplate property to get such comments generated for specific model elements.
- \$Login - the login name of the user who generated the file
- \$CodeGeneratedDate - the date on which the code was generated
- \$CodeGeneratedTime - the time at which the code was generated
- \$RhapsodyVersion - the version of Rhapsody that was used to generate the file
- For operations: \$Signature - the operation signature, \$Arguments - the text provided on the Description tab of the Features window for the operation arguments

- For attributes: \$Type - the type of the attribute
- For arguments: \$Type - the argument type, \$Direction - the argument direction (In, Out, or InOut)
- For events: \$Arguments - the text provided on the Description tab of the Features window for the event arguments
- For association ends: \$Target - the other end of the association
- For requirements: \$ID - the ID specified for the requirement, \$Specification - the specification text for the requirement
- The values of tags can be included by using the syntax \$tag\_name. For example, if you use a tag called author for your classes, you can use the keyword \$author in the value of DescriptionTemplate in order to have the value of the author tag included in the element description.
- The values of properties can be included by using the syntax \$property\_name. For example, you can use the keyword \$Animate in the value of DescriptionTemplate in order to have the value of the Animate property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as \$Name)
- Tag keywords
- Property keywords

If the value of a keyword is a MultiLine value, then each new line (except the first one) starts with the value of the <lang>\_CG::Configuration::DescriptionBeginLine property, and each line ends with the value of the <lang>\_CG::Configuration::DescriptionEndLine property.

*Default = Blank*

## GenerateAccessorMacro

The GenerateAccessorMacro property specifies whether to generate accessor macros for attributes (including direct-relations attributes).

Rational Rhapsody provides macros for accessing an attribute without having to specify the memory segment to which it is assigned.

For example, if you keep the default value of the property, \$class\_\$attribute, the code generated will resemble: #define file\_0\_variable\_0 file\_0\_Seg1.variable\_0.

The provided macros are for use inside the class only. For accessing attributes from outside the class, the user can use the accessor/mutator operations.

The possible values are:

- Always - Always generate accessor macros.
- Never - Never generate accessor macros.
- WhenSegmentedMemoryIsEnabled - Generate accessor macros only when the CG::Configuration::EnableSegmentedMemory property is set to Checked.

*Default = WhenSegmentedMemoryIsEnabled*

## GenerateVariableHelpers

By default, Rational Rhapsody generates getter and setter methods for class attributes, but not for global variables. If you want Rational Rhapsody to generate getter and setter methods for global variables, set the value of the GenerateVariableHelpers property to True.

*Default = Cleared*

## ImplementationEpilog

The ImplementationEpilog property enables you to add any code that you want to be added as verbatim text (to be ignored by Rational Rhapsody) to the end of the definition of a model element..

For example, you could wrap a section of code with an `#ifdef-#endif` pair, add compiler-specific keywords, or add a `#pragma` statement. For example, to specify that an operation is available only when the code is compiled with `_DEBUG`, set the following properties for the operation:

- Set SpecificationProlog to `#ifdef _DEBUG cr.`
- Set SpecificationEpilog to `#endif.`
- Set ImplementationProlog to `#ifdef _DEBUG cr.`
- Set ImplementationEpilog to `#endif.`

The following table lists whether leading and trailing linefeeds are generated.

Metaclass Leading Linefeed Added? Class Yes Package No

*Default = Empty MultiLine*

## ImplementationName

The ImplementationName property enables you to give an attribute one model name and generate it with another name.

*Default = Empty string*

## ImplementationProlog

The ImplementationProlog property adds any code that you want to be added as verbatim text (to be ignored by the software) to the beginning of the definition of a model element.

For example, you could wrap a section of code with an `#ifdef-#endif` pair, add compiler-specific keywords, or add a `#pragma` statement. For example, to specify that an operation is available only when the code is compiled with `_DEBUG`, set the following properties for the operation:

- Set SpecificationProlog to `#ifdef _DEBUG cr.`

- Set SpecificationEpilog to #endif.
- Set ImplementationProlog to #ifdef \_DEBUG cr.
- Set ImplementationEpilog to #endif.

The following table lists whether leading and trailing linefeeds are generated.

Metaclass Trailing Linefeed Added? Class No Package Yes

*Default = Empty MultiLine*

## InitializationStyle

The InitializationStyle property specifies the initialization style used for attributes. When you specify an initial value for an attribute, Rational Rhapsody initializes the attribute based on the value of this property. In Rational Rhapsody Developer for C++, the possible values are:

- ByInitializer - Initialize the attribute in the initializer (a(y)). This is the default value.
- If the initialization style is ByInitializer, the attribute initialization should be done after the user initializer, in the same order as the order of attributes in the code.
- ByAssignment - Initialize the attribute in the constructor body (a = y).

In Rational Rhapsody Developer for C, the attribute is initialized in the initializer body.

*Default = ByInitializer*

## Inline

The Inline property specifies how inline operations are generated. Which operations are affected by the Inline property depends on the metaclass:

- Attribute - Applies only to operations that handle attributes (such as accessors and mutators)
- Operation - Applies to all operations
- Relation - Applies only to operations that handle relations

Inlining in Rational Rhapsody Developer for C You can inline attribute and relation accessors and mutators for increased code performance. The visibility of the inlined operations can be either public or private. The accessor can contain only a return statement, or more than just a return statement.

For Rational Rhapsody Developer for C, there are two possible settings for this property:

none - The operation is not generated inline. This is the default. Example of "none": /\* Mutator of Tank::ItsDishwasher relation \*/ void Tank\_setItsDishwasher(struct Tank\_t\* const me, struct Dishwasher\_t\* p\_Dishwasher) { if(p\_Dishwasher != NULL) Dishwasher\_\_setItsTank(p\_Dishwasher, me); Tank\_\_setItsDishwasher(me, p\_Dishwasher); } /\* Accessor to Tank::ItsDishwasher relation \*/ struct Dishwasher\_t\* Tank\_getItsDishwasher( const struct Tank\_t\* const me) { return (struct Dishwasher\_t\*)me->itsDishwasher; }

The second possible setting is "in\_header" to indicate that the operation is generated inline. Mutators are defined as macro definitions.

```
Example of "in_header": /* Inline Mutator of Tank::ItsDishwasher relation */ #define
Tank_setItsDishwasher(me, p_Dishwasher) \ { \ if((p_Dishwasher) != NULL) \
Dishwasher__setItsTank((p_Dishwasher), (me)); \ Tank__setItsDishwasher((me), (p_Dishwasher)); \ }
```

Accessors that contain only return statements are defined as macros (the return statement and the semicolon at the end of expression are omitted); other accessors are generated as operations. For example:

```
/* Inline Accessor to Tank::ItsDishwasher relation */ #define Tank_getItsDishwasher(me)
((me)-itsDishwasher)
```

If the attribute visibility is defined as Private, the macro definitions are placed in the implementation (.c) file.

If the attribute visibility is defined as Public, the macro definitions are placed in the specification (.h) file.

Note the following:

- Each instance of the parameters for the macro is parenthesized.
- You cannot inline an accessor if it contains statements other than the return statement. For example, accessors to relations implemented by using RiCCollection cannot be generated as function-like macros.
- You cannot set the Inline property separately for specific helpers (for example, only mutators) - this property affects all helpers of the attribute or relation.
- If a multi-lined mutator macro is called as the body of the “then” part of an “if ...else” statement, you must enclose it in parentheses or it generates a compilation error. For example:

```
// Erroneous code: If (itsDishwasher != NULL) Tank_setItsDishwasher(me, itsDishwasher); Else Return;
// Correct code: If (itsDishwasher != NULL) { Tank_setItsDishwasher(me, itsDishwasher); } Else Return;
```

## IsAliased

The IsAliased property is a Boolean value that specifies whether attributes are aliased.

*Default = Cleared*

## IsMutable

The IsMutable property is a Boolean value that allows you to specify that an attribute is a mutable attribute.

*Default = Cleared*

## IsVolatile

The IsVolatile property allows you to specify that an attribute should be declared as volatile.

*Default = Cleared*

## Kind

The Kind property specifies the kind of operation that should be generated for an element. The kind of operations that can be generated is language-dependent (for example, virtual and abstract exist only in C++ and Java). In Java, Kind can be defined only for attributes and operations, but not for relations.

This property affects class operations, in addition to accessors and mutators for relations and attributes. The possible values are:

- common - Class operations and accessor/mutator are non-virtual.
- virtual - Class operations and accessor/mutator are virtual. This type is valid for C++ and Java only.
- abstract - Class operations and accessor/mutator are pure virtual. This type is valid for C++ and Java only.

*Default = common*

## MarkPrologEpilogInAnnotations

The MarkPrologEpilogInAnnotations property specifies whether to generate ignore annotations for the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties so they are ignored during roundtrip. When you insert code element declarations (variables, types, functions, and so on) in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, after a full roundtrip those elements are added to the model and are duplicated on the next code generation.

When you use the MarkPrologEpilogInAnnotations property, you can have Rational Rhapsody automatically ignore the information specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties instead of adding the ignore annotations manually. The possible values for the MarkPrologEpilogInAnnotations property are:

- None - Rational Rhapsody does not generate any annotations. Any models created before Version 4.1 automatically have this property setting.
- Ignore - Rational Rhapsody generates the `/// ignore annotation before the code specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, and generates the /// annotation after the code specified in those properties.`
- Auto - If the code in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties is one line (it does not contain any newline characters (`\n`)), no annotations are generated (the same behavior has the None setting). If there is more than one line, Rational Rhapsody generates the `/// ignore annotation before the code specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, and generates the /// annotation after the code specified in those properties (the same behavior as the Ignore setting).`

During roundtrip, any ignore annotations in the comments of the element are not included in its description. Because the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties are generated between the annotation and its declaration for the element, you cannot rename those elements on roundtrip. If you change the name of an element, it is removed from the model and added with the new name. Some model information (for example, property settings) might be lost.



*Default = Auto*

## MemorySegmentImplementationProlog

The MemorySegmentImplementationProlog property lets you code as verbatim text (to be ignored by Rational Rhapsody) to the end of the data declaration related to a Memory Segment. Use this property to define a memory segment using a stereotype. In addition, see the following properties:

- C\_CG::Attribute::MemorySegmentSpecificationProlog
- C\_CG::Attribute::MemorySegmentSpecificationEpilog
- C\_CG::Attribute::MemorySegmentImplementationEpilog
- C\_CG::Attribute::SegmentedAttributeMacroName
- C\_CG::Attribute:MemorySegmentName

This property is valid only when the Segmented Memory capability is enabled (see CG::Configuration::EnableSegmentedMemory).

*Default = empty MultiLine*

Empty line is needed when updating Memory segment properties.

## MemorySegmentImplementationEpilog

The MemorySegmentImplementationEpilog property lets you add code as verbatim text (to be ignored by Rational Rhapsody) to the end of the data declaration related to a Memory Segment. Use this property to define a memory segment using a stereotype. In addition, see the following properties:

- C\_CG::Attribute::MemorySegmentSpecificationProlog
- C\_CG::Attribute::MemorySegmentImplementationProlog
- C\_CG::Attribute::MemorySegmentSpecificationEpilog
- C\_CG::Attribute::SegmentedAttributeMacroName
- C\_CG::Attribute:MemorySegmentName

This property is valid only when the Segmented Memory capability is enabled (see CG::Configuration::EnableSegmentedMemory).

*Default = empty MultiLine*

Empty line is needed when updating Memory segment properties.

## MemorySegmentName

The MemorySegmentName property lets you define the name of the memory segment to be used. Use this property to define a memory segment using a stereotype. In addition, see the following properties:

- C\_CG::Attribute::MemorySegmentSpecificationProlog

- C\_CG::Attribute::MemorySegmentImplementationProlog
- C\_CG::Attribute::MemorySegmentSpecificationEpilog
- C\_CG::Attribute::MemorySegmentImplementationEpilog
- C\_CG::Attribute::SegmentedAttributeMacroName

This property is valid only when the Segmented Memory capability is enabled (see CG::Configuration::EnableSegmentedMemory).

*Default = empty string*

## MemorySegmentSpecificationEpilog

The MemorySegmentSpecificationEpilog property lets you add code as verbatim text (to be ignored by Rational Rhapsody) to the beginning of the forward data declaration related to a Memory Segment. Use this property to define a memory segment using a stereotype. In addition, see the following properties:

- C\_CG::Attribute::MemorySegmentSpecificationProlog
- C\_CG::Attribute::MemorySegmentImplementationProlog
- C\_CG::Attribute::MemorySegmentImplementationEpilog
- C\_CG::Attribute::SegmentedAttributeMacroName
- C\_CG::Attribute::MemorySegmentName

This property is valid only when the Segmented Memory capability is enabled (see CG::Configuration::EnableSegmentedMemory).

*Default = empty MultiLine*

## MemorySegmentSpecificationProlog

The MemorySegmentSpecificationProlog property lets you add any code as verbatim text (to be ignored by Rational Rhapsody) to the beginning of the forward data declaration related to a Memory Segment. Use this property to define a memory segment using a stereotype. In addition, see the following properties:

- C\_CG::Attribute::MemorySegmentSpecificationEpilog
- C\_CG::Attribute::MemorySegmentImplementationProlog
- C\_CG::Attribute::MemorySegmentImplementationEpilog
- C\_CG::Attribute::SegmentedAttributeMacroName
- C\_CG::Attribute::MemorySegmentName

This property is valid only when the Segmented Memory capability is enabled (see CG::Configuration::EnableSegmentedMemory).

*Default = empty MultiLine*

## MutatorGenerate

The MutatorGenerate property specifies whether to generate mutators for attributes. The possible values are:

- Smart - Mutators are not generated for attributes that have the Constant modifier.
- Always - Mutators are generated, regardless of the modifier.
- Never - Mutators are not generated.

*Default = Never*

## MutatorVisibility

The MutatorVisibility property specifies the access level of the generated mutator for attributes. This property defines the access level of a mutator operation regardless of the visibility of the attribute. The possible values are as follows:

- fromAttribute - Use the access level for the attribute for the mutator.
- public - Set the access level to public for the mutator.
- private - Set the access level to private for the mutator.
- protected - Set the access level to protected for the mutator. This value is not available in Rational Rhapsody Developer for C.
- default - Set the access level to default for the mutator. This value is available only in Rational Rhapsody Developer for Java.

*Default = fromAttribute*

## PreDeclarationModifier

The PreDeclarationModifier property is used to enable Rational Rhapsody to reverse engineer non-standard keywords that appear in attribute declarations. Keywords that appear before the attribute type are stored as the value of this property, and the property is then used during code generation to re-create the original code.

Since this is a code generation property, it can also be used to add non-standard keywords to code even when reverse engineering is not used.

This property is used in conjunction with the DeclarationModifier and PostDeclarationModifier properties.

*Default = Blank*

## PostDeclarationModifier

The PostDeclarationModifier property is used to enable Rational Rhapsody to reverse engineer non-standard keywords that appear in attribute declarations. Keywords that appear after the attribute name are stored as the value of this property, and the property is then used during code generation to re-create the original code.

Since this is a code generation property, it can also be used to add non-standard keywords to code even

when reverse engineering is not used.

This property is used in conjunction with the `PreDeclarationModifier` and `DeclarationModifier` properties.

*Default = Blank*

## RefactorRenameRegularExpression

When you use the `Refactor:Rename` feature, Rhapsody searches the user-defined code in the model for all occurrences of the name of the specified element, including places where the element name appears as part of the name of elements whose name is derived from the element that is being renamed.

To locate these elements, Rhapsody uses a set of regular expressions.

The regular expressions used for this search are defined by the property `RefactorRenameRegularExpression`.

Different values are used for this property, depending on what type of element you are renaming. For example, you will see that this property has a different value under `Class` than it does under `Attribute`. For some types of elements, Rhapsody uses a more generic expression, taken from the value of the property under `ModelElement`.

In general, the value of this property is a regular expression that consists of a number of "Or" components. For the renaming to work properly, each such component must contain three groupings, with the middle grouping using the variable `$keyword`, which represents the name of the element that is being renamed. In some cases, the expression takes this information from the value of another property rather than using the `$keyword` variable.

```
Default = ([^A-Za-z0-9_]|^)($keyword)([^A-Za-z0-9_]|$)
|([A-Za-z0-9_]<CG::Attribute::Mutator>[^A-Za-z0-9_]|$)
|([A-Za-z0-9_]<CG::Attribute::Accessor>[^A-Za-z0-9_]|$)
```

## ReferenceImplementationPattern

The `ReferenceImplementationPattern` property specifies how the `Reference` option for attribute/typedefs (composite types) is mapped to code.

*Default = \**

## Renames

The `Renames` property enables one element to rename another element of the same type. You can also rename an element by using a renames dependency. In the case of a conflict, the dependency has precedence. Note the following:

- For attributes, this property works only for static attributes in a class or for attributes in a package.
- For operations, this property contains the name of the operation this operation in renaming. The signatures of the two operations must match.

*Default = empty string*

## SegmentedAttributeMacroName

The SegmentedAttributeMacroName property specifies the method for generating an access macro to segmented data for a class. When this property is enabled, class attributes might be accessed by way of an access macro, eliminating the need to specify the segment name that contains the attribute.

*Default = \$class\_\$attribute*

## Simplify

If you are using the Rational Rhapsody customizable code generation mechanism, the Simplify property can be used to change the way specific types of elements are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - The element is ignored.
- Copy - The element is copied from the original to the simplified model. It is not modified in any way.
- Default - Uses the standard simplification for this item, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user.
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for the element has been applied.

*Default = "Default"*

## SpecificationEpilog

The SpecificationEpilog property allows you to add code to the end of the declaration of a model element.

For example, you can use this property to add an `#ifdef` to indicate that an operation is available only when the code is compiled with `_DEBUG`, by setting the following properties:

- For SpecificationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For SpecificationEpilog, enter the value `#endif`
- For ImplementationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For ImplementationEpilog, enter the value `#endif`

*Default = Blank*

## SpecificationProlog

The SpecificationProlog property allows you to add code to the beginning of the declaration of a model element.

For example, you can use this property to add an `#ifdef` to indicate that an operation is available only when the code is compiled with `_DEBUG`, by setting the following properties:

- For `SpecificationProlog`, enter the value `#ifdef _DEBUG` and a new line.
- For `SpecificationEpilog`, enter the value `#endif`
- For `ImplementationProlog`, enter the value `#ifdef _DEBUG` and a new line.
- For `ImplementationEpilog`, enter the value `#endif`

*Default = Blank*

## VariableInitializationFile

The `VariableInitializationFile` property specifies how global constant variables are initialized. You can initialize these variables in the specification file. You can use these variables as compile-time constants that can be used to define array sizes, for example. Rational Rhapsody automatically identifies constant variables with `const`. By modifying this property, you can choose the initialization file directly. The possible values are as follows:

- `Default` - The variable is initialized in the specification file if the type declaration begins with `const`. Otherwise, the variable is initialized in the implementation file.
- `Implementation` - Initialize global constant variables in the implementation file.
- `Specification` - Initialize global constant variables in the specification file.

*Default = Default*

## VariablePrivateQualifier

The `VariablePrivateQualifier` property specifies the qualifier that is printed at the beginning of the declaration that is generated for a private variable. If you don't want any qualifier to be generated, you can set the value of the property to an empty string.

*Default = static*

## VariablePublicQualifier

The `VariablePublicQualifier` property specifies the qualifier that is printed at the beginning of the declaration that is generated for a public variable. If you don't want any qualifier to be generated, you can set the value of the property to an empty string.

*Default = Blank*

## Visibility

The `Visibility` property specifies the visibility of that kind of model element. Code generation maps the visibility specified for an element to the same visibility in the generated language. The `Visibility` setting has the following applicability:

- **Classes** - Applies only to nested classes, which are defined inside other classes.
- **Types** - Applies only to types that are defined inside classes. It does not apply to global types, which are defined in packages.

*Default = Protected*

## AutosarRTE3x

A list of file names, with full paths, separated with commas. The generator copies this list of files to the folder of the makefile only if the file does not already exist.

## AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = "\$(UNQUOTEDOMROOT)/LangC/mxf/Adaptors/AutosarRTE/AR3x\_BMT"*

## AdditionalReservedWords

The AdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = \_\_asm \_\_finally naked \_\_based \_\_inline \_\_single\_inheritance \_\_cdecl \_\_int8 \_\_stdcall  
\_\_declspec \_\_int16 \_\_declspec \_\_int32 \_\_try \_\_dllimport \_\_int64 \_\_virtual\_inheritance \_\_except \_\_leave  
\_\_fastcall \_\_multiple\_inheritance*

## AlternativeOMROOTForMakefile

The generated makefile uses the OMROOT path to build the application. The value of the OMROOT variable is defined in the Rhapsody.ini file. Use the AlternativeOMROOTForMakefile property to specify an alternative location to be used instead of the OMROOT path.

*Default = Empty string*

## AssertMacroName

MicroC Execution Framework (MXF) has calls to the assert macro embedded in its code. Use the AssertMacroName property to specify the name of this macro in its environment. See also the UseAssertMacro property.

*Default = assert*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration. To change this property, use the Configuration window in the browser - do not change it by using the Properties window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link.

The possible values are:

- Debug - Generate the debug command that is set in the makefile.
- Release - Generate the release command that is set in the makefile.

*Default = Debug*

## C\_EXT

The C\_EXT property defines a token to be used in the definition for the C.CG:<environment>:MakeFileContent property. The C\_EXT token might appear in the MakeFileContent property and then Rational Rhapsody replaces it with the value set in the C\_EXT property.

*Default = .c*

## CompileSwitches

The CompileSwitches property specifies the compiler switches.

*Default = /I . /I \$OMDefaultSpecificationDirectory /I "\$(UNQUOTEDOMROOT)\LangC" /I "\$(UNQUOTEDOMROOT)\LangC\mx" /nologo /W3 /GX \$OMCPPCompileCommandSet /D "\_AFXDLL" /D "WIN32" /D "\_CONSOLE" /D "\_MBCS" /D "\_WINDOWS" \$(INST\_FLAGS)*



*\$(INCLUDE\_PATH) \$(INST\_INCLUDES) /c*

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = \$(CREATE\_OBJ\_DIR) \$(CC) \$OMFileCPPCompileSwitches /Fo"\$OMFileObjPath"  
"\$OMFileImpPath"*

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*Default = /Zi*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*Default = Empty string*

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: `$OMFileObjPath : $OMFileImpPath "*.bmp" "*.ico" "*.rc2"`

*Default = \$OMFileObjPath : \$OMFileImpPath \$OMFileSpecPath \$OMFileDependencies  
\$(FRAMEWORK\_CFG\_H)*

## EntryPoint

The EntryPoint property specifies the name of the main program for a given environment.

*Default = main*

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. The ErrorMessageTokensFormat property defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. The ErrorMessageTokens property has three parameters, each with an integer value:

- TotalNumberOfTokens - The number of tokens in the regular expression
- FileTokenPosition - The position of the file name token in the expression
- LineTokenPosition - The position of the line number token in the expression

*Default = TotalNumberOfTokens=3,FileTokenPosition=1,LineTokenPosition=2*

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the C\_CG::<Environment>::ExeName property plus the value of the ExeExtension property.

*Default = .exe*

## ExtraFilesToCopy

A list of file names (with full paths) separated with commas. The generator copies this list of files to the folder of the makefile only if the file does not already exist.

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements. The file inclusions are generated in the makefile.

*Default = \$OMSpecIncludeInElements \$OMImpIncludeInElements*

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile.

Note: If you select Release Build Set, these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

*Default =*  
*\$(UNQUOTEDOMROOT)\LangC\lib\\$(LIB\_PREFIX)WebComponents\$(LIB\_POSTFIX)\$(LIB\_EXT),*  
*\$(UNQUOTEDOMROOT)\lib\\$(LIB\_PREFIX)WebServices\$(LIB\_POSTFIX)\$(LIB\_EXT),*  
*ws2\_32\$(LIB\_EXT)*

## H\_EXT

The H\_EXT property defines a token to be used in the definition for the C\_CG:<environment>:MakeFileContent property. The H\_EXT token might appear in the MakeFileContent property and then Rational Rhapsody replaces it with the value set in the H\_EXT property.

*Default = .h*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment.

*Default = .c*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*Default = !INCLUDE*

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*Default = "\$executable"*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, include the name of the property preceded by \$. As shown in the following example (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT/etc/vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT/etc/msmake.bat\" \$makefile \$maketarget"*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .lib*

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

*Default = Empty string*

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

*Default = Empty string*

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

*Default = \$OMLinkCommandSet /NOLOGO*

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags

- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration. For example, the default content of the target type section of a C++ makefile for the Microsoft environment is as follows: ##### Target type (Debug/Release)

```
#####
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease BuildSet=$OMBuildSet
SUBSYSTEM=$OMSubSystem COM=$OMCOM RPFrameWorkDll=$OMRPFrameWorkDll
ConfigurationCPPCompileSwitches= $OMReusableStatechartSwitches
$OMConfigurationCPPCompileSwitches !IF "$ (RPFrameWorkDll)" == "True"
ConfigurationCPPCompileSwitches= $(ConfigurationCPPCompileSwitches) /D "FRAMEWORK_DLL"
!ENDIF !IF "$ (COM)" == "True" SUBSYSTEM=/SUBSYSTEM:windows !ENDIF
```

**Compilation Flags** The compilation flags section of the makefile contains the default compilation flags stored in the CompileSwitches property. For example, the default content of the compilation flags section of a C++ makefile for the Microsoft environment is as follows: ##### Compilation flags

```
#####
INCLUDE_QUALIFIER=/I LIB_PREFIX=MS
```

**Commands Definitions** The commands definition section of the makefile specifies programs to execute from the makefile. For example, the default commands definition section of a C++ makefile for the Microsoft environment is as follows: ##### Commands definition #####

```
##### RMDIR = rmdir LIB_CMD=link.exe -lib
LINK_CMD=link.exe LIB_FLAGS=$OMConfigurationLinkSwitches
LINK_FLAGS=$OMConfigurationLinkSwitches $(SUBSYSTEM) / MACHINE:I386
```

**Generated Macros** The generated macros section of the makefile contains a variable that expands to the Rational Rhapsody-generated macros in the makefile. For example: ##### Generated macros

```
##### $OMContextMacros
OBJ_DIR=$OMObjectsDir !IF "$ (OBJ_DIR)" != "" CREATE_OBJ_DIR=if not exist $(OBJ_DIR) mkdir
$(OBJ_DIR) CLEAN_OBJ_DIR= if exist $(OBJ_DIR) $(RMDIR) $(OBJ_DIR) !ELSE
CREATE_OBJ_DIR= CLEAN_OBJ_DIR= !ENDIF
```

The \$OMContextMacros keyword expands several macros in the makefile. Each makefile macro has its own keyword. You can use these keywords separately to customize the makefile. The \$OMContextMacros variable modifies target-specific variables. Replace the \$OMContextMacros line in the MakeFileContent property with the following: FLAGSFILE=\$OMFlagsFile

```
RULESFILE=$OMRulesFile OMROOT=$OMROOT C_EXT=$OMImplExt H_EXT=$OMSpecExt
OBJ_EXT=$OMObjExt EXE_EXT=$OMExeExt LIB_EXT=$OMLibExt
INSTRUMENTATION=$OMInstrumentation TIME_MODEL=$OMTimeModel
TARGET_TYPE=$OMTargetType TARGET_NAME=$OMTargetName $OMAllDependencyRule
TARGET_MAIN=$OMTargetMain LIBS=$OMLibs INCLUDE_PATH=$OMIncludePath
ADDITIONAL_OBJS=$OMAdditionalObjs OBJS= $OMObjs
```

**Predefined Macros** The predefined macros section of the makefile contains other macros than the Rational

Rhapsody-generated macros specified in the generated macros section. For example, the default predefined macros section of a C++ makefile for the Microsoft environment is as follows:

```
##### Predefined macros #####
##### $(OBJS) : $(INST_LIBS)
$(OXF_LIBS) LIB_POSTFIX= !IF "$$(BuildSet)"=="Release" LIB_POSTFIX=R !ENDIF !IF
"$(TARGET_TYPE)" == "Executable" LinkDebug=$(LinkDebug) /DEBUG
LinkRelease=$(LinkRelease) /OPT:NOREF !ELSEIF "$(TARGET_TYPE)" == "Library"
LinkDebug=$(LinkDebug) /DEBUGTYPE:CV !ENDIF !IF "$(INSTRUMENTATION)" == "Animation"
INST_FLAGS=/D "OMANIMATOR" INST_INCLUDES=/I $(OMROOT)\LangCpp\om /I
$(OMROOT)\LangCpp\om !IF "$(RPFrameWorkDll)" == "True" INST_LIBS=
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxanimdll$(LIB_POSTFIX) $(LIB_EXT)
!ELSE INST_LIBS= $(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomanim$(LIB_POSTFIX) $(LIB_EXT)
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfirst$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)" == "Tracing" INST_FLAGS=/D
"OMTRACER" INST_INCLUDES=/I $(OMROOT)\LangCpp\om /I $(OMROOT)\LangCpp\om !IF
"$(RPFrameWorkDll)" == "True" INST_LIBS= OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxtracedll$(LIB_POSTFIX) $(LIB_EXT) !ELSE
INST_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)tomtrace$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomtrace$(LIB_POSTFIX) $(LIB_EXT) OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfirst$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)" == "None" INST_FLAGS=
INST_INCLUDES= INST_LIBS= !IF "$(RPFrameWorkDll)" == "True"
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfdll$(LIB_POSTFIX) $(LIB_EXT) !ELSE
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB= !ELSE !ERROR An invalid Instrumentation $(INSTRUMENTATION) is specified.
!ENDIF
```

Generated Dependencies The generated dependencies section of the makefile contains a variable that expands to Rational Rhapsody-generated dependencies and compilation instructions. For example, the generated dependencies section of a C++ makefile for the Microsoft environment is as follows:

```
##### Generated dependencies #####
##### SOMContextDependencies
$OMFileObjPath : $OMMainImplementationFile $(OBJS) $(CPP) $(ConfigurationCPPCompileSwitches)
/Fo"$OMFileObjPath" $OMMainImplementationFile
```

Linking Instructions The linking instructions section of the makefile contains the predefined linking instructions. For example, the default linking instructions section of a C++ makefile for the Microsoft environment is as follows:

```
##### Linking instructions #####
#####
$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS) $OMFileObjPath
$OMMakefileName $OMModelLibs @echo Linking $(TARGET_NAME)$(EXE_EXT) $(LINK_CMD)
$OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \ $(OXF_LIBS) \
$(SOCK_LIB) \ $(LINK_FLAGS) /out:$(TARGET_NAME)$(EXE_EXT)
$(TARGET_NAME)$(LIB_EXT) : $(OBJS) $(ADDITIONAL_OBJS) $OMMakefileName @echo
Building library $@ $(LIB_CMD) $(LIB_FLAGS) /out:$(TARGET_NAME)$(LIB_EXT) $(OBJS)
$(ADDITIONAL_OBJS) clean: @echo Cleanup $OMCleanOBJS if exist $OMFileObjPath erase
$OMFileObjPath if exist *$(OBJ_EXT) erase *$(OBJ_EXT) if exist $(TARGET_NAME).pdb erase
$(TARGET_NAME).pdb if exist $(TARGET_NAME)$(LIB_EXT) erase
$(TARGET_NAME)$(LIB_EXT) if exist $(TARGET_NAME).ilk erase $(TARGET_NAME).ilk if exist
$(TARGET_NAME)$(EXE_EXT) erase $(TARGET_NAME)$(EXE_EXT) $(CLEAN_OBJ_DIR)
```

*Default =*



```

##### #
Description: The environment makefile for building the AutosarRTE3x software ## Notes: 1. make-utility
# Microsoft 'nmake' # 2. Input or internal variables: ## cfg release/debug option # release Release
(non-debug) version will be produced # debug Debug version will be produced ## 3. Targets: ##
<default> build application # clean remove produced files ## 4. Used environment variables ##
INCLUDE should point to the MSVC headers # LIB should point to the MSVC libraries #
##### #
Environment !IF "$(OS)" == "Windows_NT" NULL= RM=del /q !ELSE NULL=nul !ENDIF OUTDIR=
#----- $OMContextMacros
#-----
FRAMEWORK_CFG_H=$mxfCfgTemplateName.h COMPILER_OUTPUT_FLAG=/Fo
QUOTE_CHAR=" !IF "$(INSTRUMENTATION_MODE)" == "TargetMonitoring"
TARGET_MON_FLAGS=/D "_OM_TARGET_MON" !IF "$(TARGET_MON_COMM_FLAG)" != ""
TARGET_MON_FLAGS=$(TARGET_MON_FLAGS) /D $(TARGET_MON_COMM_FLAG) !ENDIF
TARGET_MON_OBJ = TargetMonitor.obj TARGET_MON_PROTOCOL = TargetMonitorProtocol.obj
!ENDIF OXF_DIR = $(UNQUOTEDOMROOT)$(PathDelimiter)LangC$(PathDelimiter)mxf #
Release/Debug cfg = $OMBuildSet # Type of the target MCU/CPU mcu = NT # Compiler cc = cl ld =
link # System Generation utility # Suffixes .SUFFIXES: .oil .s # Options INCLUDE_QUALIFIER=/I ccopt
= /I . /I . /I "$(UNQUOTEDOMROOT)$(PathDelimiter)LangC" /I "$(UNQUOTEDOMROOT)" /I
"$(UNQUOTEDOMROOT)$(PathDelimiter)LangC$(PathDelimiter)mxf" /nologo /W3
$(TARGET_MON_FLAGS) /GX $(CPPCompileDebug) /D "WIN32" /D "_CONSOLE" /D "_MBCS" /D
"_WINDOWS" $(INCLUDE_PATH) /c ldopt = /NOLOGO /SUBSYSTEM:console /MACHINE:I386
/STACK:0x100000,0x8000 /PDB:"$*.pdb" /OUT:"$@" /MAP:"$*.map" !IF "$(cfg)" == "Release" ccopt
= $(ccopt) $(OMCPPCompileRelease) ldopt = $(ldopt) /INCREMENTAL:no !ELSEIF "$(cfg)" ==
"Debug" ccopt = $(ccopt) $(OMCPPCompileDebug) ldopt = $(ldopt) /INCREMENTAL:no /DEBUG
!ENDIF # $(cfg) ##### # OXF part:
LIB_EXT=$(LibExtension) PDB_EXT=.pdb CFLAGS=$(ccopt) /Fo"$(MAKEDIR)"/ $(Include)
"$(OXF_DIR)/mxfFiles.list" SRCS = "$(OXF_DIR)/RiCOSAutosarRte.c" \ "$(OXF_DIR)/os(C_EXT)" \
$(OXF_SCR) HDRS = "$(OXF_DIR)/RiCOSAutosarRte.h" \ $(OXF_INC) OXF_LIB =
RiCOSAutosarRte$(ObjExtension) \ $(OXF_OBJ) OXF_LIB_CURR = RiCOSAutosarRte.obj \
$(OXF_OBJ) RiCOSAutosarRte.obj: $(FRAMEWORK_CFG_H) $(cc) $(ccopt) /Fo./
"$(OXF_DIR)/RiCOSAutosarRte.c" TargetMonitor.obj :
"$(UNQUOTEDOMROOT)/LangC/aom/TargetMonitor.c" $(TARGET_NAME).mak $(CC) $(ccopt)
$(ConfigurationCPPCompileSwitches) /Fo"TargetMonitor.obj"
"$(UNQUOTEDOMROOT)/LangC/aom/TargetMonitor.c"
##### OBJ_EXT=$(ObjExtension) # If using these
variables - put at the end of the path '\'. # Example: d:\sources\my_sources\ GLOBSRCPRE=
MODSRCPRE= MODOBJPRE= # Default target default: $(TARGET_NAME)$(ExeExtension) @echo
***** Target '$@' done ***** # Remove produced files clean: @if exist $(MODOBJPRE)*$(OBJ_EXT)
del $(MODOBJPRE)*$(OBJ_EXT) @if exist $(MODOBJPRE)*.pdb del $(MODOBJPRE)*.pdb @if exist
$(OXF_DIR)$(PathDelimiter)*$(OBJ_EXT) del $(OXF_DIR)$(PathDelimiter)*$(OBJ_EXT) @if exist
$(OXF_DIR)$(PathDelimiter)*.pdb del $(OXF_DIR)$(PathDelimiter)*.pdb @if exist *.vcp del *.vcp @if
exist *$(ExeExtension) del *$(ExeExtension) @if exist *.map del *.map @if exist *.mdp del *.mdp @echo
***** Target '$@' done ***** # Next line will be replaced with definition of:globalfilessrc, globalfilesobj
globalfilessrc= globalfilesobj= # Next line will be replaced with definition of:modulefilessrc,
modulefilesobj modulefilesobj=$(OBS) additionalSources = additionalSourcesObjects =
additionalObjects = userobj = \ $(modulefilesobj) \ $(globalfilesobj) \ $(additionalSourcesObjects) \
$(additionalObjects) $(TARGET_NAME)$(ExeExtension): $(userobj) $(OXF_LIB)
$(TARGET_MON_OBJ) $(TARGET_NAME).mak $(ld) $(ldopt) $(userobj) $(TARGET_MON_OBJ)
$(OXF_LIB_CURR) # Compiling C-files {$(MODOBJPRE)}$(OBJ_EXT):
{$(MODSRCPRE)}$(OBJ_EXT) {$(MODSRCPRE)}$(C_EXT)$(OBJ_EXT): $(cc) $(ccopt)
/Fo$(MODOBJPRE) $< !IF "$(additionalSourcesObjects)" != "" $(additionalSourcesObjects): $(cc)
$(ccopt) $(additionalSources) !ENDIF

```

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## ObjCleanCommand

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default = if exist \$OMFileObjPath erase \$OMFileObjPath*

## ObjectName

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = Empty string*

## ObjectsDirectory

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = Empty string*

## ObjExtension

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*Default = .obj*

## OSFileSystemCaseSensitive

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Cleared*

## ParseErrorMessage



The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning)

Along with the ErrorMessageTokensFormat property, the ParseErrorMessage property specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = ([^()]+)([()([0-9]+)]\*)[:] (error|warning|fatal error)*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation.

*Default = \*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Checked*

## ReactiveVtblKind

A reactive element uses a virtual table to drive its execution. Each execution framework has a different set of fields in this virtual table structure. Use the ReactiveVtblKind property to tell the tool which kind of virtual table you are using in order to initialize it correctly.

The possible values are:

- OXF - The version of the virtual table supported by C OXF.
- IDF - The version of the virtual table supported by C IDF.
- OSEK - The version of the virtual table supported by MXF.

*Default = OSEK*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running the product, whereas the target is the machine running the application. To run remotely, the UseRemoteHost property must be set to Checked.

If the UseRemoteHost property is set to Checked and the RemoteHost property is blank, the current host name is used for the remote host. You can use this as a workaround if you have problems running animated applications on Windows 95.

You can leave the RemoteHost property blank if both the application and Rational Rhapsody are running on the same machine.

*Default = Empty string*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*Default = .h*

## UseAssertMacro

Use the UseAssertMacro property to enable or disable compilation of code related to the assert macro. See also the AssertMacroName property.

*Default = Cleared*

## UseRemoteHost

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Cleared*

# CallOperation

The CallOperation metaclass contains properties relating to code generation for call operation elements in a model.

## DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- `$Name` - the element name
- `$FullName` - the full path of the element (P1::P2::C.a)
- `$Description` - the text provided on the Description tab of the Features window for the element
- `$Requirements` - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the `$Requirements` keyword in the `DescriptionTemplate` property to get such comments generated for specific model elements.
- `$Login` - the login name of the user who generated the file
- `$CodeGeneratedDate` - the date on which the code was generated
- `$CodeGeneratedTime` - the time at which the code was generated
- `$RhapsodyVersion` - the version of Rhapsody that was used to generate the file
- For operations: `$Signature` - the operation signature, `$Arguments` - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: `$Type` - the type of the attribute
- For arguments: `$Type` - the argument type, `$Direction` - the argument direction (In, Out, or InOut)
- For events: `$Arguments` - the text provided on the Description tab of the Features window for the event arguments
- For association ends: `$Target` - the other end of the association
- For requirements: `$ID` - the ID specified for the requirement, `$Specification` - the specification text for the requirement
- The values of tags can be included by using the syntax `$tag_name`. For example, if you use a tag called `author` for your classes, you can use the keyword `$author` in the value of `DescriptionTemplate` in order to have the value of the `author` tag included in the element description.
- The values of properties can be included by using the syntax `$property_name`. For example, you can use the keyword `$Animate` in the value of `DescriptionTemplate` in order to have the value of the `Animate` property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as `$Name`)
- Tag keywords
- Property keywords

If the value of a keyword is a MultiLine value, then each new line (except the first one) starts with the value of the `<lang>_CG::Configuration::DescriptionBeginLine` property, and each line ends with the value of the `<lang>_CG::Configuration::DescriptionEndLine` property.

*Default = State \$Name [[Description: \$Description]]*

## Class

The Class metaclass contains properties that affect the generated classes.

### ActiveBaseBeginMyTaskTemplate

The ActiveBaseBeginMyTaskTemplate property contains the template for the begin code section of the inlined doExecute code. See also the ActiveBaseEndMyTaskTemplate and ActiveBaseDoExecuteTemplate properties.

```
Default = ?<begin> $<IsActivePeriodic>?<==>True ?<||> $<IsForMainTask>?<==>True ?<||>  
$<IsActiveAsync>?<==>True?<?> if(?<begin>$<IsForMainTask>?<!=>True?<?>(me != NULL)  
&& ?<:>?<end>($taskMember != NULL) && (($taskMember)->myThread != NULL) ) { RiCTaskEM  
*myTaskMember = ($taskMember); ?<begin>$<IsActiveAsync>?<==>True?<?>?<begin>  
$<UseWhile>?<==>True?<?>while?<:>if?<end>($taskBase_hasPendingEvents(myTaskMember)){  
RiCOSPendingEvent_clear(&(myTaskMember->myThread->osTask)); ?<:>?<begin>  
$<IsActivePeriodic>?<==>True ?<||> $<IsForMainTask>?<==>True  
?<?>RiCOSPendingEvent_clear(&(myTaskMember->myThread->osTask));?<:>?<end>?<end>?<:>?<end>
```

### ActiveBaseDoExecuteTemplate

The ActiveBaseDoExecuteTemplate property contains the template for the code between "begin" and "end" sections of the inlined doExecute code. See also the ActiveBaseBeginMyTaskTemplate and ActiveBaseEndMyTaskTemplate properties.

```
Default = ?<begin>$<C.CG::Class::InlineActiveBaseDoExecute>?<!=>False  
?<?>?<begin>$<CG::Class::GenerateVTBLsInConstructorn>?<!=>False ?<&&>  
$<CG::Framework::ActiveVtblName>?<!=>?<&&>  
$<CG::Framework::ActiveExecuteOperationName>?<!=> ?<?> /*LDRA_INSPECTED 57 S : LDRA  
Analysis incorrect. Statement has side effect */  
(myTaskMember->myThread->vtbl->execute)(myTaskMember);?<:> { RiCEvent * ev = NULL;  
?<begin>$<useInstrumentation>?<!=>no?<&&>  
$<Animation::TargetMonitoring::UseTargetMonitoring>?<!=>On?<?> static RiC_INT8 first =  
RiCTRUE; /* force a cleanup of the call stack when the thread starts its event loop */ /* Ensure we do not  
report again after "dummy" or cancelled events */ RiCBoolean shouldNotify = RiCTRUE; /* Using  
function ARCEQ_isEmpty() for instrumentation */ ARCEventQueue * aomEQ =  
ARCT_getEventQueue(myTaskMember->aomTask); if(first) { first = RiCFALSE;  
ARCCS_popall(OMCurrentCallStack); /*lint !e746 () */ } ARCCS_notify(OMCurrentCallStack, NULL,  
omMethodMethod); while (!ARCEQ_isEmpty(aomEQ)){ ?<:> do { ?<end>  
?<begin>$<useInstrumentation>?<!=>no?<&&>  
$<Animation::TargetMonitoring::UseTargetMonitoring>?<!=>On?<?> if (shouldNotify){  
if(!$<C.CG::Framework::ActiveBase>IsEndOfProcess()) {  
ARCS_notifyEvent(RiCTaskEM_getStepper(myTaskMember)); shouldNotify = RiCFALSE; } }  
?<:>?<end> /* Actually dispatch the event */ ?<begin>$<useInstrumentation>?<!=>no?<&&>  
$<Animation::TargetMonitoring::UseTargetMonitoring>?<!=>On?<?> if(aomEQ != NULL){ ev =  
ARCEQ_get(aomEQ); } ?<:> { /* Access the event queue */  
RiCOSMutex_lock(&(myTaskMember->taskMonitor)); /* Justification: Violation of Misra98 Rule45 /  
Misra04 Rule11.4 */ /* Cast from/to void* pointer is required to implement general purpose data
```

```

collections *//*LDRA_INSPECTED 94 S *//*LDRA_INSPECTED 95 S *//*LDRA_INSPECTED 203 S
*/ ev = (RiCEvent *) RiCOSMessageQueue_get( &(myTaskMember->eventQueue));
if(RiCOSMessageQueue_isEmpty(&(myTaskMember->eventQueue)) != RiCFALSE) {
if(myTaskMember->myThread != NULL){ (RhpVoid)
RiCOSEventFlag_reset(&(myTaskMember->myThread->taskEventFlag));
RiCOSPendingEvent_reset(&(myTaskMember->myThread->osTask)); } }
RiCOSMutex_free(&(myTaskMember->taskMonitor)); } /* mutex is freed */?<end> if(ev != NULL) { if
((RiCEvent_getId(ev)) != RiCCancelledEvent_id) { ?<begin>$<useInstrumentation>?<!=>no?<&&>
$<Animation::TargetMonitoring::UseTargetMonitoring>?<!=>On?<?> shouldNotify = RiCTRUE;
?<:>?<end> */LDRA_INSPECTED 45 D : Pointer is checked for null within function */
(void)RiCReactive_takeEvent(RiCEvent_getDestination(ev), ev); } if
(RiCEvent_isDeleteAfterConsume(ev) != RiCFALSE) {?<begin>
$<useInstrumentation>?<!=>no?<&&>
$<Animation::TargetMonitoring::UseTargetMonitoring>?<!=>On?<?> if
(!RiCEvent_isFrameworkEvent(ev)) ARCE_going2DestryEvent(ev);?<:>?<end> if (ev->eventDestroyOp
!= NULL) { */LDRA_INSPECTED 19 D : LDRA Analysis incorrect. Function Pointer call has no
available function definition *//*LDRA_INSPECTED 41 D : LDRA Analysis incorrect. Function Pointer
call has no available function declaration *//*LDRA_INSPECTED 496 S */ ev->eventDestroyOp(ev); /*
the real event destroy op */ }
?<begin>$<useInstrumentation>?<!=>no?<||>$<CG::Framework::MicroCOxfCleanup>?<==>True?<?>
else { RiCEvent_destroy(ev); }?<:>?<end> } }?<begin>$<useInstrumentation>?<==>no?<||>
$<Animation::TargetMonitoring::UseTargetMonitoring>?<==>On?<?>while (ev !=
NULL);?<:>?<end> ?<begin>$<useInstrumentation>?<!=>no?<&&>
$<Animation::TargetMonitoring::UseTargetMonitoring>?<!=>On ?<?> /* done with the events in this
iteration so the call stack can be pop-ed entirly */ ARCCS_popall(OMCurrentCallStack); if (ev ==
NULL) { /* Entering idle state If should stop on idle -- then stop else become idle Note: if should stop on
idle you never realy go idle */
(void)$<C_CG::Framework::ActiveBase>_hasPendingEvents(myTaskMember); } ?<:>?<end>
}?<end>?<:>?<end>

```

## ActiveBaseEndMyTaskTemplate

The ActiveBaseEndMyTaskTemplate property contains the template for the end code section of the inlined doExecute code. See also the ActiveBaseBeginMyTaskTemplate and ActiveBaseDoExecuteTemplate properties.

```

Default = ?<begin> $<IsActivePeriodic>?<==>True ?<||> $<IsForMainTask>?<==>True ?<||>
$<IsActiveAsync>?<==>True?<?>?<begin>
$<IsActiveAsync>?<==>True?<?>?<begin>?$<UseWhile>?<==>True ?<?>
$taskBase_flagWait(myTaskMember); ?<:>?<end> } ?<:>?<begin>$<IsActivePeriodic>?<==>True
?<||> $<IsForMainTask>?<==>True ?<?>
RiCOSTask_endMyTask(&(myTaskMember->myThread->osTask)); ?<:>?<end>?<end>}?<:>?<end>

```

## AccessTypeName

The AccessTypeName property specifies the name of the access type generated for the class record.

Default = empty string

## ActiveMessageQueueSize

The `ActiveMessageQueueSize` property specifies the size of the message queue allocated for active classes. The possible values are as follows:

- A string - Specifies the message queue size for an active class.
- An empty string (blank) - The value is set in an operating system-specific manner.

*Default = empty string*

## **ActiveStackSize**

The `ActiveStackSize` property specifies the size of the stack allocated for active objects. The possible values are as follows:

- Any integer - Specifies that a stack of that size is allocated for active objects.
- An empty string (blank) - If not specified, the stack size is set in an operating system-specific manner, based on the value of the `ActiveStackSize` property for the framework.

*Default = empty string*

## **ActiveThreadName**

The `ActiveThreadName` property indicates the real OS task or thread name. This property only matters when the class is set to active. This facilitates debugging in complex environments in which many threads are constantly being created and deleted on-the-fly. This property is effective for all targets. All strings entered must be enclosed in quotes (" ").

*Default = NULL*

## **ActiveThreadPriority**

The `ActiveThreadPriority` property specifies the priority of active class threads. The possible values are as follows:

- A string - Specifies thread priority of an active class.
- An empty string (blank) - The value is set in an operating system-specific manner.

*Default = empty string*

## **AdditionalBaseClasses**

The `AdditionalBaseClasses` property adds inheritance from external classes to the model.

*Default = empty string*

## **AdditionalNumberOfInstances**

The `AdditionalNumberOfInstances` property is a string that specifies the size of the local heap allocated

for events when the current pool is full. Triggered operations use the event properties. This property provides support for static architectures found in hard real-time and safety-critical systems without memory management capabilities during run time. All events are dynamically allocated during initialization.

Once allocated, the event queue for a thread remains static in size. The possible values are:

- An empty string (blank) - No additional memory is to be allocated when the initial memory pool is exhausted.
- *n* (a positive integer) - Specifies the size of the array allocated for additional instances.

*Default = empty string*

## AllocateMemory

The AllocateMemory property specifies the string generated to allocate memory dynamically for objects or events. This string is used in the Create() operation. The default memory allocation string is as follows: (\$cname \*) malloc(sizeof(\$cname)); The variable \$cname is replaced with the name of the object type during code generation. For example, the Create() operation generated for an object A uses this string to allocate memory for a new object as follows: A \* A\_Create(RiCTask \* p\_task) { A\* me = (A \*) malloc(sizeof(A)); A\_Init(me, p\_task); return me; } You can edit the memory allocation string to use a different mechanism than malloc(). The string used to free memory is specified with the FreeMemory property.

## Animate

The Animate property specifies whether animation code is generated for an element. You can specify your own animation function by using the CG::Attribute::AnimSerializeOperation property. The semantics of the Animate property is always in favor of the owner settings:

- If a package Animate property is set to False, all the classes owned by the package are not animated, regardless of the class Animate settings.
- If a class Animate property is set to False, all the elements in the class (attributes, operations, relations, and so on) are not animated.
- If an operation Animate property is set to False, all the arguments are not animated.
- If the AnimateArguments property is set to False, all the arguments are not animated, regardless of the specific argument Animate property settings.

*Default = Checked*

## AnimSerializeOperation

The animation feature is capable of animating attributes that are of primitive types such as integers or that are one-dimensional arrays of such types. If you want the animation to also include attributes that are based on types or classes you have defined, you must write serialization and unserialization functions for handling these types and classes.

The AnimSerializeOperation property is used to tell Rational Rhapsody the name of the function that should be used for serialization.



The value of the property can be set at the Attribute, Type, or Class level.

If you set the value at the Attribute level, the function specified are only used for that specific attribute.

If you did not set the value at the Attribute level, Rational Rhapsody checks whether the attribute is based on a type or on a class. If the attribute is based on a type, then Rational Rhapsody takes the value of the property defined at the Type level. If the attribute is based on a class, Rational Rhapsody takes the value of the property defined at the Class level.

The property value should consist only of the name of the function. You must make sure that your code contains the include statements that are necessary to find the serialization function.

*Default = Blank*

## **AnimUnserializeOperation**

The animation feature is capable of animating attributes that are of primitive types such as integers or that are one-dimensional arrays of such types. If you want the animation to also include attributes that are based on types or classes you have defined, you must write serialization and unserialization functions for handling these types and classes.

The AnimUnserializeOperation property is used to tell Rational Rhapsody the name of the function that should be used for unserialization.

The value of the property can be set at the Attribute, Type, or Class level.

If you set the value at the Attribute level, the function specified are only used for that specific attribute.

If you did not set the value at the Attribute level, Rational Rhapsody checks whether the attribute is based on a type or on a class. If the attribute is based on a type, then Rational Rhapsody takes the value of the property defined at the Type level. If the attribute is based on a class, Rational Rhapsody takes the value of the property defined at the Class level.

The property value should consist only of the name of the function. You must make sure that your code contains the include statements that are necessary to find the unserialization function.

*Default = Blank*

## **AnimUseMultipleSerializationFunctions**

The AnimSerializeOperation and AnimUnserializeOperation properties are used to specify user-provided functions for serialization/unserialization of objects to allow inclusion of such objects in animation.

Because Rational Rhapsody allows you to fine-tune code generation of arguments by using the In, Out, InOut, and TriggerArgument properties, you might need to provide multiple serialization/unserialization functions to handle these different types of arguments. You can use the AnimUseMultipleSerializationFunctions property to instruct Rational Rhapsody to use multiple user-provided serialization/unserialization functions.

If you set the value of this property to Checked, Rational Rhapsody searches for user-provided



serialization functions whose names consist of the string entered for the AnimSerializeOperation property and the suffixes "In", "Out", "InOut", and "TriggerArgument".

The same is true for unserialization functions. However, for unserialization, Rational Rhapsody cannot handle Out arguments, so the relevant suffixes are "In", "InOut", and "TriggerArgument".

Since the AnimSerializeOperation and AnimUnserializeOperation properties exist under both the Type metaclass and the Class metaclass, the AnimUseMultipleSerializationFunctions property also exists under both these metaclasses.

*Default = Cleared*

## BaseNumberOfInstances

The BaseNumberOfInstances property is a string that specifies the size of the local heap memory pool allocated for either:

- Instances of the class (C\_CG::Class)
- Instances of the event (C\_CG::Event)
- This property provides support for static architectures found in hard real-time and safety-critical systems without memory management capabilities during run time. All instances of events are dynamically allocated during initialization. Once allocated, the event queue for a thread remains static in size.

Triggered operations use the properties defined for events. When the memory pool is exhausted, an additional amount, specified by the AdditionalNumberOfInstances property, is allocated. Memory pools for classes can be used only with the Flat statechart implementation scheme. The possible values are as follows:

- An empty string (blank) - Memory is always dynamically allocated.
- n (positive integer) - An array is allocated in this size for instances.

The related properties are as follows:

- AdditionalNumberOfInstances - Specifies the number of instances to allocate if the pool runs out.
- ProtectStaticMemoryPool - Specifies whether the pool should be protected (to support a multithreaded environment)
- EmptyMemoryPoolCallback - Specifies a user callback function to be called when the pool is empty. This property should be used instead of the AdditionalNumberOfInstance property for error handling.
- EmptyMemoryPoolMessage - When set to true, this property causes a message to be displayed if the pool runs out of memory in instrumented mode.

*Default = empty string*

## ComplexityForInlining

The ComplexityForInlining property specifies the upper bound for the number of lines in user code that are allowed to be inlined. User code is the action part of transitions in statecharts. For example, when you use the value 3, all transitions with actions consisting of three lines or fewer of code are automatically

inlined in the calling function. Inlining is replacing a function call in the generated code with the actual code statements that make up the body of the function.

This optimizes the code execution at the expense of an increase in code size. For example, increasing the number of lines that can be inlined from 3 to 5 has shortened the code execution time in some cases up to 10%. This property applies only to the Flat implementation scheme for statecharts.

*Default = 0*

## DeclarationModifier

The DeclarationModifier property adds a string to the class or event declaration. The string is displayed between the class keyword and the class name in the generated code. For example, for a class A, the DeclarationModifier would appear as follows: `class DeclarationModifier> A {...}`; This property adds a modifier to the class declaration. For example, if you have a class myExportableClass that is exported from a DLL by using the MYDLL\_API macro, you can set the DeclarationModifier property to "MYDLL\_API." The generated code would then be as follows: `class MYDLL_API myExportableClass {...}`; This property supports two keywords: \$component and \$class.

*Default = empty string*

## DefaultValue

In generated C code, there are cases where operations that belong to an interface can end up returning an uninitialized variable.

The DefaultValue property allows you to define a value, such as null, that can be returned in such situations.

Providing a value for this property also prevents problematic return values in situations where your application tries to call an operation that is part of a required interface for a port, but the service is not available.

*Default = Blank*

## DependenciesAutoArrange

The DependenciesAutoArrange property determines the criterion used to organize usage dependencies in the generated code. If the value of the property is set to True, the dependencies are ordered alphabetically in the code. If you set the value to False, the order of dependencies in the code is in accordance with the order that you specified in the "Edit usage dependencies order" window. This property corresponds to the "Use default order" check box in the "Edit usage dependencies order" window.

*Default = True*

## DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be

generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- `$Name` - the element name
- `$FullName` - the full path of the element (P1::P2::C.a)
- `$Description` - the text provided on the Description tab of the Features window for the element
- `$Requirements` - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the `$Requirements` keyword in the `DescriptionTemplate` property to get such comments generated for specific model elements.
- `$Login` - the login name of the user who generated the file
- `$CodeGeneratedDate` - the date on which the code was generated
- `$CodeGeneratedTime` - the time at which the code was generated
- `$RhapsodyVersion` - the version of Rhapsody that was used to generate the file
- For operations: `$Signature` - the operation signature, `$Arguments` - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: `$Type` - the type of the attribute
- For arguments: `$Type` - the argument type, `$Direction` - the argument direction (In, Out, or InOut)
- For events: `$Arguments` - the text provided on the Description tab of the Features window for the event arguments
- For association ends: `$Target` - the other end of the association
- For requirements: `$ID` - the ID specified for the requirement, `$Specification` - the specification text for the requirement
- The values of tags can be included by using the syntax `$tag_name`. For example, if you use a tag called `author` for your classes, you can use the keyword `$author` in the value of `DescriptionTemplate` in order to have the value of the `author` tag included in the element description.
- The values of properties can be included by using the syntax `$property_name`. For example, you can use the keyword `$Animate` in the value of `DescriptionTemplate` in order to have the value of the `Animate` property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as `$Name`)
- Tag keywords

- Property keywords

If the value of a keyword is a MultiLine value, then each new line (except the first one) starts with the value of the `<lang>_CG::Configuration::DescriptionBeginLine` property, and each line ends with the value of the `<lang>_CG::Configuration::DescriptionEndLine` property.

*Default = Blank*

## Destructor

The Destructor property controls the generation of virtual destructors in C++. The property exists for C for historical reasons, with a single value of `auto`, but it has no effect on the generated C code. The possible values are as follows:

- `auto` - A virtual destructor is generated for an object only if it has at least one virtual function.
- `virtual` - A virtual destructor is generated in all cases.
- `abstract` - A virtual destructor is generated as a pure virtual function.
- `common` - A nonvirtual destructor is generated.

*Default = Auto*

## Embeddable

The Embeddable property is a Boolean property that specifies whether a class can be allocated by value (nested) inside another class or package.

For example, if the Embeddable property is `Checked`, 20 instances of a class `A` can be allocated inside another class by using the following syntax: `A itsA[20]`; The possible values are as follows:

- `Checked` - The object can be allocated by value inside a composite object or package. The object declaration and definition are generated in the specification file of the composite.
- `Cleared` - The object cannot be embedded inside another object (not supported in RiC). The object declaration and definition are generated in the implementation file of the composite.

The Embeddable property is used with the `EmbeddedScalar` and `EmbeddedFixed` properties to determine how to generate code for an embedded object. The Embeddable property must be set to `True` for either of those properties to take effect. It is also closely related to the `ImplementWithStaticArray` property, which also needs to be set in order to support by-value allocation. Relations can be generated by value only under the following circumstances:

- The multiplicity of the relation is well-defined (not “\*”).
- The `ImplementWithStaticArray` property of the component relation is set to `FixedAndBounded`.

When the Embeddable property is `False` (RiC only):

- The attributes of the object are encapsulated. Clients of the object are forced to use it only by way of its operations, because there is no direct access to its attributes.
- Dynamic allocation must be used. The compiler does not know how to statically allocate an object when its declaration is not visible.

*Default = Checked*

## EnableDynamicAllocation

The EnableDynamicAllocation property specifies whether to use dynamic memory allocation for objects. The possible values are as follows:

- Checked - Dynamic allocation of events is enabled. Create() and Destroy() operations are generated for the object or object type.
- Cleared - Events are dynamically allocated during initialization, but not during run time. Create() and Destroy() operations are not generated for the object. This setting is recommended for static architectures that do not use dynamic memory management during run time.

*Default = Checked*

## EnableUseFromCPP

The EnableUseFromCPP property specifies whether to wrap generated C structs and functions with an appropriate extern C { } wrapper to prevent problems when code is compiled with a C++ compiler.

Wrapping C code with extern C includes C code in a C++ application.

For example, if EnableUseFromCPP is set to True, the following wrapper code is generated around the struct and functions that are generated:

```
#ifdef __cplusplus extern "C" { #endif /* __cplusplus */  
  
#ifdef __cplusplus } #endif /* __cplusplus */
```

*Default = False*

## ExecutableOperationName

The ExecutableOperationName property specifies the name of the doExecute operation for an Active/ExecutionManager/Executable class.

The doExecute operation is the operation that is called by the OS for Active classes, is the operation in which an ExecutionManager has the code that manages its manageable parts, and the operation that an Executable implements its functionality to be called by its ExecutionManager.

*Default = doExecute*

## ExecutionManagerDispatchEventOrder

The ExecutionManagerDispatchEventOrder property specifies the order in which an Execution Manager (a class defined on its General tab of its Features window) manages its content. The possible values are:

- AfterParts – The event dispatching is done after the managed parts are executed.

- BeforeParts – The event dispatching is done before the managed parts are executed.

In the Extended Execution Model, a call to the part means calling its doExecute operation. See the C.CG::Class::ExecutableOperationName property.

*Default = AfterParts*

## InlineActiveBaseDoExecute

The InlineActiveBaseDoExecute property controls whether to inline the doExecute code from the RiCTaskEM function into the generated application code.

*Default = Cleared*

## ExecutionManagerEventQueueSize

The ExecutionManagerEventQueueSize property specifies the size of the events queue related to an ExecutionManager. This is the maximum number of events that can be pending at a given time.

*Default = 10U*

## ExecutionManagerUseOSEventQueue

For models that use the SafetyCriticalForCDevelopers settings, the ExecutionManagerUseOSEventQueue property is used to fine-tune the generated code for the environments supported, such as VxWorks 653 and QNX 6.5.0. When the value of the property is set to True, the code for active classes does not include the event queue attribute ric\_eventQueue, which is not necessary for these environments because they use a native event queue implementation. For example:

```
struct Dishwasher_t {

    RiCTaskEM ric_task;

    RiCReactive ric_reactive;

    RiCEvent* ric_eventQueue[10U]; /* this line won't be generated if ExecutionManagerUseOSEventQueue
is set to True

    RiCThread ric_thread;

    ...

}
```

*Default = True*

## ExecutionMethod

The ExecutionMethod property specifies the execution method.

The possible values are:

- None
- Executable
- ExecutionManager

*Default = None*

## Final

The Final property, when set to Cleared, specifies that the generated record for the class is a tagged record. This property applies to Ada95.

*Default = Cleared*

## FreeMemory

The FreeMemory property specifies the string generated to free memory previously allocated for objects or events. This string is used in the Destroy() operation. For an object, the free memory string is as follows: free(\$meName); The variable meName is replaced with the string used for the me context variable during code generation. For example, the Destroy() operation generated for an object A uses this string to free memory when an instance of A is destroyed as follows: void A\_Destroy(A\* const me) { A\_Cleanup(me); free(me); } You can edit the string used to free memory to use a different mechanism than free(). The string used to allocate memory is specified with the AllocateMemory property.

*Default = free(\$meName);*

## GenerateAccessType

The GenerateAccessType property determines which access types are generated for the class. The possible values are as follows:

- None - Access types are not generated.
- Standard - An access type is generated.
- General - General access types are generated.

*Default = General*

## GenerateDestructor

The GenerateDestructor property specifies whether to generate a destructor for a class.

*Default = Checked*

## **GenerateRecordType**

The GenerateRecordType property determines whether the class record is generated.

*Default = Checked*

## **GenerateVTBLsInConstructor**

The GenerateVTBLsInConstructor property forces the code generation of the virtual table struct in the constructor function of the class it is related to.

The entities in Rational Rhapsody for C that have virtual table struct and are influenced by this property are:

- Reactive classes
- Active classes with the ActiveExecuteOperationName property set
- Classes with realization of interfaces

When the GenerateVTBLsInConstructor property does not exist (when the CGCompatibilityPre75C profile is not loaded and the user does not add it manually) the behavior is as if the value for the property is set to Checked.

*Default = Cleared*

## **HasUnknownDiscriminant**

The HasUnknownDiscriminant property determines whether an unknown discriminant >) is generated for this class.

*Default = Cleared*

## **ImplIncludes**

The ImplIncludes property specifies the names (including full paths) of header files to be included at the top of implementation files generated for classes, objects or object types, or packages. Separate multiple file names by using commas, without spaces.

*Default = empty string*

## **ImplementationEpilog**

The ImplementationEpilog property enables you to add any code that you want to be added as verbatim text (to be ignored by Rational Rhapsody) to the end of the definition of a model element.. For example, you could wrap a section of code with an #ifdef-#endif pair, add compiler-specific keywords, or add a



#pragma statement. For example, to specify that an operation is available only when the code is compiled with `_DEBUG`, set the following properties for the operation:

- Set SpecificationProlog to `#ifdef _DEBUG cr.`
- Set SpecificationEpilog to `#endif.`
- Set ImplementationProlog to `#ifdef _DEBUG cr.`
- Set ImplementationEpilog to `#endif.`

The following table lists whether leading and trailing linefeeds are generated.

Metaclass	Leading Linefeed Added?	Class	Yes	Package	No
-----------	-------------------------	-------	-----	---------	----

*Default = Empty MultiLine*

## ImplementationPragmas

The ImplementationPragmas property specifies the user-defined pragmas to generate in the body.

*Default = Empty MultiLine*

## ImplementationPragmasInContextClause

The ImplementationPragmasInContextClause property specifies the user-defined pragmas to generate in the context clause of the body.

*Default = Empty MultiLine*

## ImplementationProlog

The ImplementationProlog property enables you to add any code that you want to be added as verbatim text (to be ignored by the software) to the beginning of the definition of a model element. For example, you could wrap a section of code with an `#ifdef-#endif` pair, add compiler-specific keywords, or add a `#pragma` statement. For example, to specify that an operation is available only when the code is compiled with `_DEBUG`, set the following properties for the operation:

- Set SpecificationProlog to `#ifdef _DEBUG cr.`
- Set SpecificationEpilog to `#endif.`
- Set ImplementationProlog to `#ifdef _DEBUG cr.`
- Set ImplementationEpilog to `#endif.`

The following table lists whether leading and trailing linefeeds are generated.

Metaclass	Trailing Linefeed Added?	Class	No	Package	Yes
-----------	--------------------------	-------	----	---------	-----

*Default = Empty MultiLine*

## In

The In property specifies how code is generated when the type is used with an argument that has the modifier In. The C value "const \$type\*" is the default.

## InitCleanUpRelations

The InitCleanUpRelations property specifies whether to generate initRelations() and cleanUpRelations() operations for sets of related global instances. This property applies only to composites and global relations.

*Default = Checked*

## InitializationCode

The InitializationCode property adds the specified initialization code in the body of the class. A non-abstract class can have initialization code that is executed during elaboration of the associated package. (empty MultiLine)

## InOut

The InOut property specifies how code is generated when the type is used with an argument that has the modifier "InOut."

## InstanceDeclaration

The InstanceDeclaration property specifies how instances are declared in code.

*Default = struct \$cname\$suffix In the generated code, the variable \$cname is replaced with the object (or object type) name. The variable \$suffix is replaced with the type suffix "\_t," if the object is of implicit type.*

## IsCompletedOperation

The IsCompletedOperation specifies whether state\_IS\_COMPLETED operations are generated as functions or macros (by using #define). The possible values are as follows:

- Plain - state\_IS\_COMPLETED operations are generated as functions (pre-V4.2 behavior). This is the default value.
- Inline - state\_IS\_COMPLETED operations are generated by using #define macros, if the body contains only a return statement.

*Default = Plain*

## IsInOperation

When Rational Rhapsody generates code for a class with a statechart, the code includes, by default, an `_IN` function for each state defined (for example, `class_0_state_1_IN`) whose return value indicates whether or not the system is currently in that state.

For statecharts that contain one or more And states, the generated code includes calls to these `_IN` functions.

The `IsInOperation` property can be used to control whether or not these `_IN` functions should be generated, and what the code looks like when they are generated. The possible values for the property are:

- **Inline** - Rational Rhapsody generates the `_IN` code as macros
- **Plain** - Rational Rhapsody generates the `_IN` code as functions
- **Never** - `_IN` functions are not generated for the states in the statechart

Note that if you set this property to **Never**, and your statechart includes one or more And states, the generated code does not compile because it contains calls to non-existent functions.

*Default = Plain*

## IsLimited

The `IsLimited` property determines whether the class or record type is generated as limited.

*Default = Cleared*

## IsNested

The `IsNested` property specifies whether to generate the class or package as nested.

*Default = Cleared*

## IsPrivate

The `IsPrivate` property specifies whether to generate the class or package as private.

*Default = Cleared*

## IsReactiveInterface

The `IsReactiveInterface` property modifies the way reactive classes are generated. It has the following effects:

- Virtual inheritance from `OMReactive`
- Prevents instrumentation

- Prevents the thread argument and the initialization code (setting the active context) in the class constructor
- Creates a pure-virtual destructor (by default)

This property affects only classes that declare themselves as interfaces by having a stereotype with a name that contains the word “interface” (case-insensitive). In previous versions of Rational Rhapsody, a class could inherit from a single reactive class only, regardless of whether it was an interface or implementation class. Beginning with Version 4.0.1 MR2, a class can inherit (implement) several reactive interfaces.

In Rational Rhapsody Developer for C++, you must explicitly designate reactive interfaces because the code generator applies special translation rules involving multiple inheritance from the Rational Rhapsody framework. You can designate a reactive interface in two ways:

- Set the `C_CG::Class::IsReactiveInterface` property to true.
- Use the predefined stereotype `Reactive_interface`. This stereotype uses stereotype-based code generation in order to automatically apply the correct property value.

Alternatively, you can define another stereotype (such as `PortSpec`) that sets `IsReactiveInterface` to true and use that stereotype. A class is considered reactive if it meets all the following conditions:

- The `C_CG::Framework::ReactiveBase` property is not empty.
- The `C_CG::Framework::ReactiveBaseUsage` property is set to true.
- One or more of the following conditions are true:
  - The class has a statechart or activity diagram.
  - The class is a composite class.
  - The class has event receptions or triggered operations.

*Default = Checked*

## MarkPrologEpilogInAnnotations

The `MarkPrologEpilogInAnnotations` property specifies whether to generate ignore annotations for the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties so they are ignored during roundtrip. When you insert code element declarations (variables, types, functions, and so on) in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, after a full roundtrip those elements are added to the model and are duplicated on the next code generation.

Use you use the `MarkPrologEpilogInAnnotations` property, you can have Rational Rhapsody automatically ignore the information specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties instead of adding the ignore annotations manually. The possible values for the `MarkPrologEpilogInAnnotations` property are:

- None - Rational Rhapsody does not generate any annotations. Any models created before Version 4.1 automatically have this property setting.
- Ignore - Rational Rhapsody generates the `///  
[ ignore annotation before the code specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, and generates the ///  
] annotation after the code specified in those properties.`
- Auto - If the code in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties is one line (it does not contain any newline characters (`\n`)), no

annotations are generated (the same behavior has the None setting). If there is more than one line, Rational Rhapsody generates the `///  
[ ignore` annotation before the code specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, and generates the `///  
]` annotation after the code specified in those properties (the same behavior as the Ignore setting).

During roundtrip, any ignore annotations in the comments of the element are not included in its description. Because the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties are generated between the annotation and its declaration for the element, you cannot rename those elements on roundtrip. If you change the name of an element, it is removed from the model and added with the new name. Some model information (for example, property settings) might be lost.

*Default = Auto*

## MaximumPendingEvents

The MaximumPendingEvents property specifies the maximum number of events that can be simultaneously pending in the event queue of the active class. The possible values are as follows:

- -1 - Memory is dynamically allocated.
- Positive integer - Specifies the maximum number of events.

*Default = -1*

## NestingVisibility

The NestingVisibility property specifies the visibility of the generated specification of the nested class or package.

*Default = Public*

## NetworkPortGetAPI

The NetworkPortGetAPI property specifies the template for the access method with which the data related to the inNetworkPort might be retrieved.

Network ports allow the model to access external data by using a call to a macro/function. It is useful for accessing external data, Memory mapped I/O, Bus messages, and so on.

Available tokens are: \$Name (the name of the inNetworkPort) and \$Attribute (represents the way the data should be retrieved: & \$Attribute – by reference, and \$Attribute = ... - by assignment).

*Default = Get\_ \$Name( & \$Attribute );*

## NetworkPortPollingPolicy

The NetworkPortPollingPolicy property specifies the timing in which input network ports

(inNetworkPort) retrieve the data to which they refer:

The possible values are:

- Synchronous – The data is retrieved during the execution of the doExecute operation of its owning Execution Manager.
- Periodic – The data is retrieved every "period" time units after "delay" time units from system startup. Both "Period" and "Delay" can be defined on the General tab of the Features window for inNetworkPort.

Network ports allow the model to access external data by using a call to a macro/function. It is useful for accessing external data, Memory mapped I/O, Bus messages, and so on.

*Default = Synchronous*

## NetworkPortSetAPI

The NetworkPortSetAPI property specifies the template for the access method with which the data related to the outNetworkPort might be written.

Network ports allow the model to access external data by using a call to a macro/function. It is useful for accessing external data, Memory mapped I/O, Bus messages, and so on.

Available tokens are: \$Name (the name of the outNetworkPort) and \$value (represents the value to be written).

*Default = Set\_ \$Name( \$Value );*

## NetworkPortSignalsAccessLibPath

The NetworkPortSignalsAccessLibPath property specifies a path to the .dll file that returns a list of signals that might later be selected by the user for a given network port. The .dll file needs to implement a set of functions with which the list of signals is retrieved. The list is available for the user in the Features window for inNetworkPort/outNetworkPort.

Network ports allow the model to access external data by using a call to a macro/function. It is useful for accessing external data, Memory mapped I/O, Bus messages, and so on.

Available tokens are: \$Name (the name of the outNetworkPort) and \$value (represents the value to be written).

*Default = Empty string*

## ObjectTypeAsSingleton

The ObjectTypeAsSingleton property generates singleton code for object-types and actors. This functionality saves a singleton-type (actor) in its own repository unit, and manage that unit by using a configuration management tool. Set this property for a single object-type or higher. An object-type is generated as a singleton when all of the following conditions are met:

- The object-type has the «Singleton» stereotype.
- There is one and only one object of the object-type and the object multiplicity is 1.
- The ObjectTypeAsSingleton property is set to True.

Note that when you expose a singleton object (for example, by creating a singleton object-type), Rational Rhapsody also modifies the code that is generated for the singleton.

*Default = Cleared*

## OptimizeStatechartsWithoutEventsMemoryAllocation

The OptimizeStatechartsWithoutEventsMemoryAllocation property determines whether the generated code uses dynamic memory allocation for statecharts that use only triggered operations.

*Default = Cleared*

## Out

The Out property specifies how code is generated when the type is used with an argument that has the modifier "Out."

*Default = \$type\*\**

## PublishedName

This is the name that is used to identify the reactive object in order to send a distributed event to it. If there is only one reactive instance of the class, the value of this property is used to identify the object.

If there is more than one reactive instance of the class, each named explicitly, the name used to identify the reactive object is the name that you have given to the object, and not the property value.

In the case of multiplicity, where the objects are not named explicitly, the name used to identify the reactive object is the published name + the index of the object, for example, if the value of the PublishedName property is truck, then the objects would be identified by truck[0], truck[1]....

*Default = \$name*

## PublishInstance

This Boolean property indicates whether or not the object should be published as a reactive instance that is capable of receiving distributed events.

*Default = Cleared*

## ReactiveThreadSettingPolicy

The `ReactiveThreadSettingPolicy` property specifies how threads are set for reactive classes. The possible values are as follows:

- **Default** - During code generation, Rational Rhapsody adds a thread argument to the constructor.
- **MainThread** - Rational Rhapsody does not add an argument; the thread is set to the main thread.
- **UserDefined** - Rational Rhapsody does not add an argument; you must set the value for the thread yourself.

*Default = Default*

## RecordTypeName

The `RecordTypeName` property specifies the name of the class record type. If this is not set, Rational Rhapsody uses `class_name>_t`.

*Default = empty string*

## RefactorRenameRegularExpression

When you use the `Refactor:Rename` feature, Rhapsody searches the user-defined code in the model for all occurrences of the name of the specified element, including places where the element name appears as part of the name of elements whose name is derived from the element that is being renamed.

To locate these elements, Rhapsody uses a set of regular expressions.

The regular expressions used for this search are defined by the property `RefactorRenameRegularExpression`.

Different values are used for this property, depending on what type of element you are renaming. For example, you will see that this property has a different value under `Class` than it does under `Attribute`. For some types of elements, Rhapsody uses a more generic expression, taken from the value of the property under `ModelElement`.

In general, the value of this property is a regular expression that consists of a number of "Or" components. For the renaming to work properly, each such component must contain three groupings, with the middle grouping using the variable `$keyword`, which represents the name of the element that is being renamed. In some cases, the expression takes this information from the value of another property rather than using the `$keyword` variable.

```
Default = (^its|[^A-Za-z0-9_]its |^$<CG::Relation::Add>Its |[^A-Za-z0-9_] $<CG::Relation::Add>Its  
|^$<CG::Relation::Clear>Its |[^A-Za-z0-9_] $<CG::Relation::Clear>Its  
|^$<CG::Relation::CreateComponent>Its |[^A-Za-z0-9_] $<CG::Relation::CreateComponent>Its  
|^$<CG::Relation::DeleteComponent>Its |[^A-Za-z0-9_] $<CG::Relation::DeleteComponent>Its  
|^$<CG::Relation::Find>Its |[^A-Za-z0-9_] $<CG::Relation::Find>Its |^$<CG::Relation::Get>Its  
|[^A-Za-z0-9_] $<CG::Relation::Get>Its |^$<CG::Relation::Remove>Its  
|[^A-Za-z0-9_] $<CG::Relation::Remove>Its |^$<CG::Relation::Set>Its  
|[^A-Za-z0-9_] $<CG::Relation::Set>Its) ($keyword:c)([^A-Za-z0-9_] |_[1-9])+[^A-Za-z0-9_] |$)  
|([A-Za-z0-9_] |^)( $keyword)([^A-Za-z0-9_] |$) |([A-Za-z0-9_] |^$<C_CG::Operation::PublicName>)
```



## RelativeEventDataRecordTypeComponentsNaming

The RelativeEventDataRecordTypeComponentsNaming property enables relative naming of event data record type components that represent events and triggered operation parameters. If this is True, no events or triggered operations share argument names because they would generate record components with the same name (which would not compile).

*Default = Cleared*

## Renames

The Renames property enables one element to rename another element of the same type. You can also rename an element by using a renames dependency. In the case of a conflict, the dependency has precedence. Note the following:

- For attributes, this property works only for static attributes in a class or for attributes in a package.
- For operations, this property contains the name of the operation this operation is renaming. The signatures of the two operations must match.

*Default = empty string*

## ReturnType

The ReturnType property specifies how code is generated when the type is used as a return type.

*Default = \$type\**

## Simplify

If you are using the Rational Rhapsody customizable code generation mechanism, the Simplify property can be used to change the way specific types of elements are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - The element is ignored.
- Copy - The element is copied from the original to the simplified model. It is not modified in any way.
- Default - Uses the standard simplification for this item, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user.
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for the element has been applied.

*Default = "Default"*

## SimplifyConstructors

If you are using the Rational Rhapsody customizable code generation mechanism, the `SimplifyConstructors` property can be used to change the way constructors are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- `None` - The constructors are ignored.
- `Copy` - The constructor are copied from the original to the simplified model. They do not be modified in any way.
- `Default` - Uses the standard simplification for constructors, as defined in Rational Rhapsody.
- `ByUser` - Uses the customized simplification provided by the user.
- `ByUserPostDefault` - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for constructors has been applied.

*Default = "Default"*

## **SimplifyDestructors**

If you are using the Rational Rhapsody customizable code generation mechanism, the `SimplifyDestructors` property can be used to change the way destructors are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- `None` - The destructors are ignored.
- `Copy` - The destructor are copied from the original to the simplified model. They are not modified in any way.
- `Default` - Uses the standard simplification for destructors, as defined in Rational Rhapsody.
- `ByUser` - Uses the customized simplification provided by the user.
- `ByUserPostDefault` - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for destructors has been applied.

*Default = "Default"*

## **SimplifyPackageFiles**

If you are using the Rational Rhapsody customizable code generation mechanism, the `SimplifyPackageFiles` property can be used to change the way File elements are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- `None` - File elements are ignored.
- `Copy` - File element are copied from the original to the simplified model. They are not modified in any way.
- `Default` - Uses the standard simplification for File elements, as defined in Rational Rhapsody.
- `ByUser` - Uses the customized simplification provided by the user.

- **ByUserPostDefault** - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for File elements has been applied.

*Default = "Default"*

## SingletonExposeThis

The SingletonExposeThis property, when set to False, specifies that all non-static methods are considered as static methods and does not pass in this parameter.

*Default = Cleared*

## SpecificationEpilog

The SpecificationEpilog property allows you to add code to the end of the declaration of a model element.

For example, you can use this property to add an `#ifdef` to indicate that an operation is available only when the code is compiled with `_DEBUG`, by setting the following properties:

- For SpecificationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For SpecificationEpilog, enter the value `#endif`
- For ImplementationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For ImplementationEpilog, enter the value `#endif`

*Default = Blank*

## SpecificationPragmas

The SpecificationPragmas property specifies the user-defined pragmas to generate in the specification.

*Default = Empty MultiLine*

## SpecificationPragmasInContextClause

The SpecificationPragmasInContextClause property specifies the user-defined pragmas to generate in the context clause of the specification.

*Default = Empty MultiLine*

## SpecificationProlog

The SpecificationProlog property allows you to add code to the beginning of the declaration of a model element.

For example, you can use this property to add an `#ifdef` to indicate that an operation is available only when the code is compiled with `_DEBUG`, by setting the following properties:

- For SpecificationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For SpecificationEpilog, enter the value `#endif`
- For ImplementationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For ImplementationEpilog, enter the value `#endif`

*Default = Blank*

## SpecIncludes

The SpecIncludes property specifies the names (including full paths) of header files to be included at the top of specification files generated for classes (C++ and Java), objects or object types (C), and packages. Separate multiple file names by using commas, without spaces.

*Default = empty string*

## TargetMonitoringSerializeAttributesDelay

The TargetMonitoringSerializeAttributesDelay property specifies the time from system startup until the first time attributes are serialized and sent from a target to Rational Rhapsody when using target monitoring.

Target monitoring is an animation instrumentation mode in which instrumentation code is optimized and limited, and the user cannot control the execution of the application on the target, only to view its status.

*Default = 0U*

See also the properties for Animation::TargetMonitoring.

## TargetMonitoringSerializeAttributesPeriod

The TargetMonitoringSerializeAttributesPeriod property specifies the time between two executive attributes serialization from a target to Rational Rhapsody when using target monitoring.

Target monitoring is an animation instrumentation mode in which instrumentation code is optimized and limited, and the user cannot control the execution of the application on the target, only to view its status.

*Default = 500U*

See also the properties for Animation::TargetMonitoring.

## TaskBody

The TaskBody property defines an alternate task body for Ada Task and Ada Task Type classes.

*Default = empty string*

## TriggerArgument

*The TriggerArgument property specifies how the type should be passed in when used as an argument for events\triggered operations. By default, classes that are used as an argument are passed by way of a pointer. There are 4 other properties that effect how types are passed into and returned by regular operations: "in", "out", "in/out" and "return." (Default = \$type\*) See also:*

- In
- InOut
- Out

*Default = \$type\**

## TypedefStructSuffix

Prior to version 7.5, a MISRA rule was violated in code generated by Rational Rhapsody because certain typedef statements used the same string for both the data type and the alias to use, for example:

```
typedef struct t_Test_Event t_Test_Event;
```

To alleviate this problem, the TypedefStructSuffix property was added to the MISRA98 profile with a default value of "\_t".

The value of the property is used as a suffix for the data type. So the above code example is generated as:

```
typedef struct t_Test_Event_t t_Test_Event;
```

## Visibility

The Visibility property is used to determine whether the struct definition and function prototypes for a class are generated in an .h file or in the .c file for the class.

If the property is set to Public, these elements are generated in the .h file. If the property is set to Private, these elements are generated in the .c file.

*Default = Public*

## Configuration

The Configuration metaclass contains properties for implementing configurations.

## AllCategoriesInitializingMode

The AllCategoriesInitializingMode property specifies mode of initialization of the data structures (objects,

relations, ports, user attributes, and so on) for a system:

- RunTime - The system is initialized at run time.
- CompileTime - The system is initialized at compile time.
- ByCategory - The system is initialized differently for different categories.

Each category is represented by a property:

C.CG::Configuration::FrameworkInitializingMode

C.CG::Configuration::RelationInitializingMode

C.CG::Configuration::DirectFlowPortsInitializingMode

C.CG::Configuration::AttributeInitializingMode

## AsyncActiveActivationDelay

The AsyncActiveActivationDelay property specifies the delay time (in time units), from system startup, in which an Asynchronous Active is activated.

*Default = 0U*

## AsyncActiveActivationPeriod

The AsyncActiveActivationPeriod property specifies the time period (in time units) between two executive executions of an Asynchronous Active.

*Default = 10U*

## AttributeInitializingMode

The AttributeInitializingMode property specifies mode of initialization of user attributes:

- RunTime - The attributes initialize at run time
- CompileTime - The attributes initialize at compile time

Valid only when the C.CG::Configuration::AllCategoriesInitializingMode property is set to "ByCategory."

## ClassesPerCGCall

The ClassesPerCGCall property can be used to specify the maximum number of classes that Rational Rhapsody should include in a single code generation "chunk". Above this number, the code generation action will be broken into a number of smaller code generation actions. For example, if you specify 500 for the value of the property, then if your model has 501-1000 classes, Rational Rhapsody will try to break the code generation action into two smaller code generation actions.

Note that while the property name includes the term *Classes*, this number also takes into account similar model elements, such as actors and files.

If the value is set to -1, the chunking mechanism is not used, meaning that the code generation action will not be broken into a number of smaller actions regardless of how large the model is.

*Default = -1*

## ClassStateDeclaration

The *ClassStateDeclaration* property supports C compilers that cannot handle enum declarations inside struct declaration. The possible values are as follows:

- *InClassDeclaration* - Generate the reactive statechart enum declaration in the class declaration.
- *BeforeClassDeclaration* - Generate the reactive class statechart enum declaration before the declaration of the class.

*Default = InClassDeclaration*

## CodeGenerationDirectoryLevel

The *CodeGenerationDirectoryLevel* property is found in the pre-72 compatibility profiles for C and C++.

The directories specified with the *DefaultSpecificationDirectory* and *DefaultImplementationDirectory* properties were created at the beginning of the path to the generated files, for example, `..\spec_directory\package_a\subpackage_1` and `..\impl_directory\package_a\subpackage_1`.

Beginning with version 7.2 of Rational Rhapsody, the directories specified with *DefaultSpecificationDirectory* and *DefaultImplementationDirectory* are created at the end of the path to the generated files, for example, `..\package_a\subpackage_1\spec_directory` and `..\package_a\subpackage_1\impl_directory`.

To provide the old code generation behavior for pre-72 models, the compatibility profiles include the *CodeGenerationDirectoryLevel* property, with the default value of the property set to *Top*. If you want your pre-72 models to use the new behavior that was introduced in version 7.2, change the value of this property to *Bottom*.

*Default = Top*

## CodeGeneratorTool

The *CodeGeneratorTool* property specifies which code generation tool to use for the given configuration. The possible values are as follows:

- *Advanced* - Rational Rhapsody uses its internal code generator to generate code
- *External* - instructs Rational Rhapsody to use the registered external code generator
- *Customizable* - instructs Rational Rhapsody to use the customizable code generation mechanism.

If the property is set to Customizable, Rational Rhapsody carries out the following steps:

1. Creates a refined model from the original model. This model is referred to as the simplified model. (This step represents the transformation stage.)
2. Invokes the external RulesComposer code writer to create the code itself. (This step represents the writing stage.)

Note: The RulesComposer code writer is a .dll that is installed in the framework of the standard Rational Rhapsody installation. However, you can only use this writer if you have a valid license for this feature.

Both of these steps (creation of the simplified model and generation of code from the simplified model) can be customized.

*Default = Advanced*

## **CommentaryCompileTimeInit**

When compile-time initialization is used with the MicroC profile, the property CommentaryCompileTimeInit specifies whether the generated initialization code should include comments to represent the names of the struct fields that being initialized.

*Default = True*

## **CompileTimeInitializationMode**

When you use the MicroC profile, you have the option of specifying that elements should be initialized at compile time. If you are using compile-time initialization, the property CompileTimeInitializationMode determines what fields of a struct are initialized. The possible values of this property are:

- RequiredFieldsOnly - when this option is used, only the fields for which values have been specified, and the fields preceding them in the struct are initialized
- AllFields - when this options is used, all fields are initialized, even those that come after the last field for which you have specified a value. This option is useful if you are using a compiler that won't fill in remaining fields with zeros

*Default = RequiredFieldsOnly*

Note that you can use the following properties to specify the initialization values that should be used for fields for which an initialization value was not specified:

- C\_CG::Configuration::ZeroInitValueBoolean
- C\_CG::Configuration::ZeroInitValueInteger
- C\_CG::Configuration::ZeroInitValuePointer
- C\_CG::Configuration::ZeroInitValueReal
- C\_CG::Configuration::ZeroInitValueString
- C\_CG::Configuration::ZeroInitValueOther



## ContainerSet

The ContainerSet property specifies the container set used to implement relations.

*Default = RiCContainers*

## CustomizableCG

The CustomizableCG property is deprecated as of Rational Rhapsody 7.2.

To specify that customized code generation should be used, set the value of the C\_CG::Configuration::CodeGeneratorTool property to Customizable.

The description below was relevant prior to version 7.2 of Rational Rhapsody.

When you instruct Rational Rhapsody to generate code, Rational Rhapsody takes one of two different paths, depending on the value of the C\_CG::Configuration::CustomizableCG property.

If CustomizableCG is set to Cleared, Rational Rhapsody starts its standard internal code generation mechanism. (This is the default setting for the property.)

If CustomizableCG is set to Checked, Rational Rhapsody carries out the following steps:

1. Creates a refined model from the original model. This model is referred to as the simplified model. (This step represents the transformation stage.)
2. Invokes the external RulesComposer code writer to create the code itself. (This step represents the writing stage.)

Note: The RulesComposer code writer is a .dll that is installed in the framework of the standard Rational Rhapsody installation. However, you can only use this writer if you have a valid license for this feature.

Both of these steps (creation of the simplified model and generation of code from the simplified model) can be customized.

*Default = Cleared*

## DefaultImplementationDirectory

The DefaultImplementationDirectory property specifies the relative path to the default directory for generated implementation files. The value of this property is added after the configuration path. Consider the following case:

- File C.cpp is an implementation of class C mapped to a folder Foo.
- The active configuration (cfg) is under component cmp.
- DefaultImplementationDirectory is set to “src”

Rational Rhapsody generates C.cpp to root>\cmp\cfg\src\Foo. Note the following limitations:

- This feature is not supported in COM- or CORBA-related components (C++ only).
- The predefined OSE environments (OsePPCDiab and OseSfk) are not supported due to makefile flexibility issues.
- This feature is not supported by the INTEGRITY adapter build file generator.

*Default = empty string*

## DefaultROMSegmentName

The DefaultROMSegmentName property specifies the name of the default memory segment used for data related to model elements such as relations and flow ports. This property is relevant only when the Segmented Memory capability is enabled (see CG::Configuration::EnableSegmentedMemory).

*Default = ROM*

## DefaultSpecificationDirectory

The DefaultSpecificationDirectory property specifies the relative path to the default directory for generated specification files. The value of this property is added after the configuration path. Consider the following case:

- File B.h is a specification of class B that is not mapped to any file.
- The active configuration (cfg) is under component cmp.
- DefaultSpecificationDirectory is set to “inc”

Rational Rhapsody generates B.h to root>\cmp\cfg\inc. Note the following limitations:

- This feature is not supported in COM- or CORBA-related components (C++ only).
- The predefined OSE environments (OsePPCDiab and OseSfk) are not supported due to makefile flexibility issues.
- This feature is not supported by the INTEGRITY adapter build file generator.

*Default = empty string*

## DependencyRuleScheme

The DependencyRuleScheme property specifies how dependency rules should be generated in the makefile. The possible values are as follows:

- Basic - Generates only the local implementation and specification files in the dependency rule in the makefile.
- ByScope - In addition to generating the same files as the Basic option, generates the specification files of related elements (dependencies, associations, generalizations, and so on) that are in the scope of the active component.
- Extended - In addition to generating the same files as the ByScope option, generates the specification files of related external elements (specified by using the CG::Class::UseAsExternal and CG::Package::UseAsExternal properties) and elements that are not in the scope of the active component.

*Default = ByScope*

## DescriptionBeginLine

This property specifies the prefix for the beginning of comment lines in the generated code. This functionality uses a documentation system (such as Doxygen), which looks for a certain prefix to produce the documentation.

This property affects only the code that is generated for descriptions of model elements; other auto-generated comments are not affected.

*Default = /\**

## DescriptionEndLine

This property specifies the prefix for the end of comment lines in the generated code. This functionality uses a documentation system (such as Doxygen), which looks for a certain prefix to produce the documentation. This property affects only the code that is generated for descriptions of model elements; other auto-generated comments are not affected.

*Default = \*/*

## DirectFlowPorts

The DirectFlowPorts property specifies whether the tool attempts to optimize the code for flow ports.

*Default = Checked*

## DirectFlowPortsCallbackFuncName

The DirectFlowPortsCallbackFuncName property specifies the name of the generated attribute which holds the callback function of the object connected to an optimized direct flow port.

## DirectFlowPortsInitializingMode

The DirectFlowPortsInitializingMode property specifies mode of initialization of the optimized direct generated attributes for a flow port:

- RunTime - The attributes of the flow port are initialized at run time.
- CompileTime - The attributes of the flow port are initialized at compile time.

Valid only when the C\_CG::Configuration::AllCategoriesInitializingMode property is set to "ByCategory."

## DirectFlowPortsTargetName

The `DirectFlowPortsTargetName` property specifies the name of the generated attribute which points to the target object connected to an optimized direct flow port.

## DirectRelations

The `DirectRelations` property specifies whether the tool attempts to optimize the code for relations.

*Default = Checked*

## EmptyArgumentListName

The `EmptyArgumentListName` specifies the string generated for the argument list when an operation has no arguments. For example, if you set this value to "void," for an operation `foo` that has no arguments, Rational Rhapsody generates the following code:

```
int foo (void){ ... }
```

*Default = empty string*

## EnableSegmentedMemory

The `EnableSegmentedMemory` property specifies whether the Segmented Memory capability is enabled. The segmented memory capability can be used only when initializing the system in compile-time.

## Environment

The `Environment` property determines the target environment for a configuration. Generated code is targeted for that environment. See the Release Notes for the environments supported by Rational Rhapsody “out-of-the-box.”

“Out-of-the-box” support means that Rational Rhapsody includes a set of preconfigured code generation properties for the environment and precompiled versions of the relevant OXF libraries. The precompiled OXF libraries have been fully tested. You can also add new environments, for example if you want to generate code for another RTOS.

This involves retargeting the OS wrapper files in the Rational Rhapsody framework and creating a new set of code generation properties for the target environment.

*Default = MSVC*

## ExternalGenerationTimeout

The `ExternalGenerationTimeout` property specifies how long, in seconds, Rational Rhapsody waits for the each class in the configuration scope to complete so you can once again make changes to the model.

This property applies to both the full-featured external generator and makefile generator. For example, if

you set this property to 2 and you have 10 classes, Rational Rhapsody sets a timeout of 20. If the external code generator does not complete generation in this timeframe, Rational Rhapsody displays a message in the output window saying that the generator is not responding, and you are allowed to make changes to the model.

If you set this property to 0, Rational Rhapsody does not time out the generation session, and waits for the code generator to complete its task - even if it takes forever. Rational Rhapsody waits for a notification from the full-featured external code generator, or for the process termination of a makefile generator.

*Default = 0*

## ExternalGeneratorFileMappingRules

The ExternalGeneratorFileMappingRules property specifies whether the external code generator uses the same file mapping and naming scheme ( mapping rules ) as Rational Rhapsody. If the mapping rules are different, the external generator must implement handlers to the GetFileName, GetMainFileName, and GetMakefileName events that Rational Rhapsody runs to get a requested file name and path.

The possible values are as follows:

- AsRhapsody - The external generator uses the same mapping rules as Rational Rhapsody.
- DefinedByGenerator - The external generator has its own mapping rules.

*Default = AsRhapsody*

## FarPointerQualifier

The FarPointerQualifier property specifies the keyword to be used for a "far" pointer definition.

This property is used when the Segmented Memory capability is enabled (see CG::Configuration::EnableSegmentedMemory) for the data declaration of the references between memory segments for a class.

*Default = empty*

## FrameworkCollectionStaticSize

Use this property to define the collection container static size for the MicroC Execution Framework (MXF).

*Default = 20U*

## FrameworkEventsPoolSize

The FrameworkEventsPoolSize property specifies the size of the RiCEvent pool in the MicroC Execution Framework (MXF). This number represents the number of concurrent pending events in all the event queues in the model.

*Default = 10U*

## FrameworkHeapConstSize

Use this property to define the heap container static size for the MicroC Execution Framework (MXF).

*Default = 10U*

## FrameworkInitializingMode

The FrameworkInitializingMode property specifies mode of initialization of the related generated attributes (such as ric\_task) for the framework:

- RunTime - The attributes for the framework initialize at run time.
- CompileTime - The attributes for the framework initialize at compile time.

Valid only when the C\_CG::Configuration::AllCategoriesInitializingMode property is set to "ByCategory."

## FrameworkTimeoutsPoolSize

Use this property to define the timeouts pool static size for the MicroC Execution Framework (MXF).

*Default = 10U*

## GeneratorExtraPropertyFiles

The GeneratorExtraPropertyFiles property opens the default Text Editor allowing the user to edit the \$OMROOT\CodeGenerator\GenerationRules\LangC\RiC\_CG.ini file.

## GeneratorRulesSet

The GeneratorRulesSet property allows you to specify your own rule set to use for customized code generation.

*Default = \$OMROOT\CodeGenerator\GenerationRules\LangC\CompiledRules\RiCWriter.classpath*

## GeneratorScenarioName

The GeneratorScenarioName property specifies which scenario to use for the rule set, if you write your own set of code generation rules.

*Default = scenarios.Rhapsody\_Generation.main*

## GenericEventHandling

The GenericEventHandling property is a Boolean value that determines whether to generate generic event-handling code. This property supports large-scale collaboration, where you might not be aware of which classes consume a base event of your part in the event hierarchy, and might not have access to parts of the model that use base events.

The framework base event class includes a virtual method that checks the event ID against the specified ID, thereby supplying a generic mechanism for events without super events. The language-specific methods are as follows: C:

```
#define RiCEvent_isTypeOf(event, id) ((event)-IId == (id)) C++: virtual OMBoolean isTypeOf(short id) const {return IId ==id;}
```

Each generated event that has a super event overrides the method to check the ID against its own ID, then calls its base event directly to continue the check. An event without a base event returns Cleared if the ID does not equal its own.

When you set the GenericEventHandling property to Cleared, event consumption code is generated. Setting this property affects only the way events are consumed - the override on the isTypeOf() method is still generated, to allow handling of events in components that use the generic event handling. To support complete generic event handling, you should regenerate the code for all events and reactive classes.

*Default = Cleared*

## ImplementationEpilog

The ImplementationEpilog property enables you to add any code that you want to be added as verbatim text (to be ignored by Rational Rhapsody) to the end of the definition of a model element.. For example, you could wrap a section of code with an #ifdef-#endif pair, add compiler-specific keywords, or add a #pragma statement. For example, to specify that an operation is available only when the code is compiled with \_DEBUG, set the following properties for the operation:

- Set SpecificationProlog to #ifdef \_DEBUG cr.
- Set SpecificationEpilog to #endif.
- Set ImplementationProlog to #ifdef \_DEBUG cr.
- Set ImplementationEpilog to #endif.

The following table lists whether leading and trailing linefeeds are generated.

Metaclass	Leading Linefeed	Added?	Class	Yes	Package	No
-----------	------------------	--------	-------	-----	---------	----

*Default = Empty MultiLine*

## ImplementationProlog

The ImplementationProlog property enables you to add any code that you want to be added as verbatim text (to be ignored by the software) to the beginning of the definition of a model element. For example,

you could wrap a section of code with an `#ifdef-#endif` pair, add compiler-specific keywords, or add a `#pragma` statement. For example, to specify that an operation is available only when the code is compiled with `_DEBUG`, set the following properties for the operation:

- Set `SpecificationProlog` to `#ifdef _DEBUG cr.`
- Set `SpecificationEpilog` to `#endif.`
- Set `ImplementationProlog` to `#ifdef _DEBUG cr.`
- Set `ImplementationEpilog` to `#endif.`

The following table lists whether leading and trailing linefeeds are generated.

Metaclass Trailing Linefeed Added? Class No Package Yes

*Default = Empty MultiLine*

## Indentation

The `Indentation` property specifies the number of spaces that should be used for indentation when Rhapsody generates code.

Note that this property does not affect the indentation used in operation bodies since that code is provided by the user.

*Default = 4*

## InitializeEmbeddableObjectsByValue

The `InitializeEmbeddableObjectsByValue` property specifies whether embeddable classes and object types selected in the configuration initial instances list should be allocated by value in the `main()` routine.

*Default = Cleared*

## LocalVariablesDeclaration

The `LocalVariablesDeclaration` property specifies variables that you want to appear in the declaration of the entrypoint or operation.

*Default = Empty MultiLine*

## MainFunctionArgList

This property provides a list of the main function arguments. The default list is `"int argc, char* argv[]."`

*Default = int argc, char\* argv[]*



## MainFunctionRetType

The MainFunctionRetType property determines the return type for the "main" function generated by Rational Rhapsody. Note that if the value of the property is a blank string, the return type generated will be "int".

*Default = Blank*

## MainTaskName

The MainTaskName property specifies the name of the main task in the MicroC Execution Framework (MXF). The main task is generated for model elements that require management of an active class and are not owned by any active class in the model.

*Default = RiCTheMainTask*

## MarkPrologEpilogInAnnotations

The MarkPrologEpilogInAnnotations property specifies whether to generate ignore annotations for the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties so they are ignored during roundtrip. When you insert code element declarations (variables, types, functions, and so on) in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, after a full roundtrip those elements are added to the model and are duplicated on the next code generation.

When you use the MarkPrologEpilogInAnnotations property, you can have Rational Rhapsody automatically ignore the information specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties instead of adding the ignore annotations manually.

The possible values for the MarkPrologEpilogInAnnotations property are:

- None - Rational Rhapsody does not generate any annotations. Any models created before Version 4.1 automatically have this property setting.
- Ignore - Rational Rhapsody generates the `/// ignore annotation before the code specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, and generates the /// annotation after the code specified in those properties.`
- Auto - If the code in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties is one line (it does not contain any newline characters `\n`), no annotations are generated (the same behavior has the None setting). If there is more than one line, Rational Rhapsody generates the `/// ignore annotation before the code specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, and generates the /// annotation after the code specified in those properties (the same behavior as the Ignore setting).`

During roundtrip, any ignore annotations in the comments of the element are not included in its description. Because the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties are generated between the annotation and its declaration for the element, you cannot rename those elements on roundtrip. If you change the name of an element, it is removed from

the model and added with the new name. Some model information (for example, property settings) might be lost.

*Default = None*

## MultipleAddressSpaces

When this property is set to Checked, Rational Rhapsody uses the code generation settings that are required for use of the multiple address space feature.

Since the default value of this property is Cleared, you must change the value to enable this feature.

*Default = Cleared*

## mxfCfgAdditionalIncludes

The mxfCfgAdditionalIncludes property specifies the names of header files to be included at the top of the mxf\_cfg.h file. This is a multiline field that can contain additional include "<file.h>" statements separated by a new line.

*Default = Empty MultiLine*

## mxfCfgTemplate

The mxfCfgTemplate property serves as a template for generating the header file (by default named mxf\_cfg.h) which is used for configuring the build of the MicroC Execution Framework (MXF).

You can include the values of various Rational Rhapsody properties in the generated .h file, by using either of the following two substitution mechanisms:

For properties located under C\_CG::Configuration or under C\_CG::<environment>, you can use the format \$<property name> and Rational Rhapsody gets the value of that property. For example, if the value of the MyProp property is 100, you can include a line like #define MY\_FLAG \$<MyProp> in the template property, and in the generated .h file this is replaced by #define MY\_FLAG 100.

For any Rational Rhapsody property that applies to the active configuration, you can use the format \$<Subject.MetaClass.Property> and Rational Rhapsody gets the value of that property. For example, if the value of the C\_CG::Configuration::Environment property is "Microsoft." you can include a line like #define MY\_ENVIRONMENT \$<C\_CG.Configuration.Environment> in the template property, and in the generated .h file, this is replaced by #define MY\_ENVIRONMENT Microsoft.

## ImplicitObjectTypeSuffix

For each implicit object in your model, generated C++ code includes a class that the object instantiates. Generated C code contains a struct that is the type of the implicit object. To differentiate the class/struct from the object, the name of the class/struct is composed of the name of the object plus a suffix. You can use the property ImplicitObjectTypeSuffix to customize the suffix that is used.

*Default = \_t*

## ReactiveMaxNullSteps

Use this property to define the maximum NULL steps for a reactive trying to do a "run-to-completion" for the MicroC Execution Framework (MXF).

*Default = 100*

## RelationDelimiter

When Rational Rhapsody generates C code for bidirectional associations, the code contains double underscores, which are a violation of the ANSI C++ standard. In release 8.1.2.1, the property C\_CG::Configuration::RelationDelimiter was added to provide an option of generating code that does not include double underscores.

*Default = \_ (underscore)*

## RelationInitializingMode

The RelationInitializingMode property specifies mode of initialization of the generated attributes for the optimized direct relation:

- RunTime - The attributes of the relation are initialized at run time.
- CompileTime - The attributes of the relation are initialized at compile time.

This property is valid only when the C\_CG::Configuration::AllCategoriesInitializingMode property is set to "ByCategory."

*Default = CompileTime*

## ShowCgSimplifiedModelPackage

The first step of the code generation process consists of the building of a simplified model based on the Rational Rhapsody model.

By default, the simplified model is not displayed in Rational Rhapsody. To have the simplified model displayed in the browser, set the ShowCgSimplifiedModelPackage property property to True. Once you have done so, the next time you generate code, the simplified model is added automatically at the top of the project tree in the browser.

*Default = Cleared*

## SimplifyMainFiles

If you are using the Rational Rhapsody customizable code generation mechanism, the SimplifyMainFiles

property can be used to change the way main files are handled by Rational Rhapsody when it transforms the model into a simplified model. This allows you to customize code generation for main files, beyond the initialization code you can specify in Rational Rhapsody at the configuration level.

The property can take any of the following values:

- None - Main files are ignored.
- Copy - Main file are copied from the original to the simplified model. They are not modified in any way.
- Default - Uses the standard simplification for main files, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user.
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for main files has been applied.

*Default = "Default"*

## **SimplifyMakeFile**

If you are using the Rational Rhapsody customizable code generation mechanism, the SimplifyMakeFile property can be used to change the way makefiles are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - Makefile elements are ignored.
- Copy - Makefile element are copied from the original to the simplified model. They are not modified in any way.
- Default - Uses the standard simplification for makefiles, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user.
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for makefiles has been applied.

*Default = "Default"*

## **SpecificationEpilog**

The SpecificationEpilog property allows you to add code to the end of the declaration of a model element.

For example, you can use this property to add an `#ifdef` to indicate that an operation is available only when the code is compiled with `_DEBUG`, by setting the following properties:

- For SpecificationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For SpecificationEpilog, enter the value `#endif`
- For ImplementationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For ImplementationEpilog, enter the value `#endif`

*Default = Blank*

## SpecificationProlog

The SpecificationProlog property allows you to add code to the beginning of the declaration of a model element.

For example, you can use this property to add an `#ifdef` to indicate that an operation is available only when the code is compiled with `_DEBUG`, by setting the following properties:

- For SpecificationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For SpecificationEpilog, enter the value `#endif`
- For ImplementationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For ImplementationEpilog, enter the value `#endif`

*Default = Blank*

## StatechartImplementation

The StatechartImplementation property is available for the MicroC profile to set the implementation of statecharts. The possible values are:

- Flat - In the Flat implementation, states are implemented as enumerated types.
- OptimizedTopDown - In the OptimizedTopDown implementation, the code is optimized, as compared to the "Flat" statechart implementation, in terms of memory consumption (ROM, RAM), run-time efficiency, and code size. It is top down in the sense of statechart semantics.

If you set the StatechartImplementation property to OptimizedTopDown, use the properties in `C_CG::OptimizedTopDownStatechart` to control the optimization of top-down code generation for statecharts.

*Default = Flat*

## TimerManagerDefaultMaxTm

The TimerManagerDefaultMaxTm property specifies the MicroC Execution Framework (MXF) Timer Manager default maximum time.

*Default = 100U*

## TimerManagerDefaultTickTime

The TimerManagerDefaultTickTime property specifies the MicroC Execution Framework (MXF) Timer Manager internal time counter incrementation rate. This is the number of time units that the counter is incremented with each time the Timer Manager is invoked.

*Default = 10U*

## TimerManagerOverflowMark

The TimerManagerOverflowMark property specifies the MicroC Execution Framework (MXF) Timer Manager internal time counter overflow value. This is the value on which the Timer Manager internal counter overflows.

*Default = 0x80000000UL*

## TimerManagerTimeCycle

The TimerManagerTimeCycle property specifies the time in mili-seconds between two executive invocations of the Timer Manager for the MicroC Execution Framework (MXF).

*Default = 1U*

## TransformDiagrams

The TransformDiagram property controls whether diagrams are copied or not.

*Default = false*

## TransformationSequence

The TransformationSequence property specifies the transformation sequence based on a list of comma-separated transformer names.

*Default = false*

## UseMainTask

The UseMainTask property specifies whether a main task should be generated for model elements that are not owned by any active class in the model.

## ZeroInitValueBoolean

When compile-time initialization is used with the MicroC profile, the property ZeroInitValueBoolean specifies the initialization value that should be used for primitive attributes of type boolean if an initialization value was not specified for them.

*Default = FALSE*

## ZeroInitValueInteger

When compile-time initialization is used with the MicroC profile, the property ZeroInitValueInteger specifies the initialization value that should be used for primitive attributes of type integer if an initialization value was not specified for them.

*Default = 0*

## ZeroInitValueOther

When compile-time initialization is used with the MicroC profile, the property ZeroInitValueOther specifies the initialization value that should be used for primitive attributes of types other than boolean, integer, pointer, real, and string, if an initialization value was not specified for them.

*Default = 0*

## ZeroInitValuePointer

When compile-time initialization is used with the MicroC profile, the property ZeroInitValuePointer specifies the initialization value that should be used for primitive attributes of type pointer if an initialization value was not specified for them.

*Default = NULL*

## ZeroInitValueReal

When compile-time initialization is used with the MicroC profile, the property ZeroInitValueReal specifies the initialization value that should be used for primitive attributes of type real if an initialization value was not specified for them.

*Default = 0.0*

## ZeroInitValueString

When compile-time initialization is used with the MicroC profile, the property ZeroInitValueString specifies the initialization value that should be used for primitive attributes of type string if an initialization value was not specified for them.

*Default = "" (empty string)*

## Cygwin

The Cygwin metaclass controls the environment settings (Compiler, framework libraries, and so on) for

Cygwin.

## AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)/LangC/osconfig/Cygwin*

## AdditionalReservedWords

The AdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

## AnimIncludeDirectories

The AnimIncludeDirectories property is used to specify the directories that must be referenced in the makefile for projects where Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with INST\_INCLUDES.

*Default = \$(INCLUDE\_QUALIFIER)\$(OMROOT)/LangC/aom  
\$(INCLUDE\_QUALIFIER)\$(OMROOT)/LangCpp/tom*

## AnimInstLibs

The AnimInstLibs property is used to specify the static libraries that are required when Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with INST\_LIBS.

*Default = \$(OMROOT)/LangC/lib/cygwinaomanim\$(CPU)\$(LIB\_EXT)*



## AnimOxfLibs

The AnimOxfLibs property is used to specify the framework libraries that are required when Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with OXF\_LIBS.

*Default = \$(OMROOT)/LangC/lib/cygwinofirst\$(CPU)\$(LIB\_EXT)  
\$(OMROOT)/LangC/lib/cygwinomcomappl\$(CPU)\$(LIB\_EXT)*

## AnimPreprocessor

The AnimPreprocessor property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with INST\_FLAGS.

*Default = \$(DEFINE\_QUALIFIER)OMANIMATOR \$(DEFINE\_QUALIFIER)\_\_USE\_W32\_SOCKETS*

## BuildArgumentsInIDE

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the C\_CG::[environment]::BuildCommandInIDE property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = \$makefile*

## BuildCommandInIDE

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = cygwinmake.bat*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release

version of the configuration. To change this property, use the Configuration window in the browser - do not change it by using the Properties window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link.

The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*Default = Debug*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\"\\etc\\cygwinmake.bat cygwinbuild.mak build \"BUILD\_SET=\$BuildCommandSet\" \"CPU=\$CPU\" \"\""*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Checked*

## CompileCommand

The CompileCommand property sets the Cygwin compiler name.

*Default = gcc*

## CompileRelease

The CompileRelease property specifies additional compilation flags for a configuration set to Release mode.

## CompilerFlags

The CompilerFlags property allows you to define additional compilation flags. The value of the property is inserted into the value of the CompileSwitches property (Linux) or CPPCompileSwitches (cygwin). In the generated makefile, you can see the value of this property in the line that begins with ConfigurationCPPCompileSwitches=.

*Default = Blank*

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default =*

```
@echo Compiling $OMFileImpPath @$((CC)) $OMFileCPPCompileSwitches -o $OMFileObjPath  
$OMFileImpPath
```

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component. The default values are as follows:

*Default = -g*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*Default = -O*

## CPPCompileSwitches

The CPPCompileSwitches property specifies the compiler switches.

*Default = \$IncludeDirectories \$DefinedSymbols \$(INST\_FLAGS) \$(INCLUDE\_PATH)  
\$(INST\_INCLUDES) \$CompilerFlags \$OMCPPCompileCommandSet -c*

## CPU

The CPU property is used to specify the CPU of the target environment.

When building applications or framework libraries for MSVC9 environments, Rhapsody uses a variable called \$CPU which represents the value provided for the CPU property for these environments. This variable is provided as a parameter to the msvc9make.bat batch file, and is also used in the makefiles that are generated.

When building 32-bit applications, use the value x86 for this property.

When building 64-bit applications, use the value x64 for this property.

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

*Default = \$OMFileObjPath : \$OMFileImpPath \$OMFileSpecPath \$OMFileDependencies*

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: \$OMFileObjPath : \$OMFileImpPath "\*.bmp" "\*.ico" "\*.rc2"

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction an IDE such as Eclipse, you can use the EnableDebugIntegrationWithIDE property to specify whether or not the IDE debugger is to be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to Checked, the IDE debugger is used.

*Default = Checked*

## EndApplicationCode

When the main function ends, the system's exit() function is called automatically. The system function calls the destructors of framework instances in the order it determines. The EndApplicationCode property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an endApplication() function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to endApplication(), it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The EntryPoint property specifies the name of the main program for a given environment.

*Default = main*

You might also want to use the "Filter" facility in this window to refer to the definition of the EntryPointDeclarationModifier property for more information.

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property.

ErrorMessageTokens has three parameters, each with an integer value:

- TotalNumberOfTokens - The number of tokens in the regular expression
- FileTokenPosition - The position of the file name token in the expression
- LineTokenPosition - The position of the line number token in the expression

*Default = ToTalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2*

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the C\_CG::<Environment>::ExeName property plus the value of the ExeExtension property.

*Default = .exe*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the C\_CG::<Environment>::ExeExtension property.

*Default = Blank*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements. The file inclusions are generated in the makefile.

This default value is \$OMSpecIncludeInElements \$OMImpIncludeInElements.

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile.

Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

*Default = \$(OMROOT)\LangC\lib\cygwinWebComponents\$(CPU)\$(LIB\_EXT),  
\$(OMROOT)\lib\cygwinWebServices\$(CPU)\$(LIB\_EXT), -lws2\_32*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment. The default values are as follows:

*Default = .c*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*Default = "include."*

## InvokeAnimatedExecutable

The AnimationPortRange value in the rhapsody.ini file makes it possible to use animation with multiple instances of Rhapsody running simultaneously.

Ordinarily, when you run an application with animation, the executable is launched using the command specified by the InvokeExecutable property. However, if you have specified a value greater than zero for AnimationPortRange in your rhapsody.ini file, the executable is launched using the command specified by the InvokeAnimatedExecutable property. By default, the value of InvokeAnimatedExecutable includes a reference to the port to use for animation.

*Default = "\"\$OMROOT/etc/Executer.exe\" \"\" \"\$OMROOT\\\"\\etc\\cygwinrun.bat \$executable -port*

*\$port\*""

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT/etc/cygwinrun.bat\" \$executable"*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

*"rhapsody\_dir>\share/etc/Executer.exe""dir; dir | more"*

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

*"\$OMROOT/etc/Executer.exe" "\"\$OMROOT/etc/vx6make.bat\" \$makefile \$maketarget 6.2 \$BSP gnu"*

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\"etc/cygwinmake.bat \$makefile \$maketarget  
\"CPU=\$CPU\" \" \"*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .a*

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

*Default = -g*

## LinkerFlags

The LinkerFlags property allows you to define linker flags. The value of the property is inserted into the value of the LinkSwitches property. In the generated makefile, you can see the value of this property in the line that begins with LINK\_FLAGS=.

*Default = Blank*

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

*Default = -O*

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode. The default values are as follows:

*Default = \$OMLinkCommandSet \$LinkerFlags*

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the C\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*Default = .mak*

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros



- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration. For example, the default content of the target type section of a C++ makefile for the Microsoft environment is as follows:

```
##### Target type (Debug/Release)
#####
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease BuildSet=$OMBuildSet
SUBSYSTEM=$OMSubSystem COM=$OMCOM RPFramewOrkDll=$OMRPFramewOrkDll
ConfigurationCPPCompileSwitches= $OMReusableStatechartSwitches
$OMConfigurationCPPCompileSwitches !IF "$ (RPFramewOrkDll)" == "True"
ConfigurationCPPCompileSwitches= $(ConfigurationCPPCompileSwitches) /D "FRAMEWORK_DLL"
!ENDIF !IF "$ (COM)" == "True" SUBSYSTEM=/SUBSYSTEM:windows !ENDIF
```

**Compilation Flags** The compilation flags section of the makefile contains the default compilation flags stored in the CompileSwitches property. For example, the default content of the compilation flags section of a C++ makefile for the Microsoft environment is as follows:

```
##### Compilation flags
#####
INCLUDE_QUALIFIER=/I LIB_PREFIX=MS
```

**Commands Definitions** The commands definition section of the makefile specifies programs to execute from the makefile. For example, the default commands definition section of a C++ makefile for the Microsoft environment is as follows:

```
##### Commands definition #####
##### RMDIR = rmdir LIB_CMD=link.exe -lib
LINK_CMD=link.exe LIB_FLAGS=$OMConfigurationLinkSwitches
LINK_FLAGS=$OMConfigurationLinkSwitches $(SUBSYSTEM) / MACHINE:I386
```

**Generated Macros** The generated macros section of the makefile contains a variable that expands to the Rational Rhapsody-generated macros in the makefile. For example:

```
##### Generated macros
##### $OMContextMacros
OBJ_DIR=$OMObjectsDir !IF "$ (OBJ_DIR)" != "" CREATE_OBJ_DIR=if not exist $(OBJ_DIR) mkdir
$(OBJ_DIR) CLEAN_OBJ_DIR= if exist $(OBJ_DIR) $(RMDIR) $(OBJ_DIR) !ELSE
CREATE_OBJ_DIR= CLEAN_OBJ_DIR= !ENDIF
```

The \$OMContextMacros keyword expands several macros in the makefile. Each makefile macro has its own keyword. You can use these keywords separately to customize the makefile. The

\$OMContextMacros variable modifies target-specific variables. Replace the \$OMContextMacros line in the MakeFileContent property with the following:

```
FLAGSFILE=$OMFlagsFile
RULESFILE=$OMRulesFile OMROOT=$OMROOT C_EXT=$OMImplExt H_EXT=$OMSpecExt
OBJ_EXT=$OMObjExt EXE_EXT=$OMExeExt LIB_EXT=$OMLibExt
INSTRUMENTATION=$OMInstrumentation TIME_MODEL=$OMTimeModel
TARGET_TYPE=$OMTargetType TARGET_NAME=$OMTargetName $OMAllDependencyRule
TARGET_MAIN=$OMTargetMain LIBS=$OMLibs INCLUDE_PATH=$OMIncludePath
ADDITIONAL_OBJS=$OMAdditionalObjs OBJS= $OMObjs
```

**Predefined Macros** The predefined macros section of the makefile contains other macros than the Rational Rhapsody-generated macros specified in the generated macros section. For example, the default predefined macros section of a C++ makefile for the Microsoft environment is as follows:

```
##### Predefined macros #####
##### $(OBJS) : $(INST_LIBS)
$(OXF_LIBS) LIB_POSTFIX= !IF "$ (BuildSet)" == "Release" LIB_POSTFIX=R !ENDIF !IF
```

```

"$(TARGET_TYPE)" == "Executable" LinkDebug=$(LinkDebug) /DEBUG
LinkRelease=$(LinkRelease) /OPT:NOREF !ELSEIF "$(TARGET_TYPE)" == "Library"
LinkDebug=$(LinkDebug) /DEBUGTYPE:CV !ENDIF !IF "$(INSTRUMENTATION)" == "Animation"
INST_FLAGS=/D "OMANIMATOR" INST_INCLUDES=/I $(OMROOT)\LangCpp\om /I
$(OMROOT)\LangCpp\tom !IF "$(RPFrameWorkDll)" == "True" INST_LIBS=
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfanimdll$(LIB_POSTFIX)$(LIB_EXT)
!ELSE INST_LIBS= $(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomanim$(LIB_POSTFIX)$(LIB_EXT)
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfinst$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComApp$(LIB_POSTFIX)$(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)" == "Tracing" INST_FLAGS=/D
"OMTRACER" INST_INCLUDES=/I $(OMROOT)\LangCpp\om /I $(OMROOT)\LangCpp\tom !IF
"$(RPFrameWorkDll)" == "True" INST_LIBS= OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxftracedll$(LIB_POSTFIX)$(LIB_EXT) !ELSE
INST_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)tomtrace$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomtrace$(LIB_POSTFIX)$(LIB_EXT) OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfinst$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComApp$(LIB_POSTFIX)$(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)" == "None" INST_FLAGS=
INST_INCLUDES= INST_LIBS= !IF "$(RPFrameWorkDll)" == "True"
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfdll$(LIB_POSTFIX)$(LIB_EXT) !ELSE
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$(LIB_POSTFIX)$(LIB_EXT) !ENDIF
SOCK_LIB= !ELSE !ERROR An invalid Instrumentation $(INSTRUMENTATION) is specified.
!ENDIF

```

Generated Dependencies The generated dependencies section of the makefile contains a variable that expands to Rational Rhapsody-generated dependencies and compilation instructions. For example, the generated dependencies section of a C++ makefile for the Microsoft environment is as follows:

```

##### Generated dependencies #####
##### SOMContextDependencies
$OMFileObjPath : $OMMainImplementationFile $(OBJS) $(CPP) $(ConfigurationCPPCompileSwitches)
/Fo"$OMFileObjPath" $OMMainImplementationFile

```

Linking Instructions The linking instructions section of the makefile contains the predefined linking instructions. For example, the default linking instructions section of a C++ makefile for the Microsoft environment is as follows:

```

##### Linking instructions #####
#####
$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS) $OMFileObjPath
$OMMakefileName $OMModelLibs @echo Linking $(TARGET_NAME)$(EXE_EXT) $(LINK_CMD)
$OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \ $(OXF_LIBS) \
$(SOCK_LIB) \ $(LINK_FLAGS) /out:$(TARGET_NAME)$(EXE_EXT)
$(TARGET_NAME)$(LIB_EXT) : $(OBJS) $(ADDITIONAL_OBJS) $OMMakefileName @echo
Building library $@ $(LIB_CMD) $(LIB_FLAGS) /out:$(TARGET_NAME)$(LIB_EXT) $(OBJS)
$(ADDITIONAL_OBJS) clean: @echo Cleanup $OMCleanOBJS if exist $OMFileObjPath erase
$OMFileObjPath if exist *$(OBJ_EXT) erase *$(OBJ_EXT) if exist $(TARGET_NAME).pdb erase
$(TARGET_NAME).pdb if exist $(TARGET_NAME)$(LIB_EXT) erase
$(TARGET_NAME)$(LIB_EXT) if exist $(TARGET_NAME).ilk erase $(TARGET_NAME).ilk if exist
$(TARGET_NAME)$(EXE_EXT) erase $(TARGET_NAME)$(EXE_EXT) $(CLEAN_OBJ_DIR)

```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the C\_CG::<Environment>::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*Default = Blank*

## **NoneIncludeDirectories**

The NoneIncludeDirectories property is used to specify the directories that must be referenced in the makefile for projects where Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with INST\_INCLUDES.

*Default = Blank*

## **NoneInstLibs**

The NoneInstLibs property is used to specify the static libraries that are required when Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with INST\_LIBS.

*Default = Blank*

## **NoneOxfLibs**

The NoneOxfLibs property is used to specify the framework libraries that are required when Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with OXF\_LIBS.

*Default = \$(OMROOT)/LangC/lib/cygwinofx\$(CPU)\$(LIB\_EXT)*

## **NonePreprocessor**

The NonePreprocessor property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with INST\_FLAGS.

*Default = Blank*

## **ObjCleanCommand**

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default = \$(RM) \$OMFileObjPath*

## **ObjectName**

The `ObjectName` property specifies an alternative name for the compiled object file in the generated makefile.

*Default = empty string*

## ObjectsDirectory

The `ObjectsDirectory` property specifies an alternate name for the directory for compiled object files in the generated makefile. The default values are as follows:

Environment Default Value INTEGRITY work Integrity ESTL MultiWin32 obj\_dir All others Empty string

## ObjExtension

The `ObjExtension` property specifies the extension appended to compiled object files for a given environment.

*Default = .o*

## OSFileSystemCaseSensitive

The `OSFileSystemCaseSensitive` property specifies whether the OS file system for a given environment is case sensitive.

*Default = Checked*

## ParseErrorDescript

The `ParseErrorDescript` property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = ([^:]+)[:]/([0-9]+)[:] (error/warning)[:] (.\*)*

## ParseErrorMessage

The `ParseErrorMessage` property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning)

Along with the `ErrorMessageTokensFormat` property, `ParseErrorMessage` specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = ([^:]+)[:]/([0-9]+)[:]*

## ParseMakeError

The ParseMakeError property is used to define a regular expression that represents the format of make process or linker error messages. This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = (make)[:](.\*) (Error)*

## ParseSeverityError

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = [:] (error/undefined/cannot find/multiple definition)*

## ParseSeverityWarning

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^:]+)[:]( [0-9]+ )[:] (warning)*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation.

*Default = /*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Checked*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running the product, whereas the target is the machine running the application. To run remotely, the UseRemoteHost property must be set to True. If UseRemoteHost is True and RemoteHost is blank, the current host name is used for the remote host.

You can use this as a workaround if you have problems running animated applications on Windows 95.

The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = empty string*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

The extension ".h" is the default for C.

## TraceIncludeDirectories

The TraceIncludeDirectories property is used to specify the directories that must be referenced in the makefile for projects where Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with INST\_INCLUDES.

*Default = \$(INCLUDE\_QUALIFIER)\$(OMROOT)/LangC/aom  
\$(INCLUDE\_QUALIFIER)\$(OMROOT)/LangCpp/tom*

## TraceInstLibs

The TraceInstLibs property is used to specify the static libraries that are required when Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with INST\_LIBS.

*Default = \$(OMROOT)/LangCpp/lib/cygwintomtraceRiC\$(CPU)\$(LIB\_EXT)  
\$(OMROOT)/LangCpp/lib/cygwinomcomappl\$(CPU)\$(LIB\_EXT)  
\$(OMROOT)/LangCpp/lib/cygwinoxf\$(CPU)\$(LIB\_EXT)  
\$(OMROOT)/LangC/lib/cygwinaomtrace\$(CPU)\$(LIB\_EXT)*

## TraceOxfLibs

The TraceOxfLibs property is used to specify the framework libraries that are required when Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with OXF\_LIBS.

*Default = \$(OMROOT)/LangC/lib/cygwinoxfinst\$(CPU)\$(LIB\_EXT)  
\$(OMROOT)/LangC/lib/cygwinomcomappl\$(CPU)\$(LIB\_EXT)*

## TracePreprocessor

The TracePreprocessor property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with INST\_FLAGS.

*Default = \$(DEFINE\_QUALIFIER)OMTRACER*

## **UnixLineTerminationStyle**

The UnixLineTerminationStyle property specifies whether generated files use the UNIX end-of-line style.

If this property is set to Cleared, the end-of-line style depends on the host type (for example, DOS style on Windows machines, and UNIX style on Solaris machines).

*Default = Checked*

## **UnixPathNameForOMROOT**

The UnixPathNameForOMROOT property specifies whether the makefile must include UNIXstyle path names.

The pRISM compilers do not tolerate DOS-style path conventions. If you do not set this property correctly, there might be many compilation problems.

*Default = Cleared*

## **UpdateBuildSettingsInIDE**

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## **UseNewBuildOutputWindow**

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the CG::General::ShowLogViewAfterBuild property to True.

*Default = Checked*

## **UseRemoteHost**

The UseRemoteHost property specifies whether you intend to run an application remotely over a network

by default in a given environment.

*Default = Checked*

## UserVariables

The UserVariables property allows you to add user variables to be set in Eclipse integrated projects.

*Default = CYGWIN=nodosfilewarning*

## Dependency

The Dependency metaclass controls the dependency for a package that defines a namespace.

## CreateUseStatement

The CreateUseStatement property determines whether a use statement is added to the code after the with statement. The supplier of the dependency must be a class or type.

*Default = Cleared*

## GenerateForwardDeclarations

The GenerateForwardDeclarations property is a Boolean property that specifies whether forward declarations are generated.

When set to True, the generation of forward declarations is carried out according to the value of the property CG::Dependency::UsageType.

When set to False, the specification file will not contain a forward declaration even if the value of UsageType is set to Existence or Implementation.

*Default = True*

## GenerateOriginComment

When set to True, generates a comment before #include statements that indicate which element "caused" the #include.

## GeneratePragmaElaborate

*The GeneratePragmaElaborate property determines whether to generate an elaborate pragma for the*



*supplier class in the client class or package. Default = Cleared*

## **GeneratePragmaElaborateAll**

*The GeneratePragmaElaborateAll property determines whether to generate a pre-elaborate pragma for the supplier class in the client class or package. Default = Cleared*

## **GenerateWithClause**

The GenerateWithClause property determines whether with clauses are generated for Usage dependencies. For example, you can generate a with clause for a package, P1, in the specification of another package, P2, using a dependency, D1, and generate a use clause for P1 in the body of P2. In addition, this functionality is useful for modeling inherited annotations across classes and packages.

*Default = Checked*

## **ImplementationEpilog**

The ImplementationEpilog property enables you to add any code that you want to be added as verbatim text (to be ignored by Rational Rhapsody) to the end of the definition of a model element.. For example, you could wrap a section of code with an #ifdef-#endif pair, add compiler-specific keywords, or add a #pragma statement. For example, to specify that an operation is available only when the code is compiled with \_DEBUG, set the following properties for the operation:

- Set SpecificationProlog to #ifdef \_DEBUG cr.
- Set SpecificationEpilog to #endif.
- Set ImplementationProlog to #ifdef \_DEBUG cr.
- Set ImplementationEpilog to #endif.

The following table lists whether leading and trailing linefeeds are generated.

Metaclass   Leading   Linefeed   Added?   Class   Yes   Package   No

*Default = Empty MultiLine*

## **ImplementationProlog**

The ImplementationProlog property enables you to add any code that you want to be added as verbatim text (to be ignored by the software) to the beginning of the definition of a model element. For example, you could wrap a section of code with an #ifdef-#endif pair, add compiler-specific keywords, or add a #pragma statement. For example, to specify that an operation is available only when the code is compiled with \_DEBUG, set the following properties for the operation:

- Set SpecificationProlog to #ifdef \_DEBUG cr.
- Set SpecificationEpilog to #endif.
- Set ImplementationProlog to #ifdef \_DEBUG cr.

- Set ImplementationEpilog to #endif.

The following table lists whether leading and trailing linefeeds are generated.

Metaclass Trailing Linefeed Added? Class No Package Yes

*Default = Empty MultiLine*

## IncludeStyle

The IncludeStyle property controls the style of #include statements. When you use this property, you can control the style of a specific dependency, or the entire configuration/component/project. To set the style for include files that are synthesized based on associations between model elements (for example, setting the type of some attribute to a class), add a «Usage» dependency between the elements and set this property to the appropriate value. The possible values are as follows:

- Default - Use angle brackets for include statements for external elements, and quotes for include statements for other elements.
- Quotes - Enclose include files in quotation marks. For example: #include "A.h"
- When a compiler encounters an include file in quotes, it searches for the file in both the current directory and the directories specified in the include path. Note that the specific algorithm used is compiler-dependent.
- AngledBrackets - Enclose include files in angle brackets. For example: #include A.h
- When a compiler encounters an include file in angle brackets, it searches for the file only in the directories specified in the include path.
- If you set the property to AngledBrackets at the configuration level, you must also change the CG::File::IncludeScheme property to RelativeToConfiguration to ensure successful compilation.

*Default = Default*

## MarkPrologEpilogInAnnotations

The MarkPrologEpilogInAnnotations property specifies whether to generate ignore annotations for the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties so they are ignored during roundtrip. When you insert code element declarations (variables, types, functions, and so on) in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, after a full roundtrip those elements are added to the model and are duplicated on the next code generation.

When you use the MarkPrologEpilogInAnnotations property, you can have Rational Rhapsody automatically ignore the information specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties instead of adding the ignore annotations manually. The possible values for the MarkPrologEpilogInAnnotations property are:

- None - Rational Rhapsody does not generate any annotations. Any models created before Version 4.1 automatically have this property setting.
- Ignore - Rational Rhapsody generates the `/// ignore annotation before the code specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, and generates the /// annotation after the code specified in those properties.`

- **Auto** - If the code in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties is one line (it does not contain any newline characters (\n)), no annotations are generated (the same behavior has the None setting). If there is more than one line, Rational Rhapsody generates the `///  
[ ignore` annotation before the code specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, and generates the `///  
]` annotation after the code specified in those properties (the same behavior as the Ignore setting).

During roundtrip, any ignore annotations in the comments of the element are not included in its description. Because the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties are generated between the annotation and its declaration for the element, you cannot rename those elements on roundtrip. If you change the name of an element, it is removed from the model and added with the new name. Some model information (for example, property settings) might be lost.

*Default = Auto*

## Simplify

If you are using the Rational Rhapsody customizable code generation mechanism, the Simplify property can be used to change the way specific types of elements are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- **Default** - Uses the standard simplification for this item, as defined in Rational Rhapsody.
- **ByUser** - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- **ByUserPostDefault** - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for the element has been applied.

*Default = "Default"*

## SpecificationEpilog

The SpecificationEpilog property allows you to add code to the end of the declaration of a model element.

For example, you can use this property to add an `#ifdef` to indicate that an operation is available only when the code is compiled with `_DEBUG`, by setting the following properties:

- For SpecificationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For SpecificationEpilog, enter the value `#endif`
- For ImplementationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For ImplementationEpilog, enter the value `#endif`

*Default = Blank*

## SpecificationProlog

The SpecificationProlog property allows you to add code to the beginning of the declaration of a model element.

For example, you can use this property to add an `#ifdef` to indicate that an operation is available only when the code is compiled with `_DEBUG`, by setting the following properties:

- For SpecificationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For SpecificationEpilog, enter the value `#endif`
- For ImplementationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For ImplementationEpilog, enter the value `#endif`

*Default = Blank*

## UseNameSpace

The UseNameSpace property models namespace usage. When you set a dependency to a package that defines a namespace and set this property to True, Rational Rhapsody generates a “using namespace” statement to the package namespace.

*Default = Cleared*

## EnumerationLiteral

The EnumerationLiteral metaclass contains properties related to the generation of code for enumerations.

## DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- `$Name` - the element name
- `$FullName` - the full path of the element (P1::P2::C.a)
- `$Description` - the text provided on the Description tab of the Features window for the element

- **\$Requirements** - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the **\$Requirements** keyword in the **DescriptionTemplate** property to get such comments generated for specific model elements.
- **\$Login** - the login name of the user who generated the file
- **\$CodeGeneratedDate** - the date on which the code was generated
- **\$CodeGeneratedTime** - the time at which the code was generated
- **\$RhapsodyVersion** - the version of Rhapsody that was used to generate the file
- For operations: **\$Signature** - the operation signature, **\$Arguments** - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: **\$Type** - the type of the attribute
- For arguments: **\$Type** - the argument type, **\$Direction** - the argument direction (In, Out, or InOut)
- For events: **\$Arguments** - the text provided on the Description tab of the Features window for the event arguments
- For association ends: **\$Target** - the other end of the association
- For requirements: **\$ID** - the ID specified for the requirement, **\$Specification** - the specification text for the requirement
- The values of tags can be included by using the syntax **\$tag\_name**. For example, if you use a tag called **author** for your classes, you can use the keyword **\$author** in the value of **DescriptionTemplate** in order to have the value of the **author** tag included in the element description.
- The values of properties can be included by using the syntax **\$property\_name**. For example, you can use the keyword **\$Animate** in the value of **DescriptionTemplate** in order to have the value of the **Animate** property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as **\$Name**)
- Tag keywords
- Property keywords

If the value of a keyword is a **MultiLine** value, then each new line (except the first one) starts with the value of the `<lang>_CG::Configuration::DescriptionBeginLine` property, and each line ends with the value of the `<lang>_CG::Configuration::DescriptionEndLine` property.

*Default = Blank*

## Event

The Event metaclass contains properties that control events.

## AllocateMemory

The AllocateMemory property specifies the string generated to allocate memory dynamically for objects or events. This string is used in the Create() operation. The default memory allocation string is as follows: (\$cname \*) malloc(sizeof(\$cname)); The variable \$cname is replaced with the name of the object type during code generation. For example, the Create() operation generated for an object A uses this string to allocate memory for a new object as follows: A \* A\_Create(RiCTask \* p\_task) { A\* me = (A \*) malloc(sizeof(A)); A\_Init(me, p\_task); return me; } You can edit the memory allocation string to use a different mechanism than malloc(). The string used to free memory is specified with the FreeMemory property.

## AnimInstanceCreate

The AnimInstanceCreate property affects event creation. If you set the C\_CG::Event::NoDynamicAllocAnimCreate property to False, Rational Rhapsody does not generate the event creation method, effectively disabling the ability to inject the event in animation. To enable the injection of the event, you can specify a different method to obtain an instance of the event by setting this property to the name of the method to use.

*Default = empty string*

## DeclarationModifier

The DeclarationModifier property adds a string to the class or event declaration. The string is displayed between the class keyword and the class name in the generated code. For example, for a class A, the DeclarationModifier would appear as follows:

class DeclarationModifier> A {...}; This property adds a modifier to the class declaration. For example, if you have a class myExportableClass that is exported from a DLL by using the MYDLL\_API macro, you can set the DeclarationModifier property to “MYDLL\_API.” The generated code would then be as follows: class MYDLL\_API myExportableClass { ...}; This property supports two keywords: \$component and \$class.

*Default = empty string*

## DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, // Description: \$Description

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- `$Name` - the element name
- `$FullName` - the full path of the element (P1::P2::C.a)
- `$Description` - the text provided on the Description tab of the Features window for the element
- `$Requirements` - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the `$Requirements` keyword in the `DescriptionTemplate` property to get such comments generated for specific model elements.
- `$Login` - the login name of the user who generated the file
- `$CodeGeneratedDate` - the date on which the code was generated
- `$CodeGeneratedTime` - the time at which the code was generated
- `$RhapsodyVersion` - the version of Rhapsody that was used to generate the file
- For operations: `$Signature` - the operation signature, `$Arguments` - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: `$Type` - the type of the attribute
- For arguments: `$Type` - the argument type, `$Direction` - the argument direction (In, Out, or InOut)
- For events: `$Arguments` - the text provided on the Description tab of the Features window for the event arguments
- For association ends: `$Target` - the other end of the association
- For requirements: `$ID` - the ID specified for the requirement, `$Specification` - the specification text for the requirement
- The values of tags can be included by using the syntax `$tag_name`. For example, if you use a tag called `author` for your classes, you can use the keyword `$author` in the value of `DescriptionTemplate` in order to have the value of the `author` tag included in the element description.
- The values of properties can be included by using the syntax `$property_name`. For example, you can use the keyword `$Animate` in the value of `DescriptionTemplate` in order to have the value of the `Animate` property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as `$Name`)
- Tag keywords
- Property keywords

If the value of a keyword is a MultiLine value, then each new line (except the first one) starts with the value of the `<lang>_CG::Configuration::DescriptionBeginLine` property, and each line ends with the value of the `<lang>_CG::Configuration::DescriptionEndLine` property.

*Default = Blank*

## EnableDynamicAllocation

The EnableDynamicAllocation property specifies whether to use dynamic memory allocation for objects. The possible values are as follows:

- **Checked** - Dynamic allocation of events is enabled. Create() and Destroy() operations are generated for the object or object type.
- **Cleared** - Events are dynamically allocated during initialization, but not during run time. Create() and Destroy() operations are not generated for the object. This setting is recommended for static architectures that do not use dynamic memory management during run time.

If you are managing your own memory pools, set this property to False and call CPPReactive\_gen() directly. The following example shows how to call RiCReactive\_gen() directly to send a static event to a reactive object A, when using a member function of A genStaticEv2A():

```
void A_genStaticEv2A(struct A_t* const me) { { /*#[ operation genStaticEv2A() */ static struct ev _ev;  
ev_Init(_ev); RiCEvent_setDeleteAfterConsume(((RiCEvent*)_ev), RiCFALSE); (void)  
RiCReactive_gen(me-ric_reactive, ((RiCEvent*)_ev), RiCFALSE); /*#]*/ } }
```

Alternatively, you can use internal memory pools by setting the BaseNumberOfInstances property, which results in the use of framework memory pools. If you use the framework memory pools, do not disable the Create() and Destroy() methods because these methods are used to manage the memory pool. When you disable the generation of the Create() and Destroy() methods, you can still inject events in animation by supplying an alternate function to get an event instance.

To do this, set the AnimInstanceCreate property.

*Default = Checked*

## EventIdAsDefine

The EventIdAsDefine property enables you to use the event ID as define (rather than const short).

*Default = Checked*

## SerializationFunction

When sending events across address spaces, if the events to be sent include objects as arguments, you must provide functions for serializing and unserializing these objects.

The property SerializationFunction is used to specify the function that Rhapsody should use for serializing the arguments.

For the serialized functions, you must set: return value - pointer to the serialized event; ev - pointer of the event to be serialized; buffer - a local buffer that can be used for storing the serialized event (the user can allocate their own buffer instead); bufferSize - the size in bytes of the parameter buffer; resultBufferSize - pointer for storing the size of the returned serialized event.

For details regarding the required structure for the serialization function, see the User Guide.

*Default = Blank*



## UnserializationFunction

When sending events across address spaces, if the events to be sent include objects as arguments, you must provide functions for serializing and unserializing these objects.

The property UnserializationFunction is used to specify the function that Rhapsody should use for unserializing the arguments.

For unserialized functions, you must set: return value - pointer to the unserialized event; ev - pointer of the event to be serialized; serializedBuffer - pointer to the serialized buffer; serializedBufferSize - the size of the parameter serializedBuffer.

For details regarding the required structure for the unserialization function, see the User Guide.

*Default = Blank*

## FreeMemory

The FreeMemory property specifies the string generated to free memory previously allocated for objects or events. This string is used in the Destroy() operation. For an object, the free memory string is as follows: free(\$meName); The variable meName is replaced with the string used for the me context variable during code generation. For example, the Destroy() operation generated for an object A uses this string to free memory when an instance of A is destroyed as follows: void A\_Destroy(A\* const me) { A\_Cleanup(me); free(me); } You can edit the string used to free memory to use a different mechanism than free(). The string used to allocate memory is specified with the AllocateMemory property.

*Default = free(\$meName);*

## In

The In property determines the exact syntax used when an event is used as an "in" parameter for an operation.

*Default = const \$type\**

## InitCleanUpDestroyFunctionsAsPrivate

The InitCleanUpDestroyFunctionsAsPrivate property specifies that the cleanup destroy function of the event is to be a private operation.

*Default = Cleared*

## InOut

The InOut property determines the exact syntax used when an event is used as an "in/out" parameter for an operation.

*Default = \$type\**

## InstanceDeclaration

The InstanceDeclaration property specifies how instances are declared in code.

*Default = struct \$cname\$suffix*

In the generated code, the variable \$cname is replaced with the object (or object type) name. The variable \$suffix is replaced with the type suffix “\_t,” if the object is of implicit type.

## Out

The Out property determines the exact syntax used when an event is used as an "out" parameter for an operation.

*Default = \$type\*\**

## ReturnType

The ReturnType property determines the exact syntax used when an event is used as the return type of an operation.

*Default = \$type\**

## Simplify

If you are using the Rational Rhapsody customizable code generation mechanism, the Simplify property can be used to change the way specific types of elements are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - The element is ignored.
- Copy - The element is copied from the original to the simplified model. It is not modified in any way.
- Default - Uses the standard simplification for this item, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user.
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for the element has been applied.

*Default = "Default"*

## File

The File metaclass contains properties that control the generated code files.

## DiffDelimiter

When code is generated, new files are generated only where a before vs. after comparison indicates that the code has changed. When dealing with comments in the code, there are cases where you may decide that a change is not significant enough to justify regeneration of the file. For example, if you record the author of a file as a comment in the file, you may decide that the file should not be regenerated if the only change is the name of the author. The property DiffDelimiter can be used to mark such insignificant changes. When you use the string specified for the DiffDelimiter property somewhere in your code, the text to the right of the delimiter will be ignored when the code comparison is done prior to the regeneration of files.

For example, the default value of the property CPP\_CG::File::SpecificationHeader includes the following text:

```
//! Generated Date: $CodeGeneratedDate
```

So if the only change in the code is the code generation date, the file will not be regenerated.

Note that the delimiter can be used at the beginning of a line (in which case the entire line will be ignored in the comparison) or in the middle of a line (in which case only the text to the right of the delimiter will be ignored).

If you do not want to use this feature in your model, change the value of the property to blank.

*Default = //!*

## Footer

The Footer property specifies a multiline footer that is added to the end of generated Java files. The default footer template is as follows:

```
"/***** File Path:
$FullCodeGeneratedFileName *****/"
```

Footer format strings can contain any of the following keywords:

- \$ProjectName - The project name.
- \$ComponentName - The component name.
- \$ConfigurationName - The configuration name.
- \$ModelElementName - The name of the element mapped to the file. If there is more than one, this is the name of the first element.
- \$FullModelElementName - The name of the element mapped to the file, including the full path. If there is more than one, this is the name of the first element.
- \$CodeGeneratedDate - The generation date.
- \$CodeGeneratedTime - The generation time.

- `$RhapsodyVersion` - The version of Rational Rhapsody that generated the file.
- `$Login` - The user who generated the file.
- `$CodeGeneratedFileName` - The name of the generated file.
- `$FullCodeGeneratedFileName` - The full file name.
- `$Tag` - The value of the specified the element tag.
- `$Property` - The value of the element property with the specified name.

To avoid redundant compilation, Rational Rhapsody avoids unnecessary changes to specific lines prefixed with a special string, defined by the `C_CG::File::DiffDelimiter` property. The default `DiffDelimiter` value is `“//!”`.

## Generate

The `Generate` property is used to specify whether code should be generated for a Class or File element. For Java code generation, this is a Boolean property. For C and C++, there are also property values that can be used to generate only the specification file or only the implementation file.

The possible values are:

- `True` - for C and C++ both specification and implementation files are generated
- `False` - no files are generated
- `Specification (C, C++)` - only the specification file is generated
- `Implementation (C, C++)` - only the implementation file is generated

*Default = True*

## Header

The `Header` property specifies a multiline header that is added to the top of all generated Java files. The default header template is as follows:

```

/***** Rhapsody : $RhapsodyVersion
Login : $Login Component : $ComponentName Configuration : $ConfigurationName Model Element :
$FullModelElementName //! Generated Date : $CodeGeneratedDate File Path :
$FullCodeGeneratedFileName *****/

```

Header format strings can contain any of the following keywords:

- `$ProjectName` - The project name.
- `$ComponentName` - The component name.
- `$ConfigurationName` - The configuration name.
- `$ModelElementName` - The name of the element mapped to the file. If there is more than one, this is the name of the first element.
- `$FullModelElementName` - The name of the element mapped to the file, including the full path. If there is more than one, this is the name of the first element.
- `$CodeGeneratedDate` - The generation date.

- `$CodeGeneratedTime` - The generation time.
- `$RhapsodyVersion` - The version of Rational Rhapsody that generated the file.
- `$Login` - The user who generated the file.
- `$CodeGeneratedFileName` - The name of the generated file.
- `$FullCodeGeneratedFileName` - The full file name.
- `$Tag` - The value of the specified the element tag.
- `$Property` - The value of the element property with the specified name.

To avoid redundant compilation, Rational Rhapsody avoids unnecessary changes to specific lines prefixed with a special string, defined by the `C_CG::File::DiffDelimiter` property. The default `DiffDelimiter` value is `"//!"`.

## ImplementationEpilog

The `ImplementationEpilog` property enables you to add any code that you want to be added as verbatim text (to be ignored by Rational Rhapsody) to the end of the definition of a model element.. For example, you could wrap a section of code with an `#ifdef-#endif` pair, add compiler-specific keywords, or add a `#pragma` statement.

For example, to specify that an operation is available only when the code is compiled with `_DEBUG`, set the following properties for the operation:

- Set `SpecificationProlog` to `#ifdef _DEBUG cr.`
- Set `SpecificationEpilog` to `#endif.`
- Set `ImplementationProlog` to `#ifdef _DEBUG cr.`
- Set `ImplementationEpilog` to `#endif.`

The following table lists whether leading and trailing linefeeds are generated.

Metaclass Leading Linefeed Added? Class Yes Package No

*Default = Empty MultiLine*

## ImplementationFooter

The `ImplementationFooter` property specifies the multiline footer to be generated at the end of implementation files. The default footer template for C is as follows:

```

/***** File Path:
$FullCodeGeneratedFileName *****/

```

Footer format strings can contain any of the following keywords:

- `$ProjectName` - The project name.
- `$ComponentName` - The component name.
- `$ConfigurationName` - The configuration name.

- `$ModelElementName` - The name of the element mapped to the file. If there is more than one, this is the name of the first element.
- `$FullModelElementName` - The name of the element mapped to the file, including the full path. If there is more than one, this is the name of the first element.
- `$CodeGeneratedDate` - The generation date.
- `$CodeGeneratedTime` - The generation time.
- `$RhapsodyVersion` - The version of Rational Rhapsody that generated the file.
- `$Login` - The user who generated the file.
- `$CodeGeneratedFileName` - The name of the generated file.
- `$FullCodeGeneratedFileName` - The full file name.
- `$Tag` - The value of the specified the element tag.
- `$Property` - The value of the element property with the specified name.

To avoid redundant compilation, Rational Rhapsody avoids unnecessary changes to specific lines prefixed with a special string, defined by the `C.CG::File::DiffDelimiter` property. The default `DiffDelimiter` value is `“//!”`. The keywords are resolved in the following order:

- Predefined keywords
- Property keywords
- Tag keywords

Note the following:

Keyword names can be written in parentheses. For example: `$(Name)`

If the value of a keyword is a `MultiLine`, each new line (except the first one) starts with the value of the `C.CG::Configuration::DescriptionBeginLine` property; each line ends with the value of the `C.CG::Configuration::DescriptionEndLine` property.

## ImplementationHeader

The `ImplementationHeader` property specifies the multiline header that is generated at the beginning of implementation files. The default header template for C is as follows:

```
/***** Rhapsody: $RhapsodyVersion
Login: $Login Component: $ComponentName Configuration: $ConfigurationName Model Element:
$FullModelElementName //! Generated Date: $CodeGeneratedDate File Path:
$FullCodeGeneratedFileName *****/
```

Header format strings can contain any of the following keywords:

- `$ProjectName` - The project name.
- `$ComponentName` - The component name.
- `$ConfigurationName` - The configuration name.
- `$ModelElementName` - The name of the element mapped to the file. If there is more than one, this is the name of the first element.
- `$FullModelElementName` - The name of the element mapped to the file, including the full path. If there

is more than one, this is the name of the first element.

- `$CodeGeneratedDate` - The generation date.
- `$CodeGeneratedTime` - The generation time.
- `$RhapsodyVersion` - The version of Rational Rhapsody that generated the file.
- `$Login` - The user who generated the file.
- `$CodeGeneratedFileName` - The name of the generated file.
- `$FullCodeGeneratedFileName` - The full file name.
- `$Tag` - The value of the specified the element tag.
- `$Property` - The value of the element property with the specified name.

To avoid redundant compilation, Rational Rhapsody avoids unnecessary changes to specific lines prefixed with a special string, defined by the `C_CG::File::DiffDelimiter` property. The default `DiffDelimiter` value is `“//!”`. The keywords are resolved in the following order:

- Predefined keywords
- Property keywords
- Tag keywords

Note the following:

- Keyword names can be written in parentheses. For example: `$(Name)`
- If the value of a keyword is a `MultiLine`, each new line (except the first one) starts with the value of the `C_CG::Configuration::DescriptionBeginLine` property; each line ends with the value of the `C_CG::Configuration::DescriptionEndLine` property.

## ImplementationProlog

The `ImplementationProlog` property enables you to add any code that you want to be added as verbatim text (to be ignored by the software) to the beginning of the definition of a model element. For example, you could wrap a section of code with an `#ifdef-#endif` pair, add compiler-specific keywords, or add a `#pragma` statement.

For example, to specify that an operation is available only when the code is compiled with `_DEBUG`, set the following properties for the operation:

- Set `SpecificationProlog` to `#ifdef _DEBUG cr.`
- Set `SpecificationEpilog` to `#endif.`
- Set `ImplementationProlog` to `#ifdef _DEBUG cr.`
- Set `ImplementationEpilog` to `#endif.`

The following table lists whether leading and trailing linefeeds are generated.

Metaclass Trailing Linefeed Added? Class No Package Yes

*Default = Empty MultiLine*

## MarkPrologEpilogInAnnotations

The `MarkPrologEpilogInAnnotations` property specifies whether to generate ignore annotations for the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties so they are ignored during roundtrip. When you insert code element declarations (variables, types, functions, and so on) in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, after a full roundtrip those elements are added to the model and are duplicated on the next code generation.

When you use the `MarkPrologEpilogInAnnotations` property, you can have Rational Rhapsody automatically ignore the information specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties instead of adding the ignore annotations manually. The possible values for the `MarkPrologEpilogInAnnotations` property are:

- **None** - Rational Rhapsody does not generate any annotations. Any models created before Version 4.1 automatically have this property setting.
- **Ignore** - Rational Rhapsody generates the `///  
[ ignore annotation before the code specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, and generates the ///  
] annotation after the code specified in those properties.`
- **Auto** - If the code in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties is one line (it does not contain any newline characters (`\n`)), no annotations are generated (the same behavior has the **None** setting). If there is more than one line, Rational Rhapsody generates the `///  
[ ignore annotation before the code specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, and generates the ///  
] annotation after the code specified in those properties (the same behavior as the Ignore setting).`

During roundtrip, any ignore annotations in the comments of the element are not included in its description. Because the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties are generated between the annotation and its declaration for the element, you cannot rename those elements on roundtrip. If you change the name of an element, it is removed from the model and added with the new name. Some model information (for example, property settings) might be lost.

*Default = Auto*

## Simplify

If you are using the Rational Rhapsody customizable code generation mechanism, the `Simplify` property can be used to change the way specific types of elements are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- **None** - The element is ignored.
- **Copy** - The element is copied from the original to the simplified model. It is not modified in any way.
- **Default** - Uses the standard simplification for this item, as defined in Rational Rhapsody.
- **ByUser** - Uses the customized simplification provided by the user.
- **ByUserPostDefault** - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for the element has been applied.

Note that this property refers to the simplification of component files.



*Default = "Default"*

## SpecificationEpilog

The SpecificationEpilog property allows you to add code to the end of the declaration of a model element.

For example, you can use this property to add an `#ifdef` to indicate that an operation is available only when the code is compiled with `_DEBUG`, by setting the following properties:

- For SpecificationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For SpecificationEpilog, enter the value `#endif`
- For ImplementationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For ImplementationEpilog, enter the value `#endif`

*Default = Blank*

## SpecificationFooter

The SpecificationFooter property specifies the multiline footer to be generated at the end of specification files. The default footer template for C is as follows:

```
/***** File Path:
$FullCodeGeneratedFileName *****/
```

Footer format strings can contain any of the following keywords:

- `$ProjectName` - The project name.
- `$ComponentName` - The component name.
- `$ConfigurationName` - The configuration name.
- `$ModelElementName` - The name of the element mapped to the file. If there is more than one, this is the name of the first element.
- `$FullModelElementName` - The name of the element mapped to the file, including the full path. If there is more than one, this is the name of the first element.
- `$CodeGeneratedDate` - The generation date.
- `$CodeGeneratedTime` - The generation time.
- `$RhapsodyVersion` - The version of Rational Rhapsody that generated the file.
- `$Login` - The user who generated the file.
- `$CodeGeneratedFileName` - The name of the generated file.
- `$FullCodeGeneratedFileName` - The full file name.
- `$Tag` - The value of the specified the element tag.
- `$Property` - The value of the element property with the specified name.

To avoid redundant compilation, Rational Rhapsody avoids unnecessary changes to specific lines prefixed with a special string, defined by the `C_CG::File::DiffDelimiter` property. The default `DiffDelimiter` value is `"//!"`. The keywords are resolved in the following order:

- Predefined keywords (such as \$Name)
- Property keywords
- Tag keywords

Note the following:

- Keyword names can be written in parentheses. For example: \$(Name)
- If the value of a keyword is a MultiLine, each new line (except the first one) starts with the value of the C\_CG::Configuration::DescriptionBeginLine property; each line ends with the value of the C\_CG::Configuration::DescriptionEndLine property.

## SpecificationHeader

The SpecificationHeader property specifies the multiline header to be generated at the beginning of specification files. The default header template for C is as follows:

```
/***** Rhapsody: $RhapsodyVersion
Login: $Login Component: $ComponentName Configuration: $ConfigurationName Model Element:
$FullModelElementName /*! Generated Date: $CodeGeneratedDate File Path:
$FullCodeGeneratedFileName *****/
```

Header format strings can contain any of the following keywords:

- \$ProjectName - The project name.
- \$ComponentName - The component name.
- \$ConfigurationName - The configuration name.
- \$ModelElementName - The name of the element mapped to the file. If there is more than one, this is the name of the first element.
- \$FullModelElementName - The name of the element mapped to the file, including the full path. If there is more than one, this is the name of the first element.
- \$CodeGeneratedDate - The generation date.
- \$CodeGeneratedTime - The generation time.
- \$RhapsodyVersion - The version of Rational Rhapsody that generated the file.
- \$Login - The user who generated the file.
- \$CodeGeneratedFileName - The name of the generated file.
- \$FullCodeGeneratedFileName - The full file name.
- \$Tag - The value of the specified the element tag.
- \$Property - The value of the element property with the specified name.

To avoid redundant compilation, Rational Rhapsody avoids unnecessary changes to specific lines prefixed with a special string, defined by the C\_CG::File::DiffDelimiter property. The default DiffDelimiter value is “/\*!”. The keywords are resolved in the following order:

- Predefined keywords (such as \$Name)
- Property keywords
- Tag keywords

Note the following:

- Keyword names can be written in parentheses. For example: \$(Name)
- If the value of a keyword is a MultiLine, each new line (except the first one) starts with the value of the C\_CG::Configuration::DescriptionBeginLine property; each line ends with the value of the C\_CG::Configuration::DescriptionEndLine property.

## SpecificationProlog

The SpecificationProlog property adds code to the beginning of the declaration of a model element (such as a configuration or class). For example, to create an abstract class in Java, you can set the SpecificationProlog property for the class to “abstract.” You must include the space after the word “abstract.” If the visibility for the class is set to default, the following class declaration is generated in the .java file: abstract class classname {...} The SpecificationProlog property allows you to add compiler-specific keywords, add a #pragma statement, or wrap a section of code with an #ifdef-#endif pair. For example, to specify that an operation is available only when the code is compiled with \_DEBUG, set the following properties for the operation:

- Set SpecificationProlog to #ifdef \_DEBUG cr.
- Set SpecificationEpilog to #endif.
- Set ImplementationProlog to #ifdef \_DEBUG cr.
- Set ImplementationEpilog to #endif.

The following table lists whether leading and trailing linefeeds are generated.

Metaclass	Leading Linefeed Added?	Trailing Linefeed Added?	Class	Yes	No	Package	Yes	Yes
-----------	-------------------------	--------------------------	-------	-----	----	---------	-----	-----

*Default = Empty MultiLine*

## flowPort

The flowPort metaclass controls whether code is generated for flowports.

## InvokeRelay

Use the InvokeRelay property to control the relay of data from flowports.

This possible values are:

- UponValueChange - Data is sent through the flowport only if a change from the previous data happens.
- Always - Data is always sent through the flowport.

*Default = UponValueChange*

## OptimizeCode

Code generation for ports and flow ports was optimized in version 7.5.3 of Rhapsody, relative to the code generated in previous versions. A new property named `OptimizeCode` was added with a default value of `True`. If the value of this property is set to `False`, the old code generation mechanism will be used for flow ports.

## ReceiveRelay

Use the `ReceiveRelay` property to control the notification event (the 'chXXX' event) called when data has arrived through a flowport.

This possible values are:

- `UponValueChange` - The notification event is called only if a change of data happens.
- `Always` - Notification event is called even if no change of data occurs.

*Default = UponValueChange*

## SupportMulticast

Use this property to control the multicasting ability of a flowport. Use the property to enable data or events to be sent from one sender flowport to many.

The possible values are:

- `Always` - Rational Rhapsody always generates multicasting ability to each flowport in the model.
- `Smart` - Rational Rhapsody identifies flowports that are connected to more than 1 flowport and generates code to support multicasting to those flowports only.
- `Never` - Rational Rhapsody never generates code supporting multicasting of data/event through flowports. This is the value for models created before Rational Rhapsody 7.5, so the old behavior is the same.

*Default = Smart*

## Framework

The `Framework` metaclass contains properties that affect the Rational Rhapsody framework.

### ActivateFrameworkDefaultEventLoop

The `ActivateFrameworkDefaultEventLoop` property specifies the framework call that initializes the framework main event loop.

*Default = OXF::start(\$Fork);*

The value of `$Fork` is calculated from the `CG::Configuration::StartFrameworkInMainThread` property for

regular applications and from the CORBA::Configuration::StartFrameworkInMainThread property for CORBA servers. This property can be set at the configuration level or higher.

## ActiveBase

The ActiveBase property specifies the superclass from which to specialize all threads, if the ActiveBaseUsage property is set to True.

*Default = RiCTask*

## ActiveBaseUsage

The ActiveBaseUsage property specifies whether to use the superclass specified by the ActiveBase property as the superclass for all threads.

*Default = Checked*

## ActiveDestructorGuard

The ActiveDestructorGuard property specifies the macro that starts protection for an active user object destructor.

*Default = START\_DTOR\_THREAD\_GUARDED\_SECTION*

## ActiveExecuteOperationName

The ActiveExecuteOperationName property sets the user object virtual table for an active object and passes it to a task in the task initialization function (RiCTask\_init()). Follow these steps:

- Create a method with the following signature: struct RiCReactive \* operation name> (RiCTask \* const)
- Set the operation name in the ActiveExecuteOperationName property.
- Start the execution of the active object task by calling the RICTASK\_START() macro on the object.

The virtual function table member name is stored in the ActiveVtblName property.

*Default = empty string*

## ActiveGuardInitialization

The ActiveGuardInitialization property specifies the call that makes the active object event dispatching guarded.

*Default = SetToGuardThread*

## ActiveIncludeFiles

The ActiveIncludeFiles property specifies the base class for threads when you use selective framework includes. If a class is active and this property is defined, the file specified by the property is included in the class specification file.

The default value for C is oxf/RiCTask.h.

## ActiveInit

The ActiveInit property specifies the format of the declaration generated for the initializer for an active class.

The default value for C is \$base\_init(\$member, RiCFALSE, \$Vtbl) .

## ActiveInitDistributed

The ActiveInitDistributed property specifies the code generated by Rational Rhapsody for the initializer of an active class in applications that send events across address spaces.

*Default = \$base\_initDistributed(\$member, RiCFALSE, \$Vtbl, "\$PublishedName");*

## ActiveMessageQueueSize

The ActiveMessageQueueSize property specifies the size of the message queue allocated for active objects, if the ActiveMessageQueueSize property for classes is left blank. The default value for the size of the message queue is language-dependent, as follows:

- C - Default = RiCOSDefaultMessageQueueSize, which is a variable set in the implementation file of the OS adapter for a given operating system.
- C++ - Default = OMOSThread::DefaultMessageQueueSize.
- Java - Default = an empty string (blank).

## ActiveStackSize

The ActiveStackSize property specifies the size of the stack allocated for active objects, if the ActiveStackSize property for classes is left blank. The default value for the stack size is language-dependent, as follows:

- C - Default = RiCOSDefaultStackSize, which is a variable set in the implementation file of the OS adapter for a given operating system.
- C++ - Default = OMOSThread::DefaultStackSize.
- Java - Default = an empty string (blank).

## ActiveThreadName

The `ActiveThreadName` property specifies the name of threads, if the `ActiveThreadName` property for classes is left blank. The default values are as follows:

- A string - Names the active thread.
- `NULL` - The value is set in an operating system-specific manner, based on the value of the `ActiveThreadName` property for the framework.

*Default = `NULL`*

## ActiveThreadPriority

The `ActiveThreadPriority` property specifies the priority of threads, if the `ActiveThreadPriority` property for classes is left blank. The default value for the priority of threads is language-dependent, as follows:

- C - Default = `RiCOSDefaultThreadPriority`, which is a variable set in the implementation file of the OS adapter for a given operating system.
- C++ - Default = `OMOSThread::DefaultThreadPriority`.
- Java - Default = an empty string (blank).

## ActiveVtblName

The `ActiveVtblName` property stores the name of the virtual function table that is associated with a task (the `RiCTask` member of the structure).

*Default = `$ObjectName_activeVtbl`*

## BooleanType

The `BooleanType` property specifies the Boolean type used by the framework.

*Default = `bool`*

## CurrentEventId

The `CurrentEventId` property specifies the call or macro used to obtain the ID of the currently consumed event.

*Default = `OM_CURRENT_EVENT_ID`*

## DefaultProvidedInterfaceName

The `DefaultProvidedInterfaceName` property specifies the interface that must be implemented by the "in" part of a rapid port. See the Rational Rhapsody help for more information on rapid ports.

*Default = `DefaultProvidedInterface`*

## DefaultReactivePortBase

The DefaultReactivePortBase property stores the base class for the generic rapid port (or default reactive port). This base class relays all events. See the Rational Rhapsody help for more information on rapid ports.

*Default = RiCDefaultReactivePort*

## DefaultReactivePortIncludeFiles

The DefaultReactivePortIncludeFiles property specifies the include files that are referenced in the generated file that implements the class with the rapid ports. See the Rational Rhapsody help for more information on rapid ports.

*Default = oxf/OMDefaultReactivePort.h*

## DefaultRequiredInterfaceName

The DefaultRequiredInterfaceName property specifies the interface that must be implemented by the "out" part of a rapid port. See the Rational Rhapsody help for more information on rapid ports.

*Default = DefaultRequiredInterface*

## doxHeaderFile

This property controls the header file from the dox framework subsystem in C. The dox subsystem supports "Send Event Across Address Spaces" feature in Rational Rhapsody Developer for C. In several locations where auto-generated code (by the Rational Rhapsody code generator) relies on API from dox subsystem, this header file should be included.

## EmptyClassDeclaration

When Rational Rhapsody generates C code, classes in the model are generated as structs. Some compilers will not tolerate empty structs. To overcome this problem, the EmptyClassDeclaration property provides a dummy struct member that is included in the code if the struct would be empty otherwise.

*Default = RIC\_EMPTY\_STRUCT (This is the name of a macro defined in the framework code.)*

## EnableDirectReactiveDeletion

The EnableDirectReactiveDeletion property specifies the call to the framework that supports direct deletion of reactive instances (by using the delete operator) instead of graceful framework termination (by using the reactive destroy() method). When you use destroy(), the object waits in a zombie mode until all the events that are designated to it are removed from the active context queue, and then self -destructs.



In this scheme, there is no need to traverse the queue of the active context to cancel pending events, and there is no need to make the reactive destructor guarded to ensure safe deletion. A reactive object can be either in a graceful termination or forced deletion (by using the delete operator) state: you cannot use graceful deletion on an object that allows forced deletion, and vice versa. You can set a single reactive object in a forced deletion state, or set the entire system (all reactive instances) in a forced deletion state (as is done for compatibility with earlier versions).

Graceful termination should not be used when a reactive part (of a composite class) runs in a context of an active object that is not part of, and different from, the composite active context. If you are using a Rational Rhapsody library component as part of an application where the main is not generated by Rational Rhapsody (for example, GUI applications), the framework initializes itself in full compatibility mode on the call to `OXF::init()`.

If you want to remove part or all of the compatibility features, call `OXF::initialize()` instead of `OXF::init()` (the operation takes the same arguments) and add independent activation calls that are compatible with earlier versions, prior to the `initialize()` call. Note that the `C_CG::Framework::UseDirectReactiveDeletion` property must be set to True for this property to take effect. When it is set to True, the code specified in the `EnableDirectReactiveDeletion` is generated in the main prior to the call to `OXF::init()`.

*Default = OXF::supportExplicitReactiveDeletion();*

## EventBase

The `EventBase` property specifies the base class for all events.

*Default = RiCEvent*

## EventBaseUsage

The `EventBaseUsage` property specifies whether to use the event superclass specified by the `EventBase` property as the parent of all events.

The C default value is Checked.

## EventGenerationPattern

The `EventGenerationPattern` property supplies some of the information needed to generate code for Send Action elements in C, C++, and Java.

For each language, code generation for this element is determined by the following properties:

- `C_CG::Framework::EventGenerationPattern` - general format
- `C_CG::Framework::EventToPortGenerationPattern` - used when sending even to a port

Note:

Rational Rhapsody does not support roundtripping for Send Action elements.

*Default = RiCGEN(\$meArrow\$target, \$event)*

## EventIdType

When Rhapsody generates code for an event, it creates an ID number for the event. The property EventIDType allows you to specify the type that should be used for this number if you do not want to use the type that Rhapsody generates by default. In C and C++, the value of this property affects code generation only if the property EventIdAsDefine is set to False.

*Default = short*

## EventIncludeFiles

The EventIncludeFiles property specifies the base class for events when you use selective framework includes. If events are defined in a package, the file specified by this property is included in the package specification file to enable the use of events and timeouts in the package.

*Default = <oxf/RiCEvent.h>*

## EventSetDestroyOp

The EventSetDestroyOp property is used to specify the operation that should be used to destroy events. This code is generated as part of the event's \_Init function.

*Default = RIC\_SET\_EVENT\_DESTROY\_OP(\$me, \$Event);*

## EventSetParamsStatement

The EventSetParamsStatement property specifies a template for the body of the setParams() method, provided by the Rational Rhapsody framework for Java, to set the parameters of an event. For example, for an event of type evOn(), the default template would generate the following code in the body of the setParams() method: evOn params = (evOn) event; The default value is as follows: \$eventType params = (\$eventType) event;

## EventToPortWithInterfaceGenerationPattern

The EventToPortWithInterfaceGenerationPattern property specifies the macro that should be generated for sending an event via a port with interfaces that have event receptions. For rapid ports, the macro to generate is specified by the EventToPortGenerationPattern property.

*Default = RiCGEN\_PORT\_I(\$class, \$target, \$interface, \$event)*

## FrameworkGlobalVariableName

For projects that use the MicroC profile, the property FrameworkGlobalVariableName can be used to customize the name that is generated for the framework's global variable.

*Default = mxfglobals*

## FrameworkInitialization

The FrameworkInitialization property specifies the framework initialization code that is called by the main. The default value is as follows: `OXF::initialize$(Argc)$(Argv)$(AnimationPortNumber)$(RemoteHost)$(TimerResolution)$(TimerMaxTimeouts)$(TimeModel))`

## HeaderFile

The HeaderFile property specifies the framework header files to be included in objects that are within the scope of a particular configuration. The default values are as follows:

Default Generated Statement "oxf/Ric.h" `#include oxf/Ric.h`

To optimize your code for size, leave the HeaderFile property blank. In this way, you can explicitly include the framework only when needed.

## IncludeHeaderFile

The IncludeHeaderFile property specifies whether to include the framework header files specified by the `CG::Framework::HeaderFile` property in the project.

The C default value is Checked.

## InnerReactiveClassName

The InnerReactiveInstanceName property specifies the name of a reactive class that serves as a bridge between a reactive class in your model and the framework. The implementation scheme of reactive classes is different in Java than in C++. Java does not allow inheritance from the reactive framework classes because that would mean that you would not be able to inherit from an additional base class that might not be reactive.

The chosen alternative is to delegate an inner class instance that inherits from `RiJStateReactive`. Delegation is the implementation of an interface that forwards relevant messages to the inner class instance.

*Default = Reactive*

## InnerReactiveInstanceName

The InnerReactiveInstanceName property specifies the name of a reactive instance that serves as a bridge between a reactive class in your model and the framework. The implementation scheme of reactive classes is different in Java than in C++. Java does not allow inheritance from the reactive framework classes because that would mean that you would not be able to inherit from an additional base class that might not be reactive.

The chosen alternative is to delegate an inner class instance that inherits from `RiJStateReactive`. Delegation is the implementation of an interface that forwards relevant messages to the inner class instance.

*Default = reactive*

## InstrumentVtblName

The `InstrumentVtblName` property specifies the name of the virtual function table that is associated with animation objects. Each animated object has its own virtual function table (Vtbl). This table creates your own framework, with its own virtual instrumentation functions, and connect it to Rational Rhapsody.

*Default = \$ObjectName\_instrumentVtbl*

## IsCompletedCall

The `IsCompletedCall` property specifies the call or macro that determines whether the state reached a final state so it can be exited on a null transition. The property supports the `$State` keyword so you can use state-based calls. The keyword is resolved to the state implementation (code) name.

*Default = IS\_COMPLETED(\$State)*

## IsInCall

The `IsInCall` property specifies the query that determines whether the state is in the current active configuration. The property supports the `$State` keyword so you can use state-based calls. The keyword is resolved to the state implementation (code) name.

*Default = IS\_IN(\$State)*

## MakeFileName

The `MakeFileName` property specifies a new name for the makefile. To use this property, add the following line to the .prp file:

Property MakeFileName String "MyFileName"

In this syntax, `MyFileName` specifies the name of the makefile.

## NullTransitionId

The `NullTransitionId` property specifies the ID reserved for null transition consumption.

*Default = OMEventNullId*

## OperationGuard

The OperationGuard property specifies the macro that guards an operation.

*Default = GUARD\_OPERATION*

## ProtectedBase

The ProtectedBase property specifies the base class for protected objects, if the ProtectedBaseUsage property is set to True.

The C default value is RiCMonitor.

## ProtectedBaseUsage

The ProtectedBaseUsage property specifies whether to use the class specified by the ProtectedBase property as the base class for protected objects.

The C default value is Checked.

## ProtectedClassDeclaration

The ProtectedClassDeclaration property affects how protected classes are implemented. Beginning with Rational Rhapsody 4.0, instead of inheriting from OMProtected, the class embeds an aggregate OMProtected. The aggregate member and helper methods are defined in the macro OMDECLARE\_GUARDED (defined in omprotected.h).

*Default = OMDECLARE\_GUARDED*

## ProtectedIncludeFiles

The ProtectedIncludeFiles property specifies the base class for protected classes when you use selective framework includes. The default value for C is as follows: oxf/RiCProtected.h

## ProtectedInit

The ProtectedInit property specifies the declaration generated for the initializer for guarded objects.

The default value for C is \$base\_init(\$member).

## ReactiveBase

The ReactiveBase property specifies the base class for all reactive classes, if the ReactiveBaseUsage

property is set to True.

## ReactiveBaseUsage

The ReactiveBaseUsage property specifies whether to use the class specified by the ReactiveBase property as the base class for all reactive objects.

*Default = Checked.*

## ReactiveConsumeEventOperationName

The ReactiveConsumeEventOperationName property sets the user object virtual table for a reactive object. Follow these steps:

- Create a method with the following signature: void operation name>(RiCReactive \* const, RiCEvent\*)
- Set the operation name in the ReactiveConsumeEventOperationName property.

Rational Rhapsody Developer for Ada ignores all the values for the properties under the Framework metaclass except for this one.

*Default = empty string*

## ReactiveCtorActiveArgDefaultValue

The ReactiveCtorActiveArgDefaultValue property specifies the default value of the active context argument in a reactive constructor.

*Default = 0*

## ReactiveCtorActiveArgName

The ReactiveCtorActiveArgDefaultValue property specifies the name of the active context argument in a reactive constructor.

*Default = activeContext*

## ReactiveCtorActiveArgType

The ReactiveCtorActiveArgDefaultValue property specifies the type of the active context argument in a reactive constructor.

*Default = IOxfActive\**

## ReactiveDestructorGuard

The `ReactiveDestructorGuard` property specifies the macro that starts protection of a section of code used for destruction of a reactive instance. This prevents a “race” (between the deletion and event dispatching) when deleting an active instance.

*Default = `START_DTOR_REACTIVE_GUARDED_SECTION`*

## ReactiveEnableAccessEventData

The `ReactiveEnableAccessEventData` property specifies the code to be used to enable access to the specific event data in a transition (typically by assigning a local variable of the appropriate type). The property supports the `$Event` keyword so you can specify the event type.

*Default = `RiCSETPARAMS($me, $Event);`*

## ReactiveGetStateCall

The `ReactiveGetStateCall` property is used for serialization to define the prototype of the `getState` framework method.

*Default = `RiCReactive_getState(&(me->ric_reactive));`*

## ReactiveGuardInitialization

*The `ReactiveDestructorGuard` property specifies the framework call that makes the event consumption of a specific reactive class guarded. (Default = `setToGuardReactive`)*

## ReactiveHandleEventNotConsumed

The `ReactiveHandleEventNotConsumed` property registers a method to handle unconsumed events in a reactive class. Specify the method name as the value for this property.

*Default = empty string*

## ReactiveHandleTONotConsumed

The `ReactiveHandleTONotConsumed` property registers a method to handle unconsumed trigger operations in a reactive class. Specify the method name as the value for this property.

*Default = empty string*

## ReactiveIncludeFiles

The `ReactiveIncludeFiles` property specifies the base classes for reactive classes when you use selective framework includes. If a class is reactive and this property is defined, the file specified by the property is included in the class specification file. For reactive classes, the header files specified by the following

properties are also included:

- EventIncludeFiles - For the event base class
- ActiveIncludeFiles - If the class is guarded or instrumented

*Default = oxf/RiCReactive.h*

## ReactiveInit

The ReactiveInit property specifies the declaration for the initializer generated for reactive objects.

The default pattern for C is as follows: \$base\_init(\$member, (void\*)\$mePtr, \$task, \$VtblName);

The \$base variable is replaced with the name of the reactive object during code generation.

The string “\_init” is appended to the object name in the name of the operation. For example, if the reactive object is named A, the initializer generated for A is named A\_init(). The \$member variable is replaced with the name of the reactive member (equivalent to the base class) of the object during code generation.

The \$mePtr variable is replaced with the name of the user object (the value of the Me property). The member and mePtr objects are not equivalent if the user object is active.

The \$VtblName variable is replaced with the name of the virtual function table for an object, specified by the ReactiveVtblName property.

## ReactiveInterface

The ReactiveInterface property specifies the name of the interface class that forwards messages to an inner class instance of a reactive class in order to implement its reactive behavior.

*Default = RiJStateConcept*

## ReactiveSetEventHandlingGuard

The ReactiveSetEventHandlingGuard property controls the code that is generated within the constructor of a reactive class. When you use this property with guarded triggered operations, it enables guarding of the event handling (in order to provide mutual exclusion between the event and TO handling).

*Default = setEventGuard(getGuard());*

## ReactiveInterfaceBase

The OXF framework contains an interface called IRiCReactive which is realized by reactive classes such as RiCReactive. If you have renamed this interface or created a different interface that serves this function, use the ReactiveInterfaceBase property to specify the name of the interface.

*Default = IRiCReactive*



## ReactiveSetStateCall

The ReactiveGetStateCall property is used for serialization to define the prototype of the setState framework method.

The C Default is `RiCReactive_setState(&(me->ric_reactive), oxfReactiveState);`.

## ReactiveSetTask

The ReactiveSetTask property specifies the string that tells a reactive object whether it is an active or a sequential instance. The default value for Ada is an empty string. The default value for C is as follows: `RiCReactive_setActive($member, $isActive);`

## ReactiveStateType

The ReactiveStateType property is used for serialization to define the oxfstate type.

*Default = long*

## ReactiveVtblName

The ReactiveVtblName property specifies the name of the virtual function table (Vtbl) associated with a reactive object. Each reactive object has its own Vtbl, which creates your own framework and connect it to Rational Rhapsody.

*Default = \$ObjectName\_reactiveVtbl*

## RegisterAddressSpaceSerialize

The RegisterAddressSpaceSerialize property specifies the code generated by Rational Rhapsody in order to register cross-address-space serialization and unserialization functions for events. You can specify the serialization and unserialization functions using the properties `C::Event::SerializationFunction` and `C::Event::UnserializationFunction`.

*Default = RiDRegisterEventSerializationInfo(\$eventId, \$serializeOp, \$unserializeOp);*

## SetAddressSpaceName

The SetAddressSpaceName property specifies the code generated by Rational Rhapsody to set the address space name for the application. This property is used for applications that send events across address spaces.

*Default = RiCSetAddressSpaceName(\$AddressSpaceName);*

## SetManagedTimeoutCanceling

The SetManagedTimeoutCanceling property is a property for compatibility with previous versions that specifies whether the framework uses the pre-Rational Rhapsody 6.0 scheme of timeout creation and cancellation (where OMTimerManager is responsible for cancellation of timeouts) or the Rational Rhapsody scheme. The framework moves the responsibility for a timeout cancellation from the timer manager to the timeout client (the reactive object).

This change reduces the timer manager responsibilities and the overhead in timeout management (thus improving timeout scheduling performance). The change also includes changes in the generated code (the user reactive objects hold pointers to the waiting timeouts in order to enable canceling).

If you are using a Rational Rhapsody library component as part of an application where the main is not generated by Rational Rhapsody (for example, GUI applications), the framework initializes itself in full compatibility mode on the call to OXF::init(). If you want to remove part or all of the compatibility features, call OXF::initialize() instead of OXF::init() (the operation takes the same arguments) and add independent activation calls that are compatible with previous versions, prior to the initialize() call.

*Default = OXF::setManagedTimeoutCanceling(true);*

## SetRhp5CompatibilityAPI

*The SetRhp5CompatibilityAPI property specifies the call that configures models created before Rational Rhapsody 6.0 so they use the 5. x version of the framework instead of the new one. You might also want to use the "Filter" facility in this window to refer to the definition of UseRhp5CompatibilityAPI for more information on Version 5. x compatibility mode. (Default = OXF::setRhp5CompatibleAPI(true);)*

## StaticMemoryIncludeFiles

The StaticMemoryIncludeFiles property specifies the files to be included in the package specification file if static memory management is enabled and you are using selective framework includes. (Default = oxf/MemAlloc.h)

## StaticMemoryPoolDeclaration

The StaticMemoryPoolDeclaration property specifies the declaration of the memory pool for timeouts. The default value is as follows: DECLARE\_MEMORY\_ALLOCATOR(\$Class, \$BaseNumberOfInstances)

## StaticMemoryPoolGetMemory

If you have specified a value for the property BaseNumberOfInstances in order to specify the amount of memory that should be allocated for a specific kind of event, then the application will create new events by taking memory from this allocated pool. The StaticMemoryPoolGetMemory property specifies the code that should be used for getting memory from the pool.

*Default = RIC\_MEMORY\_ALLOCATOR\_GET(\$Class);*

## StaticMemoryPoolImplementation

The StaticMemoryPoolImplementation property specifies the generated code in the implementation file for a memory pool implementation (see the BaseNumberOfInstances property). The default value is as follows:

```
IMPLEMENT_MEMORY_ALLOCATOR($Class, $BaseNumberOfInstances,  
$AdditionalNumberOfInstances, $ProtectStaticMemoryPool)
```

## StaticMemoryPoolMemberDeclaration

If you have specified a value for the property BaseNumberOfInstances in order to specify the amount of memory that should be allocated for a specific kind of event, then the application will create new events by taking memory from this allocated pool. The StaticMemoryPoolMemberDeclaration property specifies the code used for declaring the pointer used by the event/class instance to access the pool so that memory can be taken and returned to the pool as required.

*Default = RIC\_DECLARE\_MEMORY\_ALLOCATOR\_MEMBER(\$Class)*

## StaticMemoryPoolReturnMemory

If you have specified a value for the property BaseNumberOfInstances to specify that amount of memory that should be allocated for a specific kind of event, then the application will return memory to this pool when an event is destroyed. The StaticMemoryPoolReturnMemory property specifies the code that should be used for returning memory to the pool.

*Default = RIC\_MEMORY\_ALLOCATOR\_RETURN(\$me, \$Class);*

## TestEventTypeCall

*The TestEventTypeCall property specifies the test used in event consumption code to check if the currently consumed event is of a given type. (Default = IS\_EVENT\_TYPE\_OF(\$Id))*

## TimeoutId

*The TimeoutId property specifies the ID reserved for timeout events. (Default = OMTimeoutEventId)*

## TimerMaxTimeouts

The TimerMaxTimeouts property specifies the maximum number of timeouts allowed simultaneously in the system, if the TimerMaxTimeouts property for the configuration is not overridden. In the framework, the default number of timers is 100.

*Default = empty string*

## TimerResolution

The TimerResolution property allows you to override the default tick time used.

The number entered is the number of milliseconds used for the tick time.

The default tick time (currently 100 milliseconds) is defined by RiCTimerManagerDefaultTicktime in the file RiCTimer.c

*Default = Blank*

## UseDirectReactiveDeletion

The UseDirectReactiveDeletion property determines whether direct deletion of reactive instances (by using the delete operator) is used instead of graceful framework termination (by using the reactive destroy() method). When this property is set to True, the code specified in the EnableDirectReactiveDeletion is generated in the main prior to the call to OXF::init(). See the EnableDirectReactiveDeletion property definition and the upgrade history on the support site for more information on this functionality.

*Default = Cleared*

## UseExternFrameworkGlobalVariable

For projects that use the MicroC profile, the property UseExternFrameworkGlobalVariable controls the allocation used for the framework's global variable. If the value of this property is set to True, the generated code includes only an extern declaration for the framework's global variable. If the value of the property is set to False, the framework's global variable is allocated and initialized in the generated code.

See also the property C.CG::Framework::FrameworkGlobalVariableName.

*Default = False*

## UseManagedTimeoutCanceling

The UseManagedTimeoutCanceling property specifies whether the framework uses the pre-Rational Rhapsody 6.0 scheme of timeout creation and cancellation (so OMTimerManager is responsible for cancellation of timeouts). In Rational Rhapsody, the framework moves the responsibility for a timeout cancellation from the timer manager to the timeout client (the reactive object).

This change reduces the timer manager responsibilities and the overhead in timeout management (thus improving timeout scheduling performance). The change also includes changes in the generated code (the user reactive objects hold pointers to the waiting timeouts in order to enable canceling). When loading a pre-6.0 model, Rational Rhapsody sets the project C.CG::Framework::UseManagedTimeoutCanceling to True to set the system-compatibility mode. See the upgrade history on the support site for more information.

*Default = Cleared*

## UsePreviousOxfReactiveVtblStruct

At one point, the mechanism used for deleting "reactives" was changed to use a delayed deletion mechanism. When this change was made, two additional fields were added to the struct `RiCReactive_Vtbl_t`. Since the initialization code generated by Rhapsody now takes these additional fields into account, this resulted in problems for users who were using newer versions of Rhapsody but were using their own custom framework based on the Rhapsody framework that was provided before this change was made.

If you are using such a custom framework, you can set the value of the property `UsePreviousOxfReactiveVtblStruct` to `True`, and then Rhapsody will generate initialization code that works with the smaller `RiCReactive_Vtbl_t` struct.

*Default = False*

## UseRhp5CompatibilityAPI

The `UseRhp5CompatibilityAPI` property specifies whether to use the virtual functions of the core implementation classes that existed in the pre-Rational Rhapsody 6.0 framework. The Rational Rhapsody 6.0 framework introduces a set of interfaces for the core behavioral framework.

The interfaces define a concise API for the framework and enable you to replace the actual implementation of these interfaces while maintaining the framework behavior. As a result of the interfaces' introduction, the framework behavioral classes (`OMReactive`, `OMThread`, and `OMEvent`) use a new set of virtual operations to implement the interfaces and provide the behavioral infrastructure. To support existing customizations of these classes (made by inheriting and overriding the virtual operations), the framework can work in a mode where the pre-6.0 API virtual operations are called.

When loading a pre-6.0 model, Rational Rhapsody sets the project property `C_CG::Framework::UseRhp5CompatibilityAPI` to `True` to set the system-compatibility mode. If this is set to `True`, the pre-6.0 API is called by the framework instead of the interface-based API. Without this flag, user customizations compile but is not called. See the upgrade history on the support site for more information on the Version 5. x compatibility mode.

*Default = Cleared*

## Generalization

The Generalization metaclass contains a property used to support generalization. See the Rational Rhapsody help for more information on generalization.

## Animate

The `Animate` property specifies whether animation code is generated for an element. You can specify your own animation function by using the `CG::Attribute::AnimSerializeOperation` property. The semantics of the `Animate` property is always in favor of the owner settings:

- If a package Animate property is set to False, all the classes owned by the package are not animated, regardless of the class Animate settings.
- If a class Animate property is set to False, all the elements in the class (attributes, operations, relations, and so on) are not animated.
- If an operation Animate property is set to False, all the arguments are not animated.
- If the AnimateArguments property is set to False, all the arguments are not animated, regardless of the specific argument Animate property settings.

*Default = Checked*

## Simplify

If you are using the Rational Rhapsody customizable code generation mechanism, the Simplify property can be used to change the way specific types of elements are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - The element is ignored.
- Copy - The element is copied from the original to the simplified model. It is not modified in any way.
- Default - Uses the standard simplification for this item, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for the element has been applied.

*Default = "Default"*

## INTEGRITY

This metaclass contains the properties that manipulate the INTEGRITY operating system environment.

### AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

This property reflects the change in Version 4.1 where the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created.

This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

## **AdditionalReservedWords**

The AdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody. The property value is checked at run time when you name/rename an element, based on the active configuration environment setting. Note that this property affects the algorithm only when the active configuration is of the selected environment. The default values are as follows:

*Default = empty string*

## **BLDAdditionalDefines**

The BLDAdditionalDefines property specifies additional compiler preprocessor flags. The default values are as follows:

*Default = empty string*

## **BLDAdditionalOptions**

The BLDAdditionalOptions property specifies additional compilation switches. The C INTEGRITY default value is as follows:

:optimizestrategy=space :driver\_opts=--diag\_suppress=14 :driver\_opts=--diag\_suppress=550

## **BLDIncludeAdditionalBLD**

The BLDIncludeAdditionalBLD specifies additional build options.

*Default = Empty MultiLine*

## **BLDMainExecutableOptions**

The BLDMainExecutableOptions property specifies the options generated in the main build file of the executable component of the model. The default value for Ada is as follows:

:target\_os=integrity :C\_library=full :integrity\_option=dynamic :staticlink=true

## **BLDMainLibraryOptions**

The `BLDMainLibraryOptions` property specifies the options generated in the main build file of the library component of the model.

The C default value is as follows:

```
:defines=_DEBUG :target_os=integrity
```

## BLDTarget

The `BLDTarget` property specifies the target BSP (for example, `:target=Win32`). This property also affects the names of the framework libraries used in the link. The C default value is `"mbx800."`

## BuildArgumentsInIDE

The `BuildArgumentsInIDE` property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the `C_CG::[environment]::BuildCommandInIDE` property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the `BuildInIDE` property is set to `True`.

*Default = Blank*

## BuildCommandInIDE

The `BuildCommandInIDE` property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the `BuildInIDE` property is set to `True`.

*Default = Blank*

## BuildCommandSet

The `BuildCommandSet` property generates a set of commands in the makefile to build a debug or release version of the configuration. To change this property, use the Configuration window in the browser - do not change it by using the Properties window or by modifying the `site.prp` file. Note that this property also affects the names of the framework libraries used in the link. The possible values are as follows:

- `Debug` - Generate the debug command that is set in the makefile.
- `DebugNoExp` - Generate the debug command that is set in the makefile without the exceptions flag



(:cx\_option=exceptions).

- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

The default value for MultiWin32 is DebugNoExp; for the other environments, the default value is Debug.

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT/etc/IntegrityMake.bat\" IntegrityBuild.bat buildLibs bld \$BLDTarget"*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## CompileSwitches

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property.

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction an IDE such as Eclipse, you can use the EnableDebugIntegrationWithIDE property to specify whether or not the IDE debugger is to be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to Checked, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's exit() function is called automatically. The system function

calls the destructors of framework instances in the order it determines. The EndApplicationCode property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an endApplication() function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to endApplication(), it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The EntryPoint property specifies the name of the main program for a given environment.

The default value is "main."

See also the definition of the EntryPointDeclarationModifier property for more information.

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens has three parameters, each with an integer value:

- TotalNumberOfTokens - The number of tokens in the regular expression
- FileTokenPosition - The position of the file name token in the expression
- LineTokenPosition - The position of the line number token in the expression

The default values are as follows: TotalNumberOfTokens=3 FileTokenPosition=1 LineTokenPosition=2

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the C\_CG::<Environment>::ExeName property plus the value of the ExeExtension property.

*Default = .mod*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the

value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the C\_CG::<Environment>::ExeExtension property.

*(Default = Blank)*

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile.

Note: If you select Release Build Set, these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

*Default = \$(OMROOT)\LangC\lib\\$(LIB\_PREFIX)WebComponents\$(LIB\_POSTFIX)\$(LIB\_EXT),  
\$(OMROOT)\lib\\$(LIB\_PREFIX)WebServices\$(LIB\_POSTFIX)\$(LIB\_EXT)*

## HasIDEInterface

The HasIDEInterface property is a Boolean value that specifies whether IDE support is enabled.

If IDE support is enabled (Checked), the IDEInterfaceDLL property points to an IDE adapter that provides connection to the IDE. If the property is set to Cleared, IDE support is disabled and IDE services are not attempted. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = Cleared.*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment.

*Default = .cc*

## IDEInterfaceDLL

The IDEInterfaceDLL property is a string that points to the IDE adapter DLL. You should not have any reason to modify this property. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody. The default values are as follows:

Environment Default Value QNXNeutrinoCW \$OMROOT/DLLs/CodeWarriorIDE.dll INTEGRITY  
Empty string IntegrityESTL VxWorks \$OMROOT/DLLs/TornadoIDE.dll

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file. The default values are as follows:

*Default = empty string*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\IntegrityMake.bat\" \$makefile \$maketarget"*

## InvokeMakeGenerator

The InvokeMakeGenerator specifies the path to the executable program for an external makefile generator. This external generator is started each time you request a makefile generation. If the specified path is incorrect, Rational Rhapsody generates an error message.

If you are using a full-featured external code generator, this property setting is ignored.

*Default = \$OMROOT/etc/IntegrityMakefileGenerator.bat*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format.

If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .a*

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the C\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*(Default = .bld)*

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the C\_CG::<Environment>::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## ObjectName

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = empty string*

## ObjectsDirectory

The `ObjectsDirectory` property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = work*

## ObjExtension

The `ObjExtension` property specifies the extension appended to compiled object files for a given environment.

*Default = .o*

## OSFileSystemCaseSensitive

The `OSFileSystemCaseSensitive` property specifies whether the OS file system for a given environment is case sensitive.

*Default = Cleared*

## ParseErrorDescript

The `ParseErrorDescript` property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = ([^"]+)"[,][ ]line ([0-9]+)[:] (warning/error/catastrophic error) (.\*)*

## ParseErrorMessage

The `ParseErrorMessage` property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning)

Along with the `ErrorMessageTokensFormat` property, `ParseErrorMessage` specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = ([^:]+)[:]( [0-9]+)[:]*

## ParseSeverityError

The `ParseSeverityError` property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^"]+)"[,][ ]line ([0-9]+)[:] (error/catastrophic error)*

## ParseSeverityWarning

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^"]+)"[,][ ]line ([0-9]+)[:] (warning)*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = False*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely.

In such configurations, the host is the machine running the product, whereas the target is the machine running the application.

To run remotely, the UseRemoteHost property must be set to Checked. If UseRemoteHost is Checked and RemoteHost is blank, the current host name is used for the remote host. You can use this as a workaround if you have problems running animated applications on Windows 95. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = empty string*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment. The extension ".h" is the default for C.

## UpdateBuildSettingsInIDE

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## UseNewBuildOutputWindow

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the CG::General::ShowLogViewAfterBuild property to True.

*Default = Checked*

## UseRemoteHost

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Checked:*

## INTEGRITY5

This metaclass contains the properties that manipulate the INTEGRITY5 operating system environment.

## AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

This property reflects the change in Version 4.1 where the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created.

This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files. To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

## AdditionalReservedWords

The AdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody. The property value is checked at run time when you name/rename an element, based on the active configuration environment setting. Note that this property affects the algorithm only when the active configuration is of the selected



environment. The default values are as follows:

Environment Default Value Borland \_\_asm \_\_finallynaked \_\_based \_\_inline \_\_single\_inheritance \_\_cdecl  
\_\_int8 \_\_stdcall \_\_declspec \_\_int16 thread dllexport \_\_int32 \_\_try dllimport \_\_int64 \_\_virtual\_inheritance  
\_\_except \_\_leave \_\_fastcall \_\_multiple\_inheritance Microsoft MicrosoftDLL MSStandardLibrary GNAT  
Empty string INTEGRITY IntegrityESTL JDK Linux MontaVista OsePPCDiab QNXNeutrinoCW  
QNXNeutrinoGCC Solaris2 Solaris2GNU VxWorks OseSfk receive \_\_asm \_\_finally naked \_\_based  
\_\_inline \_\_single\_inheritance \_\_cdecl \_\_int8 \_\_stdcall \_\_declspec \_\_int16 dllexport \_\_int32 \_\_try  
dllimport \_\_int64 \_\_virtual\_inheritance \_\_except \_\_leave \_\_fastcall \_\_multiple\_inheritance

## AnimInstLibs

The AnimInstLibs property is used to specify the static libraries that are required when Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with INST\_LIBS.

*Default = -l\$(LibPrefix)OxfInst\$(BLDTarget)\$OMLibSuffix\$LibExtension  
-l\$(LibPrefix)AomAnim\$(BLDTarget)\$OMLibSuffix\$LibExtension  
-l\$(FrameworkLibPrefix)OmComAppl\$(BLDTarget)\$OMLibSuffix\$LibExtension*

## AnimPreprocessor

The AnimPreprocessor property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with INST\_FLAGS.

*Default = -D\_OMINSTRUMENT*

## BLDAdditionalDefines

The BLDAdditionalDefines property specifies additional compiler preprocessor flags. The default values are as follows:

Environment Default Value INTEGRITY Empty string MultiWin32 IntegrityESTL ESTL

## BLDAdditionalOptions

The BLDAdditionalOptions property specifies additional compilation switches. The default values are as follows:

Environment Default Value INTEGRITY :optimizestrategy=space :driver\_opts=--diag\_suppress=14  
:driver\_opts=--diag\_suppress=550 IntegrityESTL :optimizestrategy=space  
:driver\_opts=--diag\_suppress=14 :driver\_opts=--diag\_suppress=550 :cx\_mode=extended\_embedded  
:cx\_lib=eecx :stdcxxincdirs=\$(INTEGRITY\_ROOT)\eecxx :stdcxxincdirs=\$(INTEGRITY\_ROOT)\ansi  
MultiWin32 :cx\_template\_option=noimplicit :add\_output\_ext=checked :cx\_e\_option=msgnumbers  
:cx\_option=exceptions :check=bounds :check=assignbound :check=nilderef :cx\_template=local  
:cx\_remark=14 :cx\_remark=161 :cx\_remark=837 :cx\_remark=817 :cx\_remark=815 :cx\_remark=47  
:cx\_remark=69 :cx\_remark=830 :cx\_remark=550 :prelink.args=-r :prelink.args=-X7

## **BLDIncludeAdditionalBLD**

The BLDIncludeAdditionalBLD specifies additional build options.

*Default = Empty MultiLine*

## **BLDMainExecutableOptions**

The BLDMainExecutableOptions property specifies the options generated in the main build file of the executable component of the model.

The C default value is as follows:

`-G -Ospace -dynamic -non_shared`

## **BLDMainLibraryOptions**

The BLDMainLibraryOptions property specifies the options generated in the main build file of the library component of the model.

The C default value is as follows:

`-G -Ospace -non_shared`

## **BLDTarget**

The BLDTarget property specifies the target BSP. For example, `":target=Win32"`. This property also affects the names of the framework libraries used in the link. The C default value is `"mbx800."`

## **BuildArgumentsInIDE**

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the `C_CG::[environment]::BuildCommandInIDE` property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## **BuildCommandInIDE**

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration. To change this property, use the Configuration window in the browser - do not change it by using the Properties window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link. The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*Default = Debug*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT/etc\Integrity5Make.bat\" IntegrityBuild.bat buildLibs \$BLDTarget "*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## CompileDebug

The CompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

## CompileSwitches

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property.

There is not default value.

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

The default value is as follows: `-D_DEBUG -G`.

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

There is not default value.

## DebugLibSuffix

A suffix added to library names. OMLibSuffix keyword is replaced with this property or ReleaseLibSuffix according to the compilation to the build type: Release/Debug.

## DebugSwitches

The DebugSwitches property sets the debug level used in debug switches. The default values are as follows:

Environment	Possible Values	Default Value
INTEGRITY	Default, Multi, None, Plain, and Stack	Default
OBJECTADA	-ga, -gc, -ga -gc	-ga
RAVEN_PPC	-ga, -gc, -ga -gc	-ga
SPARK	Empty string	

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction an IDE such as Eclipse, you can use the EnableDebugIntegrationWithIDE property to specify whether or not the IDE debugger is to be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to Checked, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's `exit()` function is called automatically. The system function calls the destructors of framework instances in the order it determines. The `EndApplicationCode` property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an `endApplication()` function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to `endApplication()`, it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The `EntryPoint` property specifies the name of the main program for a given environment.

The default value is "main."

See also the definition of the `EntryPointDeclarationModifier` property for more information.

## ErrorMessageTokensFormat

The `ErrorMessageTokensFormat`, working with the `ParseErrorMessage` property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. `ErrorMessageTokensFormat` defines the number and location of tokens within the regular expression defined by the `ParseErrorMessage` property. `ErrorMessageTokens` has three parameters, each with an integer value:

- `TotalNumberOfTokens` - The number of tokens in the regular expression
- `FileTokenPosition` - The position of the file name token in the expression
- `LineTokenPosition` - The position of the line number token in the expression

*Default = TotalNumberOfTokens=3,FileTokenPosition=1,LineTokenPosition=2*

## ExeExtension

The `ExeExtension` property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the `C_CG::<Environment>::ExeName` property plus the value of the `ExeExtension` property.

*Default = .mod*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the C\_CG::<Environment>::ExeExtension property.

*(Default = Blank)*

## ExtraFilesToCopy

A list of file names (with full paths) separated with commas. The generator copies this list of files to the folder of the makefile only if the file does not already exist.

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile.

Note: If you select Release Build Set, these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

*Default = \$(LibPrefix)WebComponents\$(BLDTarget)\$OMLibSuffix\$LibExtension ,  
\$(OMRoot)/lib/\$(FrameworkLibPrefix)WebServices\$(BLDTarget)\$OMLibSuffix\$LibExtension.*

## HasIDEInterface

The HasIDEInterface property is a Boolean value that specifies whether IDE support is enabled.

If IDE support is enabled (Checked), the IDEInterfaceDLL property points to an IDE adapter that provides connection to the IDE. If the property is set to Cleared, IDE support is disabled and IDE services are not attempted. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = Cleared*

## IDEInterfaceDLL

The IDEInterfaceDLL property is a string that points to the IDE adapter DLL. You should not have any reason to modify this property. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody. The default values are as follows:

Environment Default Value QNXNeutrinoCW \$OMROOT/DLLs/CodeWarriorIDE.dll INTEGRITY  
Empty string IntegrityESTL VxWorks \$OMROOT/DLLs/TornadoIDE.dll

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment.

*Default = .c*

## IntegrityLinkFile

The name of the Integrity link file that should be added to the makefile template.

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file. The default values are as follows:

Environment Default Value Borland "\$executable" GNAT Microsoft MicrosoftDLL MSStandardLibrary  
MultiWin32 (Ada) NucleusPLUS-PPC OBJECTADA RAVEN\_PPC SPARK INTEGRITY Empty string  
IntegrityESTL MicrosoftWinCE.NET JDK "\$OMROOT/etc/Executer.exe"  
"\$OMROOT/etc/jdkrun.bat" \$makefile Main\$ComponentName" Linux \$executable MultiWin32 (C++)  
QNXNeutrinoCW QNXNeutrinoGCC MicrosoftWinCE "\$OMROOT/etc/msceRun.bat" \$executable  
IX86EM OsePPCDiab "\$OMROOT/etc/osesfkRun.bat" \$executable OseSfk Solaris2 xterm -e  
\$executable Solaris2GNU

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT/etc/vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

The InvokeMake default values are as follows:

```
Environment Default Value Borland $OMROOT/etc/Executer.exe "\"$OMROOT\etc\bc5make.bat\"
$makefile $maketarget" GNAT "$makefile" $maketarget INTEGRITY ESTL
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\IntegrityMake.bat\" $makefile $maketarget" Integrity
JDK "$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\jdkmake.bat\" $makefile $maketarget" Linux
$OMROOT/etc/linuxmake $makefile $maketarget Microsoft "$OMROOT/etc/Executer.exe"
 "\"$OMROOT\etc\msmake.bat\" $makefile $maketarget" MicrosoftDLL MSStandardLibrary
MicrosoftWinCE "$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\mscemake.bat\" $makefile
$maketarget IX86EM" MicrosoftWinCE.NET "$OMROOT/etc/Executer.exe"
 "\"$OMROOT\etc\msceNETmake.bat\" $makefile $maketarget x86" MultiWin32 (Ada)
"$OMROOT/etc/Executer.exe" "$OMROOT\etc\AdaMultiWin32Make.bat $makefile $maketarget"
OBJECTADA "$OMROOT/etc/Executer.exe" "$OMROOT\etc\ObjectAdaMake.bat $makefile
$maketarget" OsePPCDiab "$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\oseppcdiabmake.bat\"
$makefile $maketarget" OseSfk "$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\osesfkmake.bat\"
$makefile $maketarget" QNXNeutrinoCW "$OMROOT/etc/Executer.exe"
 "\"$OMROOT\etc\qnxcwmake.bat\" $makefile $maketarget" QNXNeutrinoGCC Empty string
RAVEN_PPC "$OMROOT/etc/Executer.exe" "$OMROOT\etc\ObjectAdaRavenPPCMake.bat $makefile
$maketarget" Solaris2 $OMROOT/etc/sol2make $makefile $maketarget Solaris2GNU SPARK
"$OMROOT/etc/Executer.exe" "$OMROOT\etc\SPARKMake.bat $makefile $maketarget" VxWorks
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vxmake.bat\" $makefile $maketarget"
```

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format.

If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .a*

## LibPrefix

Combines all of the prefixes of the library names. The C default value is \$(FrameworkLibPrefix)\$(OMMultipleAddressSpacesPrefix).

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode. The default values are as follows:



Environment Default Value Borland Empty string (blank) GNAT Microsoft MicrosoftDLL  
MicrosoftWinCE.NET MSStandardLibrary MultiWin32 NucleusPLUS-PPC OseSfk INTEGRITY -G  
IntegrityESTL Linux -g OsePPCDiab QNXNeutrinoCW QNXNeutrinoGCC Solaris2 Solaris2GNU  
VxWorks

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode. The default values are as follows:

Environment Default Value Borland Empty string GNAT INTEGRITY IntegrityESTL Microsoft  
MicrosoftDLL MicrosoftWinCE.NET MSStandardLibrary MultiWin32 NucleusPLUS-PPC OsePPCDiab  
OseSfk VxWorks Linux -O QNXNeutrinoCW QNXNeutrinoGCC Solaris2 Solaris2GNU

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode. The default values are as follows:

Environment Default Value Borland \$OMLinkCommandSet Linux MultiWin32 (C++)  
NucleusPLUS-PPC OsePPCDiab QNXNeutrinoGCC Solaris2 Solaris2GNU VxWorks INTEGRITY  
--one\_instantiation\_per\_object \$OMLinkCommandSet -cpu=\$(TARGET\_CPU) -map IntegrityESTL  
Microsoft \$OMLinkCommandSet /NOLOGO MicrosoftDLL MicrosoftWinCE.NET MSStandardLibrary  
OseSfk -nologo \$OMLinkCommandSet QNXNeutrinoCW -static

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the C\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

(Default = .gpj)

## MakeFileContentForExe1

The MakeFileContentForExe1 property is the content of the makefile for an executable component type. The default value is as follows:

```
#!/gbuild primaryTarget=$PrimaryTarget [INTEGRITY Application] -bsp $BLDTarget -os_dir  
$IntegrityRoot -o $OMTargetName$ExeExtension -object_dir=$ObjectsDirectory  
$BLDMainExecutableOptions $OMMultipleAddressSpacesSwitches $KernelProject  
$MakeFileNameForExe2 $IntegrateFile $BSPFile $ConnectionFile $ResourceFile  
$OMMultipleAddressSpacesAdditionalFiles
```

## MakeFileContentForExe2

The MakeFileContentForExe2 property is the content of the makefile for an executable component type. The default value is as follows:

```
#!/gbuild [Program] -o $OMTargetName -object_dir=$ObjectsDirectory $BLDAdditionalOptions
-I$OMRoot/LangC -L$OMRoot/LangC/lib $OMUserIncludePath $LinkSwitches $OMCompilationFlag
$CompileSwitches $OMReusableFlag $OMInstrumentationFlags $OMInstrumentationLibs
$OMMultipleAddressSpacesLibraries $BLDAdditionalDefines $OMUserLibs
$OMMainFiles$ImpExtension $OMSrcFiles $IntegrityLinkFile $LinkerFile
```

## MakeFileContentForLib1

The MakeFileContentForLib1 property provides the content of the makefile for a library component type. The default value is as follows:

```
#!/gbuild primaryTarget=$PrimaryTarget [Project] -bsp $BLDTarget -os_dir $IntegrityRoot
-object_dir=$ObjectsDirectory $BLDMainLibraryOptions $OMMultipleAddressSpacesSwitches
$KernelProject $MakeFileNameForLib2 $IntegrateFile $BSPFile $ConnectionFile $ResourceFile
$OMMultipleAddressSpacesAdditionalFiles
```

## MakeFileContentForLib2

The MakeFileContentForLib2 property provides the content of the makefile for a library component type. The default value is as follows:

```
#!/gbuild [Library] -o $OMTargetName$LibExtension -object_dir=$ObjectsDirectory
$BLDAdditionalOptions -I$OMRoot/LangC $OMUserIncludePath $OMCompilationFlag
$CompileSwitches $OMInstrumentationFlags $OMReusableFlag $BLDAdditionalDefines $OMSrcFiles
```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the C.CG::::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## MakeFileNameForExe1

The MakeFileNameForExe1 property is the name of the makefile for an executable component type.

*Default = \$(OMTargetName)\$MakeExtension*

## **MakeFileNameForExe2**

The MakeFileNameForExe2 property is the name of the makefile for an executable component type.

*Default = \$(OMTargetName)\_program\$MakeExtension*

## **MakeFileNameForLib1**

The MakeFileNameForLib1 property is the name of the makefile for a library component type.

*Default = \$(OMTargetName)\$MakeExtension*

## **MakeFileNameForLib2**

The MakeFileNameForLib2 property is the name of the makefile for a library component type.

*Default = \$(OMTargetName)\_library\$MakeExtension.*

## **MultipleAddressSpacesIntFileContent**

The MultipleAddressSpacesIntFileContent property provides the content of the MultipleAddressSpacesIntFileName file with the number of the Integrate configuration file for compiling a list of address spaces that use also POSIX shared memory manager.

The content values are as follows:

```
Kernel Filename DynamicDownload EndKernel $OMSubComponentInfo AddressSpace
shared_memory_manager Filename posix_shm_manager MaximumPriority 200 Language $OMLanguage
Task Initial StartIt true EndTask EndAddressSpace
```

## **MultipleAddressSpacesIntFileName**

The MultipleAddressSpacesIntFileName property identifies a file with this name to be created in case of multiple address space compilation.

*Default = \$OMTargetName.int*

## **MultipleAddressSpacesLibraries**

The MultipleAddressSpacesLibraries property names of libraries to add in case of multiple address space usage.

The default value is as follows:

`-l$(FrameworkLibPrefix)Dox$(BLDTarget)$LibExtension -llibposix$LibExtension  
-llibshm_client$LibExtension`

## MultipleAddressSpacesPrefix

The MultipleAddressSpacesPrefix property specifies the prefix that is added to libraries in case of multiple address space compilation. OMMultipleAddressSpacesPrefix keyword adds this prefix when needed.

*Default = Distributed.*

## MultipleAddressSpacesSwitches

A switch for multiple address space compilation. The makefile template can add it directly but it is preferred to use the keyword OMMultipleAddressSpacesSwitches – that checks whether this switch should be added.

## NetAndSocketLibs

A list of library names that is added to OMWebLibs keyword if web-enabling flag is on or to OMInstrumentationFlags keyword if the instrumentation is in animation mode.

## NoneInstLibs

The NoneInstLibs property is used to specify the static libraries that are required when Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with INST\_LIBS.

*Default = -l\$(LibPrefix)Oxf\$(BLDTarget)\$OMLibSuffix\$LibExtension*

## NonePreprocessor

The NonePreprocessor property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with INST\_FLAGS.

*Default = Blank*

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## ObjectName

The `ObjectName` property specifies an alternative name for the compiled object file in the generated makefile.

*Default = empty string*

## ObjectsDirectory

The `ObjectsDirectory` property specifies an alternate name for the directory for compiled object files in the generated makefile.

The default value is "work."

## ObjExtension

The `ObjExtension` property specifies the extension appended to compiled object files for a given environment.

The C default value is .o.

## OSFileSystemCaseSensitive

The `OSFileSystemCaseSensitive` property specifies whether the OS file system for a given environment is case sensitive.

*Default = Cleared.*

## ParseErrorDescript

The `ParseErrorDescript` property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = ([^"]+)"[,][ ]line ([0-9]+)[: ](warning|error|catastrophic error) (.\*)*

## ParseErrorMessage

The `ParseErrorMessage` property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning) Along with the `ErrorMessageTokensFormat` property, `ParseErrorMessage` specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

The C default value is ([^"]+)"[,][ ]line ([0-9]+)[: ](warning|error|catastrophic error).

## ParseSeverityError

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^"]+)"[,][ ]line ([0-9]+)[: ] (error/catastrophic error)*

## ParseSeverityWarning

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^"]+)"[,][ ]line ([0-9]+)[: ] (warning)*

## PosixSharedMemoryFiles

This list is copied only in case of multiple address space compilation.

*Default = \$OMROOT/MakeTpl/posix\_shm\_manager.gpj,\$OMROOT/MakeTpl/shm\_area.c.*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Cleared.*

## ReleaseLibSuffix

A suffix added to library names. OMLibSuffix keyword is replaced with this property or DebugLibSuffix according to the compilation to the build type: Release/Debug.

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely.

In such configurations, the host is the machine running the product, whereas the target is the machine running the application.

To run remotely, the UseRemoteHost property must be set to Checked. If UseRemoteHost is Checked and RemoteHost is blank, the current host name is used for the remote host. You can use this as a workaround if you have problems running animated applications on Windows 95. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = empty string*

## ReusableStatechartSwitches

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts. See the upgrade history on the support site for information on this change.

*Default = OM\_REUSABLE\_STATECHART\_IMPLEMENTATION.*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment. The extension ".ads" is the default for Ada.

## TraceInstLibs

The TraceInstLibs property is used to specify the static libraries that are required when Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with INST\_LIBS.

*Default =*  
*-l\$OMRoot/LangCpp/lib/\$(FrameworkLibPrefix)TomTraceRiC\$(BLDTarget)\$OMLibSuffix\$LibExtension*  
*-l\$OMRoot/LangCpp/lib/\$(FrameworkLibPrefix)Oxf\$(BLDTarget)\$OMLibSuffix\$LibExtension*  
*-l\$OMRoot/LangCpp/lib/\$(FrameworkLibPrefix)OmComAppl\$(BLDTarget)\$OMLibSuffix\$LibExtension*  
*-l\$(LibPrefix)OxfInstTrace\$(BLDTarget)\$OMLibSuffix\$LibExtension*  
*-l\$(LibPrefix)AomTrace\$(BLDTarget)\$OMLibSuffix\$LibExtension*  
*-l\$(FrameworkLibPrefix)OmComAppl\$(BLDTarget)\$OMLibSuffix\$LibExtension*

## TracePreprocessor

The TracePreprocessor property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with INST\_FLAGS.

*Default = \$(DEFINE\_QUALIFIER)OMTRACER*

## UpdateBuildSettingsInIDE

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## UseNewBuildOutputWindow

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the CG::General::ShowLogViewAfterBuild property to True.

*Default = Checked*

## UseRemoteHost

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Cleared.*

## WebInstLibs

A list of library names that is added to OMWebLibs keyword if web-enabling flag is on.

## Interface

The Interface metaclass contains properties related to code generation for interfaces.

## SupportTriggeredOperations

Prior to release 8.1.4, Rhapsody's C code generation did not support the use of triggered operations in interfaces. Support for triggered operations was added in release 8.1.4. In order to preserve the previous code generation behavior for pre-8.1.4 models, the property SupportTriggeredOperations was added to the backward compatibility settings for C, with a value of False.

## Link

The Link metaclass contains a property that controls how links are handled during model simplification.

## Simplify

If you are using the Rational Rhapsody customizable code generation mechanism, the Simplify property



can be used to change the way specific types of elements are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - The element is ignored.
- Copy - The element is copied from the original to the simplified model. It is not modified in any way.
- Default - Uses the standard simplification for this item, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for the element has been applied.

*Default = "Default"*

## Linux

The Linux metaclass contains the Environment settings (Compiler, framework libraries, and so on) for Linux.

### AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)/LangC/osconfig/Linux*

### AdditionalReservedWords

The AdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active

configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

## AnimIncludeDirectories

The AnimIncludeDirectories property is used to specify the directories that must be referenced in the makefile for projects where Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with INST\_INCLUDES.

*Default = \$(INCLUDE\_QUALIFIER)\$(OMROOT)/LangC/aom  
\$(INCLUDE\_QUALIFIER)\$(OMROOT)/LangCpp/tom*

## AnimInstLibs

The AnimInstLibs property is used to specify the static libraries that are required when Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with INST\_LIBS.

*Default = \$(OMROOT)/LangC/lib/linuxaomanim\$(LIB\_EXT)*

## AnimOxfLibs

The AnimOxfLibs property is used to specify the framework libraries that are required when Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with OXF\_LIBS.

*Default = \$(OMROOT)/LangC/lib/linuxoxfinst\$(LIB\_EXT)  
\$(OMROOT)/LangC/lib/linuxomcomappl\$(LIB\_EXT)*

## AnimPreprocessor

The AnimPreprocessor property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with INST\_FLAGS.

*Default = \$(DEFINE\_QUALIFIER)OMANIMATOR*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration. To change this property, use the Configuration window in the browser - do not change it by using the Properties window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link.

The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

The default value for MultiWin32 is DebugNoExp; for the other environments, the default value is Debug.

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = \$OMROOT/etc/linuxmake linuxbuild.mak build*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## CompilerFlags

The CompilerFlags property allows you to define additional compilation flags. The value of the property is inserted into the value of the CompileSwitches property (Linux) or CPPCompileSwitches (cygwin). In the generated makefile, you can see the value of this property in the line that begins with ConfigurationCPPCompileSwitches=.

*Default = Blank*

## CompileSwitches

The CompileSwitches property specifies the compiler switches.

The default value is as follows:

```
-I. -I$OMDefaultSpecificationDirectory -I$(OMROOT) -I$(OMROOT)/LangC
-I$(OMROOT)/LangC/oxf $(INST_FLAGS) $(INCLUDE_PATH) $(INST_INCLUDES)
$OMCPPCompileCommandSet -c
```

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The default values are as follows:

```
@echo Compiling $OMFileImpPath $(CREATE_OBJ_DIR) @$ (CC) $OMFileCPPCompileSwitches -o  
$OMFileObjPath $OMFileImpPath
```

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*Default = -g*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*Default = -O*

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: \$OMFileObjPath : \$OMFileImpPath "\*.bmp" "\*.ico" "\*.rc2"

The default value is \$OMFileObjPath : \$OMFileImpPath \$OMFileSpecPath \$OMFileDependencies.

## DuplicateLibsListInMakeFile

The DuplicateLibsListInMakeFile property is a Boolean value that specifies whether Rational Rhapsody should duplicate the libraries list in the generated makefile link command. This property supports linkers that are sensitive to library order in the link command.

*Default = Checked*

## EndApplicationCode

When the main function ends, the system's `exit()` function is called automatically. The system function calls the destructors of framework instances in the order it determines. The `EndApplicationCode` property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an `endApplication()` function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to `endApplication()`, it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The `EntryPoint` property specifies the name of the main program for a given environment.

The default value is "main."

See also the definition of the `EntryPointDeclarationModifier` property for more information.

## ErrorMessageTokensFormat

The `ErrorMessageTokensFormat`, working with the `ParseErrorMessage` property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. `ErrorMessageTokensFormat` defines the number and location of tokens within the regular expression defined by the `ParseErrorMessage` property. `ErrorMessageTokens` has three parameters, each with an integer value:

- `TotalNumberOfTokens` - The number of tokens in the regular expression
- `FileTokenPosition` - The position of the file name token in the expression
- `LineTokenPosition` - The position of the line number token in the expression

The default value is "ToTalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2."

## ExeExtension

The `ExeExtension` property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the `C_CG::<Environment>::ExeName` property plus the value of the `ExeExtension` property.

*Default = Empty string*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active

component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the C\_CG::<Environment>::ExeExtension property.

*(Default = Blank)*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements. The file inclusions are generated in the makefile.

The C default value is \$OMSpecIncludeInElements \$OMImpIncludeInElements.

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile.

Note: If you select Release Build Set, these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

The default value is \$(OMROOT)/LangC/lib/linuxWebComponents\$(LIB\_EXT),  
\$(OMROOT)/lib/linuxWebServices\$(LIB\_EXT).

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment. The default values are as follows:

*( C Default = .c )*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

The C default value is "include."

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*Default = empty string*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment. The default values are as follows:

*Default = .a*

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode. The default values are as follows:

*Default = -g*

## LinkerFlags

The LinkerFlags property allows you to define linker flags. The value of the property is inserted into the

value of the LinkSwitches property. In the generated makefile, you can see the value of this property in the line that begins with LINK\_FLAGS=.

*Default = -lpthread -lstdc++*

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode. The default values are as follows:

*Default = -O*

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode. The default values are as follows:

*Default = \$OMLinkCommandSet*

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the C\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*(Default = .mak)*

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.



**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration. For example, the default content of the target type section of a C++ makefile for the Microsoft environment is as follows: ##### Target type (Debug/Release)

```
#####  
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease  
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease BuildSet=$OMBuildSet  
SUBSYSTEM=$OMSubSystem COM=$OMCOM RPFrameWorkDll=$OMRPFrameWorkDll  
ConfigurationCPPCompileSwitches= $OMReusableStatechartSwitches  
$OMConfigurationCPPCompileSwitches !IF "$ (RPFrameWorkDll)" == "True"  
ConfigurationCPPCompileSwitches= $(ConfigurationCPPCompileSwitches) /D "FRAMEWORK_DLL"  
!ENDIF !IF "$ (COM)" == "True" SUBSYSTEM=/SUBSYSTEM:windows !ENDIF
```

**Compilation Flags** The compilation flags section of the makefile contains the default compilation flags stored in the CompileSwitches property. For example, the default content of the compilation flags section of a C++ makefile for the Microsoft environment is as follows: ##### Compilation flags

```
#####  
INCLUDE_QUALIFIER=/I LIB_PREFIX=MS
```

**Commands Definitions** The commands definition section of the makefile specifies programs to execute from the makefile. For example, the default commands definition section of a C++ makefile for the Microsoft environment is as follows: ##### Commands definition #####

```
##### RMDIR = rmdir LIB_CMD=link.exe -lib  
LINK_CMD=link.exe LIB_FLAGS=$OMConfigurationLinkSwitches  
LINK_FLAGS=$OMConfigurationLinkSwitches $(SUBSYSTEM) / MACHINE:I386
```

**Generated Macros** The generated macros section of the makefile contains a variable that expands to the Rational Rhapsody-generated macros in the makefile. For example: ##### Generated macros

```
##### $OMContextMacros  
OBJ_DIR=$OMObjectsDir !IF "$ (OBJ_DIR)" != "" CREATE_OBJ_DIR=if not exist $(OBJ_DIR) mkdir  
$(OBJ_DIR) CLEAN_OBJ_DIR= if exist $(OBJ_DIR) $(RMDIR) $(OBJ_DIR) !ELSE  
CREATE_OBJ_DIR= CLEAN_OBJ_DIR= !ENDIF
```

The \$OMContextMacros keyword expands several macros in the makefile. Each makefile macro has its own keyword. You can use these keywords separately to customize the makefile. The \$OMContextMacros variable modifies target-specific variables. Replace the \$OMContextMacros line in the MakeFileContent property with the following: FLAGSFILE=\$OMFlagsFile

```
RULESFILE=$OMRulesFile OMROOT=$OMROOT C_EXT=$OMImplExt H_EXT=$OMSpecExt  
OBJ_EXT=$OMObjExt EXE_EXT=$OMExeExt LIB_EXT=$OMLibExt  
INSTRUMENTATION=$OMInstrumentation TIME_MODEL=$OMTimeModel  
TARGET_TYPE=$OMTargetType TARGET_NAME=$OMTargetName $OMAllDependencyRule  
TARGET_MAIN=$OMTargetMain LIBS=$OMLibs INCLUDE_PATH=$OMIncludePath  
ADDITIONAL_OBJS=$OMAdditionalObjs OBJS= $OMObjs
```

**Predefined Macros** The predefined macros section of the makefile contains other macros than the Rational Rhapsody-generated macros specified in the generated macros section. For example, the default predefined macros section of a C++ makefile for the Microsoft environment is as follows:

```
##### Predefined macros #####  
##### $(OBJS) : $(INST_LIBS)  
$(OXF_LIBS) LIB_POSTFIX= !IF "$ (BuildSet)" == "Release" LIB_POSTFIX=R !ENDIF !IF  
"$ (TARGET_TYPE)" == "Executable" LinkDebug=$(LinkDebug) /DEBUG  
LinkRelease=$(LinkRelease) /OPT:NOREF !ELSEIF "$ (TARGET_TYPE)" == "Library"  
LinkDebug=$(LinkDebug) /DEBUGTYPE:CV !ENDIF !IF "$ (INSTRUMENTATION)" == "Animation"  
INST_FLAGS=/D "OMANIMATOR" INST_INCLUDES=/I $(OMROOT)\LangCpp\aoom /I  
$(OMROOT)\LangCpp\tom !IF "$ (RPFrameWorkDll)" == "True" INST_LIBS=
```

```

OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxanimdll$(LIB_POSTFIX) $(LIB_EXT)
!ELSE INST_LIBS= $(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomanim$(LIB_POSTFIX) $(LIB_EXT)
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfirst$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX)$(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$ (INSTRUMENTATION)" == "Tracing" INST_FLAGS=/D
"OMTRACER" INST_INCLUDES=/I $(OMROOT)\LangCpp\iom /I $(OMROOT)\LangCpp\tom !IF
"$(RPFrameWorkDll)" == "True" INST_LIBS= OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfracedll$(LIB_POSTFIX)$(LIB_EXT) !ELSE
INST_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)tomtrace$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomtrace$(LIB_POSTFIX) $(LIB_EXT) OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfirst$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX)$(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$ (INSTRUMENTATION)" == "None" INST_FLAGS=
INST_INCLUDES= INST_LIBS= !IF "$(RPFrameWorkDll)" == "True"
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfdll$(LIB_POSTFIX)$(LIB_EXT) !ELSE
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$(LIB_POSTFIX)$(LIB_EXT) !ENDIF
SOCK_LIB= !ELSE !ERROR An invalid Instrumentation $(INSTRUMENTATION) is specified.
!ENDIF

```

**Generated Dependencies** The generated dependencies section of the makefile contains a variable that expands to Rational Rhapsody-generated dependencies and compilation instructions. For example, the generated dependencies section of a C++ makefile for the Microsoft environment is as follows:

```

##### Generated dependencies #####
##### $OMContextDependencies
$OMFileObjPath : $OMMainImplementationFile $(OBJS) $(CPP) $(ConfigurationCPPCompileSwitches)
/Fo"$OMFileObjPath" $OMMainImplementationFile

```

**Linking Instructions** The linking instructions section of the makefile contains the predefined linking instructions. For example, the default linking instructions section of a C++ makefile for the Microsoft environment is as follows:

```

##### Linking instructions #####
#####
$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS) $OMFileObjPath
$OMMakefileName $OMModelLibs @echo Linking $(TARGET_NAME)$(EXE_EXT) $(LINK_CMD)
$OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \ $(OXF_LIBS) \
$(SOCK_LIB) \ $(LINK_FLAGS) /out:$(TARGET_NAME)$(EXE_EXT)
$(TARGET_NAME)$(LIB_EXT) : $(OBJS) $(ADDITIONAL_OBJS) $OMMakefileName @echo
Building library $@ $(LIB_CMD) $(LIB_FLAGS) /out:$(TARGET_NAME)$(LIB_EXT) $(OBJS)
$(ADDITIONAL_OBJS) clean: @echo Cleanup $OMCleanOBJS if exist $OMFileObjPath erase
$OMFileObjPath if exist *$(OBJ_EXT) erase *$(OBJ_EXT) if exist $(TARGET_NAME).pdb erase
$(TARGET_NAME).pdb if exist $(TARGET_NAME)$(LIB_EXT) erase
$(TARGET_NAME)$(LIB_EXT) if exist $(TARGET_NAME).ilk erase $(TARGET_NAME).ilk if exist
$(TARGET_NAME)$(EXE_EXT) erase $(TARGET_NAME)$(EXE_EXT) $(CLEAN_OBJ_DIR)

```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the C.CG::<Environment>::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## **NoneIncludeDirectories**

The NoneIncludeDirectories property is used to specify the directories that must be referenced in the makefile for projects where Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with INST\_INCLUDES.

*Default = Blank*

## **NoneInstLibs**

The NoneInstLibs property is used to specify the static libraries that are required when Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with INST\_LIBS.

*Default = Blank*

## **NoneOxfLibs**

The NoneOxfLibs property is used to specify the framework libraries that are required when Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with OXF\_LIBS.

*Default = \$(OMROOT)/LangC/lib/linuxoxf\$(LIB\_EXT)*

## **NonePreprocessor**

The NonePreprocessor property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with INST\_FLAGS.

*Default = Blank*

## **NullValue**

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## **ObjCleanCommand**

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default = \$(RM) \$OMFileObjPath*

## ObjectName

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = empty string*

## ObjectsDirectory

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = empty string*

## ObjExtension

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*Default = .o*

## OSFileSystemCaseSensitive

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Checked*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning)

Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = ([^:]+)[:]?([0-9]+)[:]*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation.

*Default = /*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Checked*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running the product, whereas the target is the machine running the application. To run remotely, the UseRemoteHost property must be set to True.

If UseRemoteHost is True and RemoteHost is blank, the current host name is used for the remote host. You can use this as a workaround if you have problems running animated applications on Windows 95.

The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = empty string*

## ReusableStatechartSwitches

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts. See the upgrade history on the support site for information on this change. The default values are as follows:

Environment Default Value Borland -DOM\_REUSABLE\_STATECHART\_IMPLEMENTATION Linux  
NucleusPLUS-PPC OsePPCDiab OseSfk QNXNeutrinoCW QNXNeutrinoGCC Solaris2 Solaris2GNU  
VxWorks Microsoft /D "OM\_REUSABLE\_STATECHART\_IMPLEMENTATION" MicrosoftDLL  
MicrosoftWinCE.NET MSStandardLibrary INTEGRITY  
OM\_REUSABLE\_STATECHART\_IMPLEMENTATION IntegrityESTL MultiWin32

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

The extension ".h" is the default for C.

## TraceIncludeDirectories

The TraceIncludeDirectories property is used to specify the directories that must be referenced in the makefile for projects where Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with INST\_INCLUDES.

*Default = \$(INCLUDE\_QUALIFIER)\$(OMROOT)/LangCpp/aom  
\$(INCLUDE\_QUALIFIER)\$(OMROOT)/LangCpp/tom*

## **TraceInstLibs**

The TraceInstLibs property is used to specify the static libraries that are required when Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with INST\_LIBS.

*Default = \$(OMROOT)/LangCpp/lib/linuxomtraceRiC\$(LIB\_EXT)  
\$(OMROOT)/LangCpp/lib/linuxomcomappl\$(LIB\_EXT) \$(OMROOT)/LangCpp/lib/linuxoxf\$(LIB\_EXT)  
\$(OMROOT)/LangC/lib/linuxaomtrace\$(LIB\_EXT)*

## **TraceOxfLibs**

The TraceOxfLibs property is used to specify the framework libraries that are required when Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with OXF\_LIBS.

*Default = \$(OMROOT)/LangC/lib/linuxoxfinst\$(LIB\_EXT)  
\$(OMROOT)/LangC/lib/linuxomcomappl\$(LIB\_EXT)*

## **TracePreprocessor**

The TracePreprocessor property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with INST\_FLAGS.

*Default = \$(DEFINE\_QUALIFIER)OMTRACER*

## **UnixLineTerminationStyle**

The UnixLineTerminationStyle property specifies whether generated files use the UNIX end-of-line style.

If this property is set to Cleared, the end-of-line style depends on the host type (for example, DOS style on Windows machines, and UNIX style on Solaris machines).

*Default = Checked*

## **UnixPathNameForOMROOT**

The UnixPathNameForOMROOT property specifies whether the makefile must include UNIXstyle path names.

The pRISM compilers do not tolerate DOS-style path conventions. If you do not set this property correctly, there might be many compilation problems.

*Default = Checked*

## **UseRemoteHost**

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Checked*

## **MainLoopCygwin**

The MainLoopCygwin metaclass contains the Environment settings (Compiler, framework libraries, and so on).

### **ExtraFilesToCopy**

A list of file names (with full paths) separated with commas. The generator copies this list of files to the folder of the makefile only if the file does not already exist.

## **MainLoopLinux**

The MainLoopLinux metaclass contains the Environment settings (Compiler, framework libraries, and so on).

### **ExtraFilesToCopy**

A list of file names (with full paths) separated with commas. The generator copies this list of files to the folder of the makefile only if the file does not already exist.

## **MainLoopMicrosoft**

The MainLoopMicrosoft metaclass contains the Environment settings (Compiler, framework libraries, and so on) for MainLoopMicrosoft.

### **AlternativeOMROOTForMakefile**

The generated makefile uses the OMROOT path to build the application. The value of the OMROOT

variable is defined in the Rhapsody.ini file. Use the AlternativeOMROOTForMakefile property to specify an alternative location to be used instead of the OMROOT path.

*Default = Empty string*

## **AssertMacroName**

MicroC Execution Framework (MXF) has calls to the assert macro embedded in its code. Use the AssertMacroName property to specify the name of this macro in its environment. See also the UseAssertMacro property.

*Default = assert*

## **ExtraFilesToCopy**

A list of file names (with full paths) separated with commas. The generator copies this list of files to the folder of the makefile only if the file does not already exist.

## **QuoteOMROOT**

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = True*

## **UseAssertMacro**

Use the UseAssertMacro property to enable or disable compilation of code related to the assert macro. See also the AssertMacroName property.

*Default = Cleared*

## **MainLoopMSVC9**

The MainLoopMicrosoft metaclass contains the Environment settings (Compiler, framework libraries, and so on) for MainLoopMSVC9.

## **AlternativeOMROOTForMakefile**

The generated makefile uses the OMROOT path to build the application. The value of the OMROOT variable is defined in the Rhapsody.ini file. Use the AlternativeOMROOTForMakefile property to specify an alternative location to be used instead of the OMROOT path.



*Default = Empty string*

## **AssertMacroName**

MicroC Execution Framework (MXF) has calls to the assert macro embedded in its code. Use the AssertMacroName property to specify the name of this macro in its environment. See also the UseAssertMacro property.

*Default = assert*

## **ExtraFilesToCopy**

A list of file names (with full paths) separated with commas. The generator copies this list of files to the folder of the makefile only if the file does not already exist.

## **QuoteOMROOT**

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = True*

## **UseAssertMacro**

Use the UseAssertMacro property to enable or disable compilation of code related to the assert macro. See also the AssertMacroName property.

*Default = Cleared*

# **MainLoopS12**

The MainLoopMicrosoft metaclass contains the Environment settings (Compiler, framework libraries, and so on) for MainLoopS12.

## **AlternativeOMROOTForMakefile**

The generated makefile uses the OMROOT path to build the application. The value of the OMROOT variable is defined in the Rhapsody.ini file. Use the AlternativeOMROOTForMakefile property to specify an alternative location to be used instead of the OMROOT path.

*Default = Empty string*

## AssertMacroName

MicroC Execution Framework (MXF) has calls to the assert macro embedded in its code. Use the AssertMacroName property to specify the name of this macro in its environment. See also the UseAssertMacro property.

*Default = assert*

## ExtraFilesToCopy

A list of file names (with full paths) separated with commas. The generator copies this list of files to the folder of the makefile only if the file does not already exist.

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = True*

## UseAssertMacro

Use the UseAssertMacro property to enable or disable compilation of code related to the assert macro. See also the AssertMacroName property.

*Default = Cleared*

## MainLoopSTM32

The MainLoopMicrosoft metaclass contains the Environment settings (Compiler, framework libraries, and so on) for MainLoopSTM32.

## AlternativeOMROOTForMakefile

The generated makefile uses the OMROOT path to build the application. The value of the OMROOT variable is defined in the Rhapsody.ini file. Use the AlternativeOMROOTForMakefile property to specify an alternative location to be used instead of the OMROOT path.

*Default = Empty string*

## AssertMacroName

MicroC Execution Framework (MXF) has calls to the assert macro embedded in its code. Use the AssertMacroName property to specify the name of this macro in its environment. See also the UseAssertMacro property.

*Default = assert*

## ExtraFilesToCopy

A list of file names (with full paths) separated with commas. The generator copies this list of files to the folder of the makefile only if the file does not already exist.

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = True*

## UseAssertMacro

Use the UseAssertMacro property to enable or disable compilation of code related to the assert macro. See also the AssertMacroName property.

*Default = Cleared*

## Microsoft

The Microsoft metaclass contains the Environment settings (Compiler, framework libraries, and so on) for Microsoft compiler.

## AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.

- Add the `AdaptorSearchPath` property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)/LangC/osconfig/WIN32*

## AdditionalReservedWords

The `AdditionalReservedWords` property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use.

In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = \_\_asm \_\_finally naked \_\_based \_\_inline \_\_single\_inheritance \_\_cdecl \_\_int8 \_\_stdcall \_\_declspec \_\_int16 dllexport \_\_int32 \_\_try dllimport \_\_int64 \_\_virtual\_inheritance \_\_except \_\_leave \_\_fastcall \_\_multiple\_inheritance*

## BuildArgumentsInIDE

The `BuildArgumentsInIDE` property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the `C_CG::[environment]::BuildCommandInIDE` property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the `BuildInIDE` property is set to `True`.

*Default = Blank*

## BuildCommandInIDE

The `BuildCommandInIDE` property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the `BuildInIDE` property is set to `True`.

*Default = Blank*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration. To change this property, use the Configuration window in the browser - do not change it by using the Properties window or by modifying the site.prp file.

Note that this property also affects the names of the framework libraries used in the link. The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*Default = Debug*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\$OMROOT/etc/Executer.exe" "%\"\$OMROOT\"%etc\msmake.bat msbuild.mak build \*  
*\"USE\_PDB=FALSE\" \"\"*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## CompileSwitches

The CompileSwitches property specifies the compiler switches.

The default value is as follows:

```
/I . /I $OMDefaultSpecificationDirectory /I "$(OMROOT)\LangC" /I "$(OMROOT)\LangC\oxf" /nologo  
/W3 /GX $OMCPCPPCompileCommandSet /D "_AFXDLL" /D "WIN32" /D "_CONSOLE" /D "_MBCS"
```

`/D "_WINDOWS" $(INST_FLAGS) $(INCLUDE_PATH) $(INST_INCLUDES) /c`

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = \$(CREATE\_OBJ\_DIR) \$(CC) \$OMFileCPPCompileSwitches /Fo"\$OMFileObjPath" "\$OMFileImpPath".*

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*Default = /Zi /Od /D "\_DEBUG" /MDd /Fd"\${TARGET\_NAME}"*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

The default value is `/Ox /D"NDEBUG" /MD /Fd"${TARGET_NAME}"`.

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: `$OMFileObjPath : $OMFileImpPath "*.bmp" "*.ico" "*.rc2"`

The default value is `$OMFileObjPath : $OMFileImpPath $OMFileSpecPath $OMFileDependencies .`

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction an IDE such as Eclipse, you can use the EnableDebugIntegrationWithIDE property to specify whether or not the IDE debugger is to be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to Checked, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's `exit()` function is called automatically. The system function calls the destructors of framework instances in the order it determines. The `EndApplicationCode` property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an `endApplication()` function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to `endApplication()`, it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The `EntryPoint` property specifies the name of the main program for a given environment.

The default value is "main."

See also the definition of the `EntryPointDeclarationModifier` property for more information.

## ErrorMessageTokensFormat

The `ErrorMessageTokensFormat`, working with the `ParseErrorMessage` property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. `ErrorMessageTokensFormat` defines the number and location of tokens within the regular expression defined by the `ParseErrorMessage` property. `ErrorMessageTokens` has three parameters, each with an integer value:

- `TotalNumberOfTokens` - The number of tokens in the regular expression
- `FileTokenPosition` - The position of the file name token in the expression
- `LineTokenPosition` - The position of the line number token in the expression

The default value is "ToTalNumberOfTokens=3,FileTokenPosition=1,LineTokenPosition=2."

## ExeExtension

The `ExeExtension` property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the `C_CG::<Environment>::ExeName` property plus the value of the `ExeExtension` property.

*Default = .exe*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the C\_CG::<Environment>::ExeExtension property.

*(Default = Blank)*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements. The file inclusions are generated in the makefile.

The default value is \$OMSpecIncludeInElements \$OMImpIncludeInElements.

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile. Note that if you select Release Build Set, these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

The default is \$(OMROOT)\LangC\lib\\$(LIB\_PREFIX)WebComponents\$(LIB\_POSTFIX)\$(LIB\_EXT), \$(OMROOT)\lib\\$(LIB\_PREFIX)WebServices\$(LIB\_POSTFIX)\$(LIB\_EXT), ws2\_32\$(LIB\_EXT).

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment. The default values are as follows:

*(Default = .c)*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*(Default = INCLUDE)*



## InvokeAnimatedExecutable

The AnimationPortRange value in the rhapsody.ini file makes it possible to use animation with multiple instances of Rhapsody running simultaneously.

Ordinarily, when you run an application with animation, the executable is launched using the command specified by the InvokeExecutable property. However, if you have specified a value greater than zero for AnimationPortRange in your rhapsody.ini file, the executable is launched using the command specified by the InvokeAnimatedExecutable property. By default, the value of InvokeAnimatedExecutable includes a reference to the port to use for animation.

*Default = \$executable -port \$port*

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*(Default = "\$executable")*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

"rhapsody\_dir>\share/etc/Executer.exe""dir; dir | more"

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$.

*(Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\msmake.bat\" \$makefile \$maketarget")*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format.

If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a

given environment.

*(Default = .lib)*

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

*Default = empty string*

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

*(Default = \$OMLinkCommandSet /NOLOGO)*

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the C\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*(Default = .mak)*

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration. For example, the default content of the target type section of a C++ makefile for the Microsoft environment is as follows: ##### Target type (Debug/Release)

```
#####  
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease  
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease BuildSet=$OMBuildSet  
SUBSYSTEM=$OMSubSystem COM=$OMCOM RPFramWorkDll=$OMRPFramWorkDll  
ConfigurationCPPCompileSwitches= $OMReusableStatechartSwitches  
$OMConfigurationCPPCompileSwitches !IF "$ (RPFramWorkDll)" == "True"  
ConfigurationCPPCompileSwitches= $(ConfigurationCPPCompileSwitches) /D "FRAMEWORK_DLL"  
!ENDIF !IF "$ (COM)" == "True" SUBSYSTEM=/SUBSYSTEM:windows !ENDIF
```

**Compilation Flags** The compilation flags section of the makefile contains the default compilation flags stored in the CompileSwitches property. For example, the default content of the compilation flags section of a C++ makefile for the Microsoft environment is as follows: ##### Compilation flags

```
#####  
INCLUDE_QUALIFIER=/I LIB_PREFIX=MS
```

**Commands Definitions** The commands definition section of the makefile specifies programs to execute from the makefile. For example, the default commands definition section of a C++ makefile for the Microsoft environment is as follows: ##### Commands definition #####

```
##### RMDIR = rmdir LIB_CMD=link.exe -lib  
LINK_CMD=link.exe LIB_FLAGS=$OMConfigurationLinkSwitches  
LINK_FLAGS=$OMConfigurationLinkSwitches $(SUBSYSTEM) / MACHINE:I386
```

**Generated Macros** The generated macros section of the makefile contains a variable that expands to the Rational Rhapsody-generated macros in the makefile. For example: ##### Generated macros

```
##### $OMContextMacros  
OBJ_DIR=$OMObjectsDir !IF "$ (OBJ_DIR)" != "" CREATE_OBJ_DIR=if not exist $(OBJ_DIR) mkdir  
$(OBJ_DIR) CLEAN_OBJ_DIR= if exist $(OBJ_DIR) $(RMDIR) $(OBJ_DIR) !ELSE  
CREATE_OBJ_DIR= CLEAN_OBJ_DIR= !ENDIF
```

The \$OMContextMacros keyword expands several macros in the makefile. Each makefile macro has its own keyword. You can use these keywords separately to customize the makefile. The \$OMContextMacros variable modifies target-specific variables.

Replace the \$OMContextMacros line in the MakeFileContent property with the following:

```
FLAGSFILE=$OMFlagsFile RULESFILE=$OMRulesFile OMROOT=$OMROOT  
C_EXT=$OMImplExt H_EXT=$OMSpecExt OBJ_EXT=$OMObjExt EXE_EXT=$OMExeExt  
LIB_EXT=$OMLibExt INSTRUMENTATION=$OMInstrumentation TIME_MODEL=$OMTimeModel  
TARGET_TYPE=$OMTargetType TARGET_NAME=$OMTargetName $OMAllDependencyRule  
TARGET_MAIN=$OMTargetMain LIBS=$OMLibs INCLUDE_PATH=$OMIncludePath  
ADDITIONAL_OBJS=$OMAdditionalObjs OBJS= $OMObjs
```

**Predefined Macros** The predefined macros section of the makefile contains other macros than the Rational Rhapsody-generated macros specified in the generated macros section. For example, the default predefined macros section of a C++ makefile for the Microsoft environment is as follows:

```
##### Predefined macros #####  
##### $(OBJS) : $(INST_LIBS)  
$(OXF_LIBS) LIB_POSTFIX= !IF "$ (BuildSet)" == "Release" LIB_POSTFIX=R !ENDIF !IF  
"$ (TARGET_TYPE)" == "Executable" LinkDebug=$(LinkDebug) /DEBUG
```

```

LinkRelease=$(LinkRelease) /OPT:NOREF !ELSEIF "$(TARGET_TYPE)" == "Library"
LinkDebug=$(LinkDebug) /DEBUGTYPE:CV !ENDIF !IF "$(INSTRUMENTATION)" == "Animation"
INST_FLAGS=/D "OMANIMATOR" INST_INCLUDES=/I $(OMROOT)\LangCpp\om /I
$(OMROOT)\LangCpp\tom !IF "$(RPFrameWorkDll)" == "True" INST_LIBS=
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfanimdll$(LIB_POSTFIX)$(LIB_EXT)
!ELSE INST_LIBS= $(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomanim$(LIB_POSTFIX)$(LIB_EXT)
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfirst$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX)$(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)" == "Tracing" INST_FLAGS=/D
"OMTRACER" INST_INCLUDES=/I $(OMROOT)\LangCpp\om /I $(OMROOT)\LangCpp\tom !IF
"$(RPFrameWorkDll)" == "True" INST_LIBS= OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfracedll$(LIB_POSTFIX)$(LIB_EXT) !ELSE
INST_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)tomtrace$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomtrace$(LIB_POSTFIX)$(LIB_EXT) OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfirst$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX)$(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)" == "None" INST_FLAGS=
INST_INCLUDES= INST_LIBS= !IF "$(RPFrameWorkDll)" == "True"
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfdll$(LIB_POSTFIX)$(LIB_EXT) !ELSE
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$(LIB_POSTFIX)$(LIB_EXT) !ENDIF
SOCK_LIB= !ELSE !ERROR An invalid Instrumentation $(INSTRUMENTATION) is specified.
!ENDIF

```

Generated Dependencies The generated dependencies section of the makefile contains a variable that expands to Rational Rhapsody-generated dependencies and compilation instructions. For example, the generated dependencies section of a C++ makefile for the Microsoft environment is as follows:

```

##### Generated dependencies #####
##### SOMContextDependencies
$OMFileObjPath : $OMMainImplementationFile $(OBJS) $(CPP) $(ConfigurationCPPCompileSwitches)
/Fo"$OMFileObjPath" $OMMainImplementationFile

```

Linking Instructions The linking instructions section of the makefile contains the predefined linking instructions. For example, the default linking instructions section of a C++ makefile for the Microsoft environment is as follows:

```

##### Linking instructions #####
#####
$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS) $OMFileObjPath
$OMMakefileName $OMModelLibs @echo Linking $(TARGET_NAME)$(EXE_EXT) $(LINK_CMD)
$OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \ $(OXF_LIBS) \
$(SOCK_LIB) \ $(LINK_FLAGS) /out:$(TARGET_NAME)$(EXE_EXT)
$(TARGET_NAME)$(LIB_EXT) : $(OBJS) $(ADDITIONAL_OBJS) $OMMakefileName @echo
Building library @@ $(LIB_CMD) $(LIB_FLAGS) /out:$(TARGET_NAME)$(LIB_EXT) $(OBJS)
$(ADDITIONAL_OBJS) clean: @echo Cleanup $OMCleanOBJS if exist $OMFileObjPath erase
$OMFileObjPath if exist *$(OBJ_EXT) erase *$(OBJ_EXT) if exist $(TARGET_NAME).pdb erase
$(TARGET_NAME).pdb if exist $(TARGET_NAME)$(LIB_EXT) erase
$(TARGET_NAME)$(LIB_EXT) if exist $(TARGET_NAME).ilk erase $(TARGET_NAME).ilk if exist
$(TARGET_NAME)$(EXE_EXT) erase $(TARGET_NAME)$(EXE_EXT) $(CLEAN_OBJ_DIR)

```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the `C.CG::<Environment>::MakeExtension` property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## **NullValue**

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## **ObjCleanCommand**

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*(Default = if exist \$OMFileObjPath erase \$OMFileObjPath)*

## **ObjectName**

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = empty string*

## **ObjectsDirectory**

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

## **ObjExtension**

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*(Default = .obj)*

## **OSFileSystemCaseSensitive**

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Cleared*

## ParseErrorDescript

The ParseErrorDescript property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = ([^()]+)([()([0-9]+)] [:] (error|warning|fatal error) (.\*)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning). Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

The default is ([^()]+)([()([0-9]+)] [:] (error|warning|fatal error).

## ParseMakeError

The ParseMakeError property is used to define a regular expression that represents the format of make process or linker error messages. This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = (NMAKE|LINK)(.\*)(fatal error)*

## ParseSeverityError

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = [:] (error|fatal error)*

## ParseSeverityWarning

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = [:] (warning)*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation.

*(Default = \ )*

## **QuoteOMROOT**

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Checked*

## **RemoteHost**

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running the product, whereas the target is the machine running the application. To run remotely, the UseRemoteHost property must be set to True.

If UseRemoteHost is True and RemoteHost is blank, the current host name is used for the remote host. You can use this as a workaround if you have problems running animated applications on Windows 95.

The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = empty string*

## **SpecExtension**

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*(Default = .h)*

## **UpdateBuildSettingsInIDE**

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## **UseNewBuildOutputWindow**

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the `CG::General::ShowLogViewAfterBuild` property to `True`.

*Default = Checked*

## UseRemoteHost

The `UseRemoteHost` property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Cleared*

## MicrosoftIDF

The `MicrosoftIDF` metaclass contains the Environment settings (Compiler, framework libraries, and so on) for `MicrosoftIDF` compiler.

## AdaptorSearchPath

The `AdaptorSearchPath` property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the `AdaptorSearchPath` property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*(Default = \$(OMROOT)/LangC/osconfig/WIN32)*

## AdditionalReservedWords

The `AdditionalReservedWords` property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use.

In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.



Note that this property affects the algorithm only when the active configuration is of the selected environment.

(Default = `__asm __finally naked __based __inline __single_inheritance __cdecl __int8 __stdcall __declspec __int16 dllexport __int32 __try dllimport __int64 __virtual_inheritance __except __leave __fastcall __multiple_inheritance`)

## BuildArgumentsInIDE

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the C\_CG::[environment]::BuildCommandInIDE property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandInIDE

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration. To change this property, use the Configuration window in the browser - do not change it by using the Properties window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link. The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*(Default = Debug)*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## CompileSwitches

The CompileSwitches property specifies the compiler switches.

The default value is as follows:

```
/I . /I "$(OMROOT)/LangC" /I "$(OMROOT)/LangC/idf" /I "$(OMROOT)/LangC/idf/Adapters/WIN32" /nologo /W3 /GX $OMCPPCompileCommandSet /D "_AFXDLL" /D "WIN32" /D "_CONSOLE" /D "_MBCS" /D "_WINDOWS" $(INST_FLAGS) $(INCLUDE_PATH) $(INST_INCLUDES) /c
```

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The default values are as follows:

```
$(CC) $OMFileCPPCompileSwitches /Fo"$OMFileObjPath" "$OMFileImpPath"
```

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*(Default = /Zi /Od /D "\_DEBUG" /MDd /Fd"\$(TARGET\_NAME)")*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*(Default = /Ox /D"NDEBUG" /MD /Fd"\$(TARGET\_NAME)" )*

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: `$OMFileObjPath : $OMFileImpPath "*.bmp" "*.ico" "*.rc2"`

The default value is `$OMFileObjPath : $OMFileImpPath $OMFileSpecPath $OMFileDependencies`.

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction an IDE such as Eclipse, you can use the EnableDebugIntegrationWithIDE property to specify whether or not the IDE debugger is to be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to Checked, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's exit() function is called automatically. The system function calls the destructors of framework instances in the order it determines. The EndApplicationCode property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an endApplication() function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to endApplication(), it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The EntryPoint property specifies the name of the main program for a given environment.

*(Default = main)*

See also the definition of the EntryPointDeclarationModifier property for more information.

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected

format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens has three parameters, each with an integer value:

- TotalNumberOfTokens - The number of tokens in the regular expression
- FileTokenPosition - The position of the file name token in the expression
- LineTokenPosition - The position of the line number token in the expression

The default value is "TotalNumberOfTokens=3,FileTokenPosition=1,LineTokenPosition=2."

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the C\_CG::<Environment>::ExeName property plus the value of the ExeExtension property.

*Default = .exe*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the C\_CG::<Environment>::ExeExtension property.

*(Default = Blank)*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements.

The file inclusions are generated in the makefile.

The default value is \$OMSpecIncludeInElements \$OMImpIncludeInElements.

## IDEVersion

When you select one of the provided Microsoft environments for a configuration in your model, you use the IDEVersion property to tell Rational Rhapsody which version of MS Visual Studio you are using, 2008 or 2010.

The value of this property is passed as a parameter to the Build Framework command, and to the command used to build your application. The value of the property is also used in properties that affect the content of the generated makefile.

The possible values are:

- VC9 - to specify that you are using Visual Studio 2008
- VC10 - to specify that you are using Visual Studio 2010
- VC11 - to specify that you are using Visual Studio 2012

*Default = VC10*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment. The default values are as follows:

*(Default = .c)*

## InvokeAnimatedExecutable

The AnimationPortRange value in the rhapsody.ini file makes it possible to use animation with multiple instances of Rhapsody running simultaneously.

Ordinarily, when you run an application with animation, the executable is launched using the command specified by the InvokeExecutable property. However, if you have specified a value greater than zero for AnimationPortRange in your rhapsody.ini file, the executable is launched using the command specified by the InvokeAnimatedExecutable property. By default, the value of InvokeAnimatedExecutable includes a reference to the port to use for animation.

*Default = \$executable -port \$port*

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*(Default = "\$OMROOT/etc/Executer.exe" "%\$OMROOT\etc\cygwinrun.bat" \$executable)*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

The InvokeMake default values are as follows:

```
Environment Default Value Borland $OMROOT/etc/Executer.exe "\"$OMROOT\etc\bc5make.bat\"  
$makefile $maketarget" GNAT "$makefile" $maketarget INTEGRITY ESTL  
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\IntegrityMake.bat\" $makefile $maketarget" Integrity  
JDK "$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\jdkmake.bat\" $makefile $maketarget" Linux  
$OMROOT/etc/linuxmake $makefile $maketarget Microsoft "$OMROOT/etc/Executer.exe"  
 "\"$OMROOT\etc\msmake.bat\" $makefile $maketarget" MicrosoftDLL MSStandardLibrary  
MicrosoftWinCE "$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\mscemake.bat\" $makefile  
$maketarget IX86EM" MicrosoftWinCE.NET "$OMROOT/etc/Executer.exe"  
 "\"$OMROOT\etc\msceNETmake.bat\" $makefile $maketarget x86" MultiWin32 (Ada)  
"$OMROOT/etc/Executer.exe" "$OMROOT\etc\AdaMultiWin32Make.bat $makefile $maketarget"  
OBJECTADA "$OMROOT/etc/Executer.exe" "$OMROOT\etc\ObjectAdaMake.bat $makefile  
$maketarget" OsePPCDiab "$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\oseppcdiabmake.bat\"  
$makefile $maketarget" OseSfk "$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\osesfkmake.bat\"  
$makefile $maketarget" QNXNeutrinoCW "$OMROOT/etc/Executer.exe"  
 "\"$OMROOT\etc\qnxcwmake.bat\" $makefile $maketarget" QNXNeutrinoGCC Empty string  
RAVEN_PPC "$OMROOT/etc/Executer.exe" "$OMROOT\etc\ObjectAdaRavenPPCMake.bat $makefile  
$maketarget" Solaris2 $OMROOT/etc/sol2make $makefile $maketarget Solaris2GNU SPARK  
"$OMROOT/etc/Executer.exe" "$OMROOT\etc\SPARKMake.bat $makefile $maketarget" VxWorks  
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vxmake.bat\" $makefile $maketarget"
```

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*(Default = .a)*

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode. The default values are as follows:

Environment Default Value Borland \$OMLinkCommandSet Linux MultiWin32 (C++)  
NucleusPLUS-PPC OsePPCDiab QNXNeutrinoGCC Solaris2 Solaris2GNU VxWorks INTEGRITY  
--one\_instantiation\_per\_object \$OMLinkCommandSet -cpu=\$(TARGET\_CPU) -map IntegrityESTL  
Microsoft \$OMLinkCommandSet /NOLOGO MicrosoftDLL MicrosoftWinCE.NET MSSstandardLibrary  
OseSfk -nologo \$OMLinkCommandSet QNXNeutrinoCW -static

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the C\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*(Default = .mak)*

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration. For example, the default content of the target type section of a C++ makefile for the Microsoft environment is as follows: ##### Target type (Debug/Release)

```
#####  
C++CompileDebug=$OMC++CompileDebug C++CompileRelease=$OMC++CompileRelease  
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease BuildSet=$OMBuildSet  
SUBSYSTEM=$OMSubSystem COM=$OMCOM RPFramWorkDll=$OMRPFramWorkDll  
ConfigurationC++CompileSwitches= $OMReusableStatechartSwitches  
$OMConfigurationC++CompileSwitches !IF "$ (RPFramWorkDll)" == "True"  
ConfigurationC++CompileSwitches= $(ConfigurationC++CompileSwitches) /D "FRAMEWORK_DLL"  
!ENDIF !IF "$ (COM)" == "True" SUBSYSTEM=/SUBSYSTEM:windows !ENDIF
```

**Compilation Flags** The compilation flags section of the makefile contains the default compilation flags stored in the CompileSwitches property. For example, the default content of the compilation flags section of a C++ makefile for the Microsoft environment is as follows: ##### Compilation flags

```
#####  
INCLUDE_QUALIFIER=/I LIB_PREFIX=MS
```

**Commands Definitions** The commands definition section of the makefile specifies programs to execute from the makefile. For example, the default commands definition section of a C++ makefile for the Microsoft environment is as follows: ##### Commands definition #####

```
##### RMDIR = rmdir LIB_CMD=link.exe -lib  
LINK_CMD=link.exe LIB_FLAGS=$OMConfigurationLinkSwitches  
LINK_FLAGS=$OMConfigurationLinkSwitches $(SUBSYSTEM) / MACHINE:I386
```

**Generated Macros** The generated macros section of the makefile contains a variable that expands to the Rational Rhapsody-generated macros in the makefile. For example: ##### Generated macros

```
##### $OMContextMacros  
OBJ_DIR=$OMObjectsDir !IF "$ (OBJ_DIR)" != "" CREATE_OBJ_DIR=if not exist $(OBJ_DIR) mkdir  
$(OBJ_DIR) CLEAN_OBJ_DIR= if exist $(OBJ_DIR) $(RMDIR) $(OBJ_DIR) !ELSE  
CREATE_OBJ_DIR= CLEAN_OBJ_DIR= !ENDIF
```

The \$OMContextMacros keyword expands several macros in the makefile. Each makefile macro has its own keyword. You can use these keywords separately to customize the makefile. The

\$OMContextMacros variable modifies target-specific variables. Replace the \$OMContextMacros line in the MakeFileContent property with the following: FLAGSFILE=\$OMFlagsFile

```
RULESFILE=$OMRulesFile OMROOT=$OMROOT C_EXT=$OMImplExt H_EXT=$OMSpecExt  
OBJ_EXT=$OMObjExt EXE_EXT=$OMExeExt LIB_EXT=$OMLibExt  
INSTRUMENTATION=$OMInstrumentation TIME_MODEL=$OMTimeModel  
TARGET_TYPE=$OMTargetType TARGET_NAME=$OMTargetName $OMAllDependencyRule  
TARGET_MAIN=$OMTargetMain LIBS=$OMLibs INCLUDE_PATH=$OMIncludePath  
ADDITIONAL_OBJS=$OMAdditionalObjs OBJS= $OMObjs
```

**Predefined Macros** The predefined macros section of the makefile contains other macros than the Rational Rhapsody-generated macros specified in the generated macros section. For example, the default predefined macros section of a C++ makefile for the Microsoft environment is as follows:

```
##### Predefined macros #####  
##### $(OBJS) : $(INST_LIBS)  
$(OXF_LIBS) LIB_POSTFIX= !IF "$ (BuildSet)" == "Release" LIB_POSTFIX=R !ENDIF !IF  
"$ (TARGET_TYPE)" == "Executable" LinkDebug=$(LinkDebug) /DEBUG  
LinkRelease=$(LinkRelease) /OPT:NOREF !ELSEIF "$ (TARGET_TYPE)" == "Library"  
LinkDebug=$(LinkDebug) /DEBUGTYPE:CV !ENDIF !IF "$ (INSTRUMENTATION)" == "Animation"
```



```

INST_FLAGS=/D "OMANIMATOR" INST_INCLUDES=/I $(OMROOT)\LangCpp\aoom /I
$(OMROOT)\LangCpp\tom !IF "$ (RPFrameWorkDll)" == "True" INST_LIBS=
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfanimdll$(LIB_POSTFIX) $(LIB_EXT)
!ELSE INST_LIBS= $(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomanim$(LIB_POSTFIX) $(LIB_EXT)
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfirst$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)" == "Tracing" INST_FLAGS=/D
"OMTRACER" INST_INCLUDES=/I $(OMROOT)\LangCpp\aoom /I $(OMROOT)\LangCpp\tom !IF
"$ (RPFrameWorkDll)" == "True" INST_LIBS= OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfracedll$(LIB_POSTFIX) $(LIB_EXT) !ELSE
INST_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)tomtrace$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomtrace$(LIB_POSTFIX) $(LIB_EXT) OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfirst$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)" == "None" INST_FLAGS=
INST_INCLUDES= INST_LIBS= !IF "$ (RPFrameWorkDll)" == "True"
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfdll$(LIB_POSTFIX) $(LIB_EXT) !ELSE
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB= !ELSE !ERROR An invalid Instrumentation $(INSTRUMENTATION) is specified.
!ENDIF

```

Generated Dependencies The generated dependencies section of the makefile contains a variable that expands to Rational Rhapsody-generated dependencies and compilation instructions. For example, the generated dependencies section of a C++ makefile for the Microsoft environment is as follows:

```

##### Generated dependencies #####
##### SOMContextDependencies
$OMFileObjPath : $OMMainImplementationFile $(OBJS) $(CPP) $(ConfigurationCPPCompileSwitches)
/Fo"$OMFileObjPath" $OMMainImplementationFile

```

Linking Instructions The linking instructions section of the makefile contains the predefined linking instructions. For example, the default linking instructions section of a C++ makefile for the Microsoft environment is as follows:

```

##### Linking instructions #####
#####
$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS) $OMFileObjPath
$OMMakefileName $OMModelLibs @echo Linking $(TARGET_NAME)$(EXE_EXT) $(LINK_CMD)
$OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \ $(OXF_LIBS) \
$(SOCK_LIB) \ $(LINK_FLAGS) /out:$(TARGET_NAME)$(EXE_EXT)
$(TARGET_NAME)$(LIB_EXT) : $(OBJS) $(ADDITIONAL_OBJS) $OMMakefileName @echo
Building library $@ $(LIB_CMD) $(LIB_FLAGS) /out:$(TARGET_NAME)$(LIB_EXT) $(OBJS)
$(ADDITIONAL_OBJS) clean: @echo Cleanup $OMCleanOBJS if exist $OMFileObjPath erase
$OMFileObjPath if exist *$(OBJ_EXT) erase *$(OBJ_EXT) if exist $(TARGET_NAME).pdb erase
$(TARGET_NAME).pdb if exist $(TARGET_NAME)$(LIB_EXT) erase
$(TARGET_NAME)$(LIB_EXT) if exist $(TARGET_NAME).ilk erase $(TARGET_NAME).ilk if exist
$(TARGET_NAME)$(EXE_EXT) erase $(TARGET_NAME)$(EXE_EXT) $(CLEAN_OBJ_DIR)

```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the C.CG::<Environment>::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## **ObjCleanCommand**

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build. The default value is as follows:

if exist \$OMFileObjPath erase \$OMFileObjPath

## **ObjectName**

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = empty string*

## **ObjectsDirectory**

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile. The default values are as follows:

Environment Default Value INTEGRITY work Integrity ESTL MultiWin32 obj\_dir All others Empty string

## **ObjExtension**

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*(Default = .obj)*

## **OSFileSystemCaseSensitive**

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Cleared*

## **ParseErrorDescript**

The ParseErrorDescript property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = ([^()]+)([()([0-9]+)] [:] (error/warning/fatal error) (.\*)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning)

Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*(Default = ([^:]+):([0-9]+):[:])*

## ParseMakeError

The ParseMakeError property is used to define a regular expression that represents the format of make process or linker error messages. This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = (NMAKE)(.\*)(fatal error)*

## ParseSeverityError

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = [:] (error/fatal error)*

## ParseSeverityWarning

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = [:] (warning)*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation.

*(Default = /)*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*(Default = Checked )*

## UpdateBuildSettingsInIDE

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## UseNewBuildOutputWindow

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the CG::General::ShowLogViewAfterBuild property to True.

*Default = Checked*

## ModelElement

The metaclass ModelElement contains properties that can be used to customize code generation by changing the way that Rational Rhapsody handles specific elements when it transforms a model into a simplified model before generating code.

In general, the properties in this metaclass relate to model elements that can be found under other types of model elements, for example, descriptions and annotations. These properties are therefore visible at different project levels - for example, package, class, and attribute.

## CallOperationGenerationPattern

The CallOperationGenerationPattern property stores the value entered in the "Code pattern" field on the General tab of the Features window for Call Operation elements.

*Default = \$operation*

## ForLoopInitialization

If you have a "for" loop in a flowchart, the ForLoopInitialization property is used to store the loop initialization code that you entered in the "Loop initialization" field on the General tab of the Features window for the relevant action or decision node.

For detailed instructions on using "for" loops in flowcharts, see the KC topic called ""While" loops and "for" loops in flowcharts".

*Default = Blank*

## ForLoopStep

If you have a "for" loop in a flowchart, the ForLoopStep property is used to store the loop increment code that you entered in the "Loop step" field on the General tab of the Features window for the relevant action or decision node.

For detailed instructions on using "for" loops in flowcharts, see the KC topic called ""While" loops and "for" loops in flowcharts".

*Default = Blank*

## RefactorRenameRegularExpression

When you use the Refactor:Rename feature, Rhapsody searches the user-defined code in the model for all occurrences of the name of the specified element, including places where the element name appears as part of the name of elements whose name is derived from the element that is being renamed.

To locate these elements, Rhapsody uses a set of regular expressions.

The regular expressions used for this search are defined by the property RefactorRenameRegularExpression.

Different values are used for this property, depending on what type of element you are renaming. For example, you will see that this property has a different value under Class than it does under Attribute. For some types of elements, Rhapsody uses a more generic expression, taken from the value of the property under ModelElement.

In general, the value of this property is a regular expression that consists of a number of "Or" components. For the renaming to work properly, each such component must contain three groupings, with the middle grouping using the variable \$keyword, which represents the name of the element that is being renamed. In some cases, the expression takes this information from the value of another property rather than using the \$keyword variable.

*Default = ([^A-Za-z0-9\_]/^)([extract\_itex]keyword)([/extract\_itex])*

## SimplifyAnnotations

If you are using the Rational Rhapsody customizable code generation mechanism, the `SimplifyAnnotations` property can be used to change the way annotations are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- `None` - Annotations are ignored.
- `Copy` - Annotations are copied from the original to the simplified model. They are not modified in any way.
- `Default` - Uses the standard simplification for Annotations, as defined in Rational Rhapsody.
- `ByUser` - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- `ByUserPostDefault` - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for Annotations has been applied.
- `CodeUpdateAnnotations` - Used in the CodeCentric settings in order to minimize the Rational Rhapsody annotations generated in code, limiting them to special cases such as animation. Note that if you try to use this value when not using the CodeCentric settings, roundtripping may not work correctly.

*Default = "Default"*

## **SimplifyDescription**

If you are using the Rational Rhapsody customizable code generation mechanism, the `SimplifyDescription` property can be used to change the way Descriptions are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- `None` - Descriptions are ignored.
- `Copy` - Description are copied from the original to the simplified model. They are not modified in any way.
- `Default` - Uses the standard simplification for Descriptions, as defined in Rational Rhapsody.
- `ByUser` - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- `ByUserPostDefault` - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for Descriptions has been applied.

*Default = "Default"*

## **SimplifyExternal**

If you are using the Rational Rhapsody customizable code generation mechanism, the `SimplifyExternal` property can be used to change the way that code is generated for external elements that are not actually part of the model, for example, base classes for classes in the model, by changing the way that such elements are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - The elements are ignored.
- Default - Uses the standard simplification for these elements, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for these element has been applied.

*Default = "Default"*

## SimplifyInstrumentation

If you are using the Rational Rhapsody customizable code generation mechanism, the SimplifyInstrumentation property can be used to customize the generation of instrumentation code (such as animation) by changing the way instrumentation is handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - Instrumentation is ignored.
- Default - Uses the standard simplification for instrumentation, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for instrumentation has been applied.

*Default = "Default"*

## SimplifyProperties

If you are using the Rational Rhapsody customizable code generation mechanism, the SimplifyProperties property can be used to customize the way that overridden properties affect code generation by changing the way that Rational Rhapsody handles these properties when it transforms the model into a simplified model.

The property can take any of the following values:

- None - Overridden properties are ignored.
- Copy - Overridden properties are copied from the original to the simplified model. Their effect are not modified in any way.
- Default - Uses the standard simplification for overridden properties, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for overridden properties has been applied.

*Default = "Default"*

## SimplifyStandardOperations

If you are using the Rational Rhapsody customizable code generation mechanism, the `SimplifyStandardOperations` property can be used to customize the way that code is generated for operations defined by using the "StandardOperation" properties by changing the way that Rational Rhapsody handles such operations when it transforms the model into a simplified model.

The property can take any of the following values:

- None - Standard Operations are ignored.
- Copy - Standard Operations are copied from the original to the simplified model. They are not modified in any way.
- Default - Uses the standard simplification for Standard Operations, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for Standard Operations has been applied.

*Default = "Default"*

## SimplifyWebify

If you are using the Rational Rhapsody customizable code generation mechanism, the `SimplifyWebify` property can be used to customize the generation of Webify code by changing the way Webify is handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - Webify is ignored.
- Default - Uses the standard simplification for Webify, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for Webify has been applied.

*Default = "Default"*

## TagsToConsider

By default, tags in your model are not copied to the simplified model for the purpose of code generation.

This can be problematic in cases where you have used the conditional property feature to include the value of specific tags in code-generation properties.

In such cases, you can use the property `TagsToConsider` to specify the names of tags that should be copied to the simplified model.



The value of the property should be a comma-separated list of tag names.

*Default = Blank*

## MSVC

The MSVC metaclass contains the Environment settings (compiler, framework libraries, and so on) for the MSVC environment.

### AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)/LangC/osconfig/WIN32*

### AdditionalReservedWords

The AdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = \_\_asm \_\_finally naked \_\_based \_\_inline \_\_single\_inheritance \_\_cdecl \_\_int8 \_\_stdcall  
\_\_declspec \_\_int16 dllexport \_\_int32 \_\_try dllimport \_\_int64 \_\_virtual\_inheritance \_\_except \_\_leave  
\_\_fastcall \_\_multiple\_inheritance*

### AnimIncludeDirectories

The `AnimIncludeDirectories` property is used to specify the directories that must be referenced in the makefile for projects where Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with `INST_INCLUDES`.

*Default = \$(INCLUDE\_QUALIFIER)\$(OMROOT)\LangC\iom*

## AnimInstLibs

The `AnimInstLibs` property is used to specify the static libraries that are required when Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with `INST_LIBS`.

*Default = \$(OMROOT)\LangC\lib\\$(LIB\_PREFIX)aomanim\$(LIB\_POSTFIX)\$(LIB\_EXT)*

## AnimOxfLibs

The `AnimOxfLibs` property is used to specify the framework libraries that are required when Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with `OXF_LIBS`.

*Default = \$(OMROOT)\LangC\lib\\$(LIB\_PREFIX)oxfinst\$(LIB\_POSTFIX)\$(LIB\_EXT)  
\$(OMROOT)\LangC\lib\\$(LIB\_PREFIX)omcomappl\$(LIB\_POSTFIX)\$(LIB\_EXT) winmm.lib*

## AnimPreprocessor

The `AnimPreprocessor` property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with `INST_FLAGS`.

*Default = \$(DEFINE\_QUALIFIER)OMANIMATOR*

## BuildArgumentsInIDE

The `BuildArgumentsInIDE` property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the `[lang]_CG::[environment]::BuildCommandInIDE` property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the `BuildInIDE` property is set to `True`.

*Default = Blank*

## BuildCommandInIDE

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration. To change this property, use the Configuration window in the browser - do not change it by using the Properties window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link.

*Default = Debug*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

```
Default = "$OMROOT/etc/Executer.exe" "\"$OMROOT\"\\etc\\msvcmake.bat msbuild.mak build $CPU  
$IDEVersion \"LIB_PREFIX=$(LibPrefix)\" \"USE_PDB=FALSE\"  
\"BUILD_SET=$BuildCommandSet\" \"\"
```

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = False*

## CompileSwitches

The CompileSwitches property specifies the compiler switches.

```
Default = /I . /I $OMDefaultSpecificationDirectory /I $(OMROOT)\LangC /I $(OMROOT)\LangC\oxf  
/nologo /W3 $(ENABLE_EH) $(CRT_FLAGS) $OMCPPCompileCommandSet /D "_AFXDLL" /D  
"WIN32" /D "_CONSOLE" /D "_MBCS" /D "_WINDOWS" $(INST_FLAGS) $(INCLUDE_PATH)
```

*\$(INST\_INCLUDES) /c*

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = \$(CREATE\_OBJ\_DIR) \$(CC) \$OMFileCPPCompileSwitches /Fo"\$OMFileObjPath"  
"\$OMFileImpPath"*

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*Default = /Zi /Od /D "\_DEBUG" \$(LIBCRT\_FLAG) /Fd"\$\$(TARGET\_NAME)"*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*Default = /Ox /D"NDEBUG" \$(LIBCRT\_FLAG) /Fd"\$\$(TARGET\_NAME)"*

## CPU

The CPU property is used to specify the CPU of the target environment.

When building applications or framework libraries for MSVC9 environments, Rhapsody uses a variable called \$CPU which represents the value provided for the CPU property for these environments. This variable is provided as a parameter to the msvc9make.bat batch file, and is also used in the makefiles that are generated.

When building 32-bit applications, use the value x86 for this property.

When building 64-bit applications, use the value x64 for this property.

*Default = x86*

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

For example, the following dependency rule lists the file dependencies for a Windows application with a

GUI, including bitmaps, icons, and resource files: \$OMFileObjPath : \$OMFileImpPath "\*.bmp" "\*.ico" "\*.rc2"

*Default = \$OMFileObjPath : \$OMFileImpPath \$OMFileSpecPath \$OMFileDependencies*

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction an IDE such as Eclipse, you can use the EnableDebugIntegrationWithIDE property to specify whether or not the IDE debugger is to be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to Checked, the IDE debugger is used.

*Default = False*

## EndApplicationCode

When the main function ends, the system's exit() function is called automatically. The system function calls the destructors of framework instances in the order it determines. The EndApplicationCode property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an endApplication() function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to endApplication(), it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The EntryPoint property specifies the name of the main program for a given environment.

*Default = main*

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. The ErrorMessageTokensFormat property defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. The ErrorMessageTokens property has three parameters, each with an integer value:

- TotalNumberOfTokens - The number of tokens in the regular expression
- FileTokenPosition - The position of the file name token in the expression
- LineTokenPosition - The position of the line number token in the expression

*Default = ToTalNumberOfTokens=3,FileTokenPosition=1,LineTokenPosition=2*

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the C\_CG::<Environment>::ExeName property plus the value of the ExeExtension property.

*Default = .exe*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the [lang]\_CG::[environment]::ExeExtension property.

*Default = Blank*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements. The file inclusions are generated in the makefile.

*Default = \$OMSpecIncludeInElements \$OMImpIncludeInElements*

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile.

Note: If you select Release Build Set, these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

*Default = \$(OMROOT)\LangC\lib\\$(LIB\_PREFIX)WebComponents\$(LIB\_POSTFIX)\$(LIB\_EXT),  
\$(OMROOT)\lib\\$(LIB\_PREFIX)WebServices\$(LIB\_POSTFIX)\$(LIB\_EXT), ws2\_32\$(LIB\_EXT)*

## IDEVersion

When you select one of the provided Microsoft environments for a configuration in your model, you use the IDEVersion property to tell Rational Rhapsody which version of MS Visual Studio you are using, 2008 or 2010.

The value of this property is passed as a parameter to the Build Framework command, and to the command used to build your application. The value of the property is also used in properties that affect the content of the generated makefile.

The possible values are:

- VC9 - to specify that you are using Visual Studio 2008
- VC10 - to specify that you are using Visual Studio 2010
- VC11 - to specify that you are using Visual Studio 2012

*Default = VC10*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment.

*Default = .c*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*Default = !INCLUDE*

## InvokeAnimatedExecutable

The AnimationPortRange value in the rhapsody.ini file makes it possible to use animation with multiple instances of Rhapsody running simultaneously.

Ordinarily, when you run an application with animation, the executable is launched using the command specified by the InvokeExecutable property. However, if you have specified a value greater than zero for AnimationPortRange in your rhapsody.ini file, the executable is launched using the command specified by the InvokeAnimatedExecutable property. By default, the value of InvokeAnimatedExecutable includes a reference to the port to use for animation.

*Default = \$executable -port \$port*

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*Default = "\$executable"*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, include the name of the property preceded by \$. As shown in the following example (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

```
Default = "$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\msvc9make.bat $makefile $maketarget  
$CPU \" "
```

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = False*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .lib*

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

*Default = Blank*

## LinkerFlags



The LinkerFlags property allows you to define linker flags. The value of the property is inserted into the value of the LinkSwitches property. In the generated makefile, you can see the value of this property in the line that begins with LINK\_FLAGS=.

*Default = Blank*

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

*Default = Blank*

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

*Default = \$OMLinkCommandSet \$LinkerFlags /NOLOGO*

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the [lang]\_CG::[environment]::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*Default = .mak*

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration. For example, the default content of the target type section of a C++ makefile for the Microsoft environment is as follows: ##### Target type (Debug/Release)

```
#####  
CXXCPPCompileDebug=$OMCXXCPPCompileDebug CXXCPPCompileRelease=$OMCXXCPPCompileRelease  
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease BuildSet=$OMBuildSet  
SUBSYSTEM=$OMSubSystem COM=$OMCOM RPFramWorkDll=$OMRPFramWorkDll  
ConfigurationCXXCPPCompileSwitches= $OMReusableStatechartSwitches  
$OMConfigurationCXXCPPCompileSwitches !IF "$ (RPFramWorkDll)" == "True"  
ConfigurationCXXCPPCompileSwitches= $(ConfigurationCXXCPPCompileSwitches) /D "FRAMEWORK_DLL"  
!ENDIF !IF "$ (COM)" == "True" SUBSYSTEM=/SUBSYSTEM:windows !ENDIF
```

**Compilation Flags** The compilation flags section of the makefile contains the default compilation flags stored in the CompileSwitches property. For example, the default content of the compilation flags section of a C++ makefile for the Microsoft environment is as follows: ##### Compilation flags

```
#####  
INCLUDE_QUALIFIER=/I LIB_PREFIX=MS
```

**Commands Definitions** The commands definition section of the makefile specifies programs to execute from the makefile. For example, the default commands definition section of a C++ makefile for the Microsoft environment is as follows: ##### Commands definition #####

```
##### RMDIR = rmdir LIB_CMD=link.exe -lib  
LINK_CMD=link.exe LIB_FLAGS=$OMConfigurationLinkSwitches  
LINK_FLAGS=$OMConfigurationLinkSwitches $(SUBSYSTEM) / MACHINE:I386
```

**Generated Macros** The generated macros section of the makefile contains a variable that expands to the Rational Rhapsody-generated macros in the makefile. For example: ##### Generated macros

```
##### $OMContextMacros  
OBJ_DIR=$OMObjectsDir !IF "$ (OBJ_DIR)" != "" CREATE_OBJ_DIR=if not exist $(OBJ_DIR) mkdir  
$(OBJ_DIR) CLEAN_OBJ_DIR= if exist $(OBJ_DIR) $(RMDIR) $(OBJ_DIR) !ELSE  
CREATE_OBJ_DIR= CLEAN_OBJ_DIR= !ENDIF
```

The \$OMContextMacros keyword expands several macros in the makefile. Each makefile macro has its own keyword. You can use these keywords separately to customize the makefile. The \$OMContextMacros variable modifies target-specific variables. Replace the \$OMContextMacros line in the MakeFileContent property with the following: FLAGSFILE=\$OMFlagsFile

```
RULESFILE=$OMRulesFile OMROOT=$OMROOT C_EXT=$OMImplExt H_EXT=$OMSpecExt  
OBJ_EXT=$OMObjExt EXE_EXT=$OMExeExt LIB_EXT=$OMLibExt  
INSTRUMENTATION=$OMInstrumentation TIME_MODEL=$OMTimeModel  
TARGET_TYPE=$OMTargetType TARGET_NAME=$OMTargetName $OMAllDependencyRule  
TARGET_MAIN=$OMTargetMain LIBS=$OMLibs INCLUDE_PATH=$OMIncludePath  
ADDITIONAL_OBJS=$OMAdditionalObjs OBJS= $OMObjs
```

**Predefined Macros** The predefined macros section of the makefile contains other macros than the Rational Rhapsody-generated macros specified in the generated macros section. For example, the default predefined macros section of a C++ makefile for the Microsoft environment is as follows:

```
##### Predefined macros #####  
##### $(OBJS) : $(INST_LIBS)  
$(OXF_LIBS) LIB_POSTFIX= !IF "$ (BuildSet)" == "Release" LIB_POSTFIX=R !ENDIF !IF  
"$ (TARGET_TYPE)" == "Executable" LinkDebug=$(LinkDebug) /DEBUG  
LinkRelease=$(LinkRelease) /OPT:NOREF !ELSEIF "$ (TARGET_TYPE)" == "Library"  
LinkDebug=$(LinkDebug) /DEBUGTYPE:CV !ENDIF !IF "$ (INSTRUMENTATION)" == "Animation"  
INST_FLAGS=/D "OMANIMATOR" INST_INCLUDES=/I $(OMROOT)\LangCpp\aoom /I  
$(OMROOT)\LangCpp\tom !IF "$ (RPFramWorkDll)" == "True" INST_LIBS=
```

```

OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxanimdll$(LIB_POSTFIX) $(LIB_EXT)
!ELSE INST_LIBS= $(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomanim$(LIB_POSTFIX) $(LIB_EXT)
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfirst$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)" == "Tracing" INST_FLAGS=/D
"OMTRACER" INST_INCLUDES=/I $(OMROOT)\LangCpp\iom /I $(OMROOT)\LangCpp\tom !IF
"$(RPFrameWorkDll)" == "True" INST_LIBS= OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfracedll$(LIB_POSTFIX) $(LIB_EXT) !ELSE
INST_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)tomtrace$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomtrace$(LIB_POSTFIX) $(LIB_EXT) OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfirst$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)" == "None" INST_FLAGS=
INST_INCLUDES= INST_LIBS= !IF "$(RPFrameWorkDll)" == "True"
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfdll$(LIB_POSTFIX) $(LIB_EXT) !ELSE
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB= !ELSE !ERROR An invalid Instrumentation $(INSTRUMENTATION) is specified.
!ENDIF

```

Generated Dependencies The generated dependencies section of the makefile contains a variable that expands to Rational Rhapsody-generated dependencies and compilation instructions. For example, the generated dependencies section of a C++ makefile for the Microsoft environment is as follows:

```

##### Generated dependencies #####
##### $OMContextDependencies
$OMFileObjPath : $OMMainImplementationFile $(OBJS) $(CPP) $(ConfigurationCPPCompileSwitches)
/Fo"$OMFileObjPath" $OMMainImplementationFile

```

Linking Instructions The linking instructions section of the makefile contains the predefined linking instructions. For example, the default linking instructions section of a C++ makefile for the Microsoft environment is as follows:

```

##### Linking instructions #####
#####
$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS) $OMFileObjPath
$OMMakefileName $OMModelLibs @echo Linking $(TARGET_NAME)$(EXE_EXT) $(LINK_CMD)
$OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \ $(OXF_LIBS) \
$(SOCK_LIB) \ $(LINK_FLAGS) /out:$(TARGET_NAME)$(EXE_EXT)
$(TARGET_NAME)$(LIB_EXT) : $(OBJS) $(ADDITIONAL_OBJS) $OMMakefileName @echo
Building library $@ $(LIB_CMD) $(LIB_FLAGS) /out:$(TARGET_NAME)$(LIB_EXT) $(OBJS)
$(ADDITIONAL_OBJS) clean: @echo Cleanup $OMCleanOBJS if exist $OMFileObjPath erase
$OMFileObjPath if exist *$(OBJ_EXT) erase *$(OBJ_EXT) if exist $(TARGET_NAME).pdb erase
$(TARGET_NAME).pdb if exist $(TARGET_NAME)$(LIB_EXT) erase
$(TARGET_NAME)$(LIB_EXT) if exist $(TARGET_NAME).ilk erase $(TARGET_NAME).ilk if exist
$(TARGET_NAME)$(EXE_EXT) erase $(TARGET_NAME)$(EXE_EXT) $(CLEAN_OBJ_DIR)

```

```

##### Target type (Debug/Release) #####
#####
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease BuildSet=$OMBuildSet
ConfigurationCPPCompileSwitches=$OMConfigurationCPPCompileSwitches #####
Compilation flags #####
##### INCLUDE_QUALIFIER=/I
LIB_PREFIX=$LibPrefix !IF "$(UseLIBCMT)" == "True" MT_PREFIX=MT LIBCRT_FLAG=/MT
!ELSE MT_PREFIX= LIBCRT_FLAG=/MD !ENDIF CRT_FLAGS=/D
"_CRT_SECURE_NO_DEPRECATED" /D "_CRT_SECURE_NO_WARNINGS" ENABLE_EH=/EHsc
WINMM_LIB=winmm.lib ##### Distributed Events Marcos & Flags #####

```

```

#####
DISTRIBUTION=$MultipleAddressSpaces DISTRIBUTION_PREFIX= DOX_LIBS= DOX_FLAGS=
##### Commands definition #####
##### RMDIR = rmdir
LIB_CMD=link.exe -lib LINK_CMD=link.exe LIB_FLAGS=$OMConfigurationLinkSwitches
LINK_FLAGS=$OMConfigurationLinkSwitches /SUBSYSTEM:console /MACHINE:$CPU
##### Generated macros #####
##### SOMContextMacros
OBJ_DIR=$OMObjectsDir !IF "$ (OBJ_DIR)"!=" CREATE_OBJ_DIR=if not exist $(OBJ_DIR) mkdir
$(OBJ_DIR) CLEAN_OBJ_DIR= if exist $(OBJ_DIR) $(RMDIR) $(OBJ_DIR) !ELSE
CREATE_OBJ_DIR= CLEAN_OBJ_DIR= !ENDIF ##### Predefined macros
#####
##### !IF "$ (OBJS)" !=
"" $(OBJS) : $(FLAGSFILE) $(RULESFILE) $(INST_LIBS) $(OXF_LIBS) !ENDIF LIB_POSTFIX=
!IF "$ (BuildSet)"=="Release" LIB_POSTFIX=R !ENDIF !IF "$ (DISTRIBUTION)" == "True"
DISTRIBUTION_PREFIX=distrib_
DOX_LIBS=$(OMROOT)\LangC\lib\$(LIB_PREFIX)dox$(LIB_POSTFIX)$(LIB_EXT) kernel32.lib
user32.lib gdi32.lib winspool.lib comdlg32.lib advapi32.lib shell32.lib ole32.lib oleaut32.lib uuid.lib
odbc32.lib odbccp32.lib kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib advapi32.lib
shell32.lib ole32.lib oleaut32.lib uuid.lib odbc32.lib odbccp32.lib DOX_FLAGS=/D
"RIC_DISTRIBUTED_SYSTEM" !ELSE DISTRIBUTION_PREFIX= DOX_LIBS= !ENDIF !IF
"$ (TARGET_TYPE)" == "Executable" LinkDebug=$(LinkDebug) /DEBUG
LinkRelease=$(LinkRelease) /OPT:NOREF !ELSEIF "$ (TARGET_TYPE)" == "Library"
LinkDebug=$(LinkDebug) !ENDIF !IF "$ (INSTRUMENTATION)" == "Animation" INST_FLAGS=/D
"OMANIMATOR" INST_INCLUDES=/I $(OMROOT)\LangC\om INST_LIBS=
$(OMROOT)\LangC\lib\$(LIB_PREFIX)$(DISTRIBUTION_PREFIX)aomanim$(LIB_POSTFIX)$(LIB_EXT)
OXF_LIBS=$(OMROOT)\LangC\lib\$(LIB_PREFIX)$(DISTRIBUTION_PREFIX)oxf$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangC\lib\$(LIB_PREFIX)omcomappl$(LIB_POSTFIX)$(LIB_EXT)
SOCK_LIB=wsock32.lib !ELSEIF "$ (INSTRUMENTATION)" == "Tracing" INST_FLAGS=/D
"OMTRACER" INST_INCLUDES=/I $(OMROOT)\LangC\om /I $(OMROOT)\LangC\pp\om
INST_LIBS=
$(OMROOT)\LangC\lib\$(LIB_PREFIX)$(DISTRIBUTION_PREFIX)aomtrace$(LIB_POSTFIX)$(LIB_EXT)
OXF_LIBS=
$(OMROOT)\LangC\lib\$(LIB_PREFIX)$(DISTRIBUTION_PREFIX)oxf$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangC\lib\$(LIB_PREFIX)omcomappl$(LIB_POSTFIX)$(LIB_EXT) ## C++ Libraries to
support link to C++ TOM ## INST_LIBS= $(INST_LIBS)
$(OMROOT)\LangC\pp\lib\$(LIB_PREFIX)tomtraceRiC$(LIB_POSTFIX)$(LIB_EXT) OXF_LIBS=
$(OXF_LIBS) $(OMROOT)\LangC\pp\lib\$(LIB_PREFIX)oxf$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangC\pp\lib\$(LIB_PREFIX)omcomappl$(LIB_POSTFIX)$(LIB_EXT)
SOCK_LIB=wsock32.lib !ELSEIF "$ (INSTRUMENTATION)" == "None" INST_FLAGS=
INST_INCLUDES= INST_LIBS=
OXF_LIBS=$(OMROOT)\LangC\lib\$(LIB_PREFIX)$(DISTRIBUTION_PREFIX)oxf$(LIB_POSTFIX)$(LIB_EXT)
SOCK_LIB= !ELSE !ERROR An invalid Instrumentation $(INSTRUMENTATION) is specified.
!ENDIF ConfigurationCPPCompileSwitches=$(ConfigurationCPPCompileSwitches) $(DOX_FLAGS)
##### Generated dependencies #####
#####
SOMContextDependencies $OMFileObjPath : $OMMainImplementationFile $(OBJS) $(FLAGSFILE)
$(RULESFILE) $(CC) $(ConfigurationCPPCompileSwitches) /Fo"$OMFileObjPath"
$OMMainImplementationFile ##### Linking instructions
#####
#####
$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS) $OMFileObjPath
$OMMakefileName $OMModelLibs @echo Linking $(TARGET_NAME)$(EXE_EXT) $(LINK_CMD)
$OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \ $(OXF_LIBS) \
$(DOX_LIBS) \ $(SOCK_LIB) \ $(WINMM_LIB) \ $(LINK_FLAGS)

```

```

/out:$$(TARGET_NAME)$(EXE_EXT) if exist $(TARGET_NAME)$(EXE_EXT).manifest mt.exe
-manifest $(TARGET_NAME)$(EXE_EXT).manifest
-outputresource:$(TARGET_NAME)$(EXE_EXT);1 $(TARGET_NAME)$(LIB_EXT) : $(OBJS)
$(ADDITIONAL_OBJS) $OMMakefileName @echo Building library $@ $(LIB_CMD) $(LIB_FLAGS)
/out:$$(TARGET_NAME)$(LIB_EXT) $(OBJS) $(ADDITIONAL_OBJS) $(LIBS) clean: @echo Cleanup
$OMCleanOBJS if exist $OMFileObjPath erase $OMFileObjPath if exist *$(OBJ_EXT) erase
*$(OBJ_EXT) if exist $(TARGET_NAME).pdb erase $(TARGET_NAME).pdb if exist
$(TARGET_NAME)$(LIB_EXT) erase $(TARGET_NAME)$(LIB_EXT) if exist
$(TARGET_NAME).ilk erase $(TARGET_NAME).ilk if exist $(TARGET_NAME)$(EXE_EXT) erase
$(TARGET_NAME)$(EXE_EXT) if exist $(TARGET_NAME)$(EXE_EXT).manifest erase
$(TARGET_NAME)$(EXE_EXT).manifest $(CLEAN_OBJ_DIR)

```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the [lang]\_CG::[environment]::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*Default = Blank*

## NoneIncludeDirectories

The NoneIncludeDirectories property is used to specify the directories that must be referenced in the makefile for projects where Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with INST\_INCLUDES.

*Default = Blank*

## NoneInstLibs

The NoneInstLibs property is used to specify the static libraries that are required when Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with INST\_LIBS.

*Default = Blank*

## NoneOxfLibs

The NoneOxfLibs property is used to specify the framework libraries that are required when Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with OXF\_LIBS.

*Default = \$(OMROOT)\LangC\lib\\$(LIB\_PREFIX)oxf\$(LIB\_POSTFIX)\$(LIB\_EXT) winmm.lib*

## NonePreprocessor

The NonePreprocessor property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with INST\_FLAGS.

*Default = Blank*

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## ObjCleanCommand

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default = if exist \$OMFileObjPath erase \$OMFileObjPath*

## ObjectName

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = Blank*

## ObjectsDirectory

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = Blank*

## ObjExtension

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*Default = .obj*

## OSFileSystemCaseSensitive



The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = False*

## ParseErrorDescript

The ParseErrorDescript property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = ([^()]+)[([0-9]+)] [:] (error/warning/fatal error) (.\*)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning)

Along with the ErrorMessageTokensFormat property, the ParseErrorMessage property specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = ([^()]+)[([0-9]+)] [:] (error/warning/fatal error)*

## ParseMakeError

The ParseMakeError property is used to define a regular expression that represents the format of make process or linker error messages. This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = (NMAKE/LINK)(.\*)(fatal error)*

## ParseSeverityError

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = [:] (error/fatal error)*

## ParseSeverityWarning

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = [:] (warning)*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation.

*Default = \*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = True*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running the product, whereas the target is the machine running the application. To run remotely, the UseRemoteHost property must be set to Checked.

If the UseRemoteHost property is set to Checked and the RemoteHost property is blank, the current host name is used for the remote host. You can use this as a workaround if you have problems running animated applications on Windows 95.

You can leave the RemoteHost property blank if both the application and Rational Rhapsody are running on the same machine.

*Default = Blank*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*Default = .h*

## TraceIncludeDirectories

The TraceIncludeDirectories property is used to specify the directories that must be referenced in the makefile for projects where Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with INST\_INCLUDES.

*Default = \$(INCLUDE\_QUALIFIER)\$(OMROOT)\LangC\ao  
\$(INCLUDE\_QUALIFIER)\$(OMROOT)\LangCpp\to*



## TraceInstLibs

The TraceInstLibs property is used to specify the static libraries that are required when Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with INST\_LIBS.

*Default = \$(OMROOT)\LangC\lib\\$(LIB\_PREFIX)aomtrace\$(LIB\_POSTFIX)\$(LIB\_EXT) \$(INST\_LIBS) \$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)tomtraceRiC\$(LIB\_POSTFIX)\$(LIB\_EXT)*

## TraceOxfLibs

The TraceOxfLibs property is used to specify the framework libraries that are required when Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with OXF\_LIBS.

*Default = \$(OMROOT)\LangC\lib\\$(LIB\_PREFIX)oxfinst\$(LIB\_POSTFIX)\$(LIB\_EXT)  
\$(OMROOT)\LangC\lib\\$(LIB\_PREFIX)omcomappl\$(LIB\_POSTFIX)\$(LIB\_EXT)  
\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)oxf\$(LIB\_POSTFIX)\$(LIB\_EXT)  
\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)omcomappl\$(LIB\_POSTFIX)\$(LIB\_EXT) winmm.lib*

## TracePreprocessor

The TracePreprocessor property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with INST\_FLAGS.

*Default = \$(DEFINE\_QUALIFIER)OMTRACER*

## UpdateBuildSettingsInIDE

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = True*

## UseLIBCMT

If you want your application to use the statically-linked library libcmt.lib, rather than the dynamically-linked library msvcrt.lib, set the value of the property UseLIBCMT to True.

If the property is set to True, the makefile will include the option /MT. Otherwise, the makefile will include the option /MD.

*Default = False*

## UseNewBuildOutputWindow

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the CG::General::ShowLogViewAfterBuild property to True.

*Default = True*

## UseRemoteHost

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = False*

## MSVCDLL

The MSVCDLL metaclass contains the Environment settings (such as compiler switches and framework libraries) for the MSVCDLL environment.

## AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default = \$(OMROOT)/LangC/osconfig/WIN32*

## AdditionalReservedWords

The `AdditionalReservedWords` property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = \_\_asm \_\_finally naked \_\_based \_\_inline \_\_single\_inheritance \_\_cdecl \_\_int8 \_\_stdcall \_\_declspec \_\_int16 dllexport \_\_int32 \_\_try dllimport \_\_int64 \_\_virtual\_inheritance \_\_except \_\_leave \_\_fastcall \_\_multiple\_inheritance*

## **AnimIncludeDirectories**

The `AnimIncludeDirectories` property is used to specify the directories that must be referenced in the makefile for projects where Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with `INST_INCLUDES`.

*Default = \$(INCLUDE\_QUALIFIER)\$(OMROOT)\\LangC\\aom*

## **AnimInstLibs**

The `AnimInstLibs` property is used to specify the static libraries that are required when Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with `INST_LIBS`.

*Default = \$(OMROOT)\\LangC\\lib\\\$(LIB\_PREFIX)aomanim\$(LIB\_POSTFIX)\$(LIB\_EXT)*

## **AnimOxfLibs**

The `AnimOxfLibs` property is used to specify the framework libraries that are required when Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with `OXF_LIBS`.

*Default = \$(OMROOT)\\LangC\\lib\\\$(LIB\_PREFIX)oxfinst\$(LIB\_POSTFIX)\$(LIB\_EXT)  
\$(OMROOT)\\LangC\\lib\\\$(LIB\_PREFIX)omcomappl\$(LIB\_POSTFIX)\$(LIB\_EXT) winmm.lib*

## **AnimPreprocessor**

The `AnimPreprocessor` property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with `INST_FLAGS`.

*Default = \$(DEFINE\_QUALIFIER)OMANIMATOR*

## BuildArgumentsInIDE

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the [lang]\_CG::[environment]::BuildCommandInIDE property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandInIDE

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration. To change this property, use the Configuration window in the browser - do not change it by using the Properties window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link.

*Default = Debug*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\"etc\msvcmake.bat msbuild.mak build \$CPU \$IDEVersion \"LIB\_PREFIX=\$(LibPrefix)\" \"USE\_PDB=FALSE\" \"BUILD\_SET=\$BuildCommandSet\" \"\"*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = False*

## CompilerFlags

The CompilerFlags property is used to specify additional compilation options. The value of this property is included in the value of the CPPCompileSwitches property.

When using the integration with Visual Studio, the options specified with this property are included in the settings for the Visual Studio project.

*Default = /EHsc*

## CompileSwitches

The CompileSwitches property specifies the compiler switches.

*Default = /I . /I \$OMDefaultSpecificationDirectory /I \$(OMROOT)\LangC /I \$(OMROOT)\LangC\oxf /nologo /W3 \$(ENABLE\_EH) \$(CRT\_FLAGS) \$OMCPPCompileCommandSet /D \"\_AFXDLL\" /D \"WIN32\" /D \"\_CONSOLE\" /D \"\_MBCS\" /D \"\_WINDOWS\" \$(INST\_FLAGS) \$(INCLUDE\_PATH) \$(INST\_INCLUDES) /c*

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = \$(CREATE\_OBJ\_DIR) \$(CC) \$OMFileCPPCompileSwitches /Fo\"\$OMFileObjPath\" \"\$OMFileImpPath\"*

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*Default = /Zi /Od /D \"\_DEBUG\" \$(LIBCRT\_FLAG)d /Fd\"\$(TARGET\_NAME)\"*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*Default = /Ox /D\ "NDEBUG\ " \$(LIBCRT\_FLAG) /Fd\ "\$(TARGET\_NAME)\ "*

## CPU

The CPU property is used to specify the CPU of the target environment.

When building applications or framework libraries for MSVC9 environments, Rhapsody uses a variable called \$CPU which represents the value provided for the CPU property for these environments. This variable is provided as a parameter to the msvc9make.bat batch file, and is also used in the makefiles that are generated.

When building 32-bit applications, use the value x86 for this property.

When building 64-bit applications, use the value x64 for this property.

*Default = x64*

## DEFExtension

The DEFExtension property specifies the extension for DLL definition files.

*Default = .def*

## DefinedSymbols

The DefinedSymbols property is used to specify preprocessor directives. The value of this property is included in the value of the CPPCompileSwitches property.

When using the integration with Visual Studio, the preprocessor directives specified with this property are included in the preprocessor-related settings for the Visual Studio project.

*Default = \$(DEFINE\_QUALIFIER)\_CRT\_SECURE\_NO\_DEPRECATED  
\$(DEFINE\_QUALIFIER)\_CRT\_SECURE\_NO\_WARNINGS*

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: \$OMFileObjPath : \$OMFileImpPath "\*.bmp" "\*.ico"

*"\*.rc2"*

*Default = \$OMFileObjPath : \$OMFileImpPath \$OMFileSpecPath \$OMFileDependencies*

## **DllExtension**

The DllExtension property specifies the extension for DLL files.

*Default = .dll*

## **EnableDebugIntegrationWithIDE**

When you use Rational Rhapsody in conjunction an IDE such as Eclipse, you can use the EnableDebugIntegrationWithIDE property to specify whether or not the IDE debugger is to be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to Checked, the IDE debugger is used.

*Default = False*

## **EndApplicationCode**

When the main function ends, the system's exit() function is called automatically. The system function calls the destructors of framework instances in the order it determines. The EndApplicationCode property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an endApplication() function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to endApplication(), it should come after any additional code that you have specified.

*Default = Blank*

## **EntryPoint**

The EntryPoint property specifies the name of the main program for a given environment.

*Default = main*

## **ErrorMessageTokensFormat**

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. The ErrorMessageTokensFormat property defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. The

ErrorMessageTokens property has three parameters, each with an integer value:

- TotalNumberOfTokens - The number of tokens in the regular expression
- FileTokenPosition - The position of the file name token in the expression
- LineTokenPosition - The position of the line number token in the expression

*Default = TotalNumberOfTokens=3,FileTokenPosition=1,LineTokenPosition=2*

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the C\_CG::<Environment>::ExeName property plus the value of the ExeExtension property.

*Default = .exe*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the [lang]\_CG::[environment]::ExeExtension property.

*Default = Blank*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements. The file inclusions are generated in the makefile.

*Default = \$OMSpecIncludeInElements \$OMImpIncludeInElements*

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile.

Note: If you select Release Build Set, these libraries are automatically added with the R postfix (the



Rational Rhapsody convention for framework libraries).

*Default = \$(OMROOT)\LangC\lib\\$(LIB\_PREFIX)WebComponents\$(LIB\_POSTFIX)\$(LIB\_EXT), \$(OMROOT)\lib\\$(LIB\_PREFIX)WebServices\$(LIB\_POSTFIX)\$(LIB\_EXT), ws2\_32\$(LIB\_EXT)*

## IDEVersion

When you select one of the provided Microsoft environments for a configuration in your model, you use the IDEVersion property to tell Rational Rhapsody which version of MS Visual Studio you are using, 2008 or 2010.

The value of this property is passed as a parameter to the Build Framework command, and to the command used to build your application. The value of the property is also used in properties that affect the content of the generated makefile.

The possible values are:

- VC9 - to specify that you are using Visual Studio 2008
- VC10 - to specify that you are using Visual Studio 2010
- VC11 - to specify that you are using Visual Studio 2012

*Default = VC10*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment.

*Default = .c*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*Default = !INCLUDE*

## IncludeDirectories

For environments such as Cygwin, the IncludeDirectories property is used to specify additional directories to search for include files. The value of this property is included in the value of the CPPCompileSwitches property and is then included in the generated makefile.

When using the integration with Visual Studio, the directories specified with this property are added to the "additional include directories" specified for the Visual Studio project.

*Default = \$(INCLUDE\_QUALIFIER). \$(INCLUDE\_QUALIFIER)\$(OMROOT)*

*\$(INCLUDE\_QUALIFIER)\$(OMROOT)\LangC*

## InvokeAnimatedExecutable

The AnimationPortRange value in the rhapsody.ini file makes it possible to use animation with multiple instances of Rhapsody running simultaneously.

Ordinarily, when you run an application with animation, the executable is launched using the command specified by the InvokeExecutable property. However, if you have specified a value greater than zero for AnimationPortRange in your rhapsody.ini file, the executable is launched using the command specified by the InvokeAnimatedExecutable property. By default, the value of InvokeAnimatedExecutable includes a reference to the port to use for animation.

*Default = \$executable -port \$port*

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*Default = \"\$executable\"*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

*"rhapsody\_dir>\share/etc/Executer.exe""dir; dir | more"*

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, include the name of the property preceded by \$. As shown in the following example (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

*"\$OMROOT/etc/Executer.exe" "\"\$OMROOT/etc/vx6make.bat\" \$makefile \$maketarget 6.2 \$BSP gnu"*

*Default = \"\$OMROOT/etc/Executer.exe\" \"\"\"\$OMROOT\\\"etc\\msvcmake.bat \$makefile \$maketarget \$CPU \$IDEVersion \\\"\"\"*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = False*

## **LibExtension**

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .dll*

## **LibPrefix**

The LibPrefix property is used to specify a prefix to add to generic names of runtime libraries. The value of this property is used in references to libraries in the generated makefile.

*Default = MS\$(IDEVersion)\$(CPU)\$(MT\_PREFIX)*

## **LinkDebug**

The LinkDebug property specifies the special link switches used to link in debug mode.

*Default = Blank*

## **LinkerFlags**

The LinkerFlags property allows you to define linker flags. The value of the property is inserted into the value of the LinkSwitches property. In the generated makefile, you can see the value of this property in the line that begins with LINK\_FLAGS=.

*Default = Blank*

## **LinkRelease**

The LinkRelease property specifies the special link switches used to link in release mode.

*Default = Blank*

## **LinkSwitches**

The LinkSwitches property specifies the standard link switches used to link in any mode.

*Default = \$OMLinkCommandSet \$LinkerFlags /NOLOGO*

## **MakeExtension**

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the [lang]\_CG::[environment]::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*Default = .mak*

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration. For example, the default content of the target type section of a C++ makefile for the Microsoft environment is as follows: ##### Target type (Debug/Release)

```
#####  
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease  
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease BuildSet=$OMBuildSet  
SUBSYSTEM=$OMSubSystem COM=$OMCOM RPFramWorkDll=$OMRPFramWorkDll  
ConfigurationCPPCompileSwitches= $OMReusableStatechartSwitches  
$OMConfigurationCPPCompileSwitches !IF "$(RPFramWorkDll)" == "True"  
ConfigurationCPPCompileSwitches= $(ConfigurationCPPCompileSwitches) /D "FRAMEWORK_DLL"  
!ENDIF !IF "$(COM)" == "True" SUBSYSTEM=/SUBSYSTEM:windows !ENDIF
```

**Compilation Flags** The compilation flags section of the makefile contains the default compilation flags stored in the CompileSwitches property. For example, the default content of the compilation flags section of a C++ makefile for the Microsoft environment is as follows: ##### Compilation flags

```
#####  
INCLUDE_QUALIFIER=/I LIB_PREFIX=MS
```

**Commands Definitions** The commands definition section of the makefile specifies programs to execute from the makefile. For example, the default commands definition section of a C++ makefile for the Microsoft environment is as follows: ##### Commands definition #####

```
##### RMDIR = rmdir LIB_CMD=link.exe -lib  
LINK_CMD=link.exe LIB_FLAGS=$OMConfigurationLinkSwitches  
LINK_FLAGS=$OMConfigurationLinkSwitches $(SUBSYSTEM) / MACHINE:I386
```

Generated Macros The generated macros section of the makefile contains a variable that expands to the Rational Rhapsody-generated macros in the makefile. For example: ##### Generated macros ##### SOMContextMacros OBJ\_DIR=\$OMObjectsDir !IF "\$ (OBJ\_DIR)"!=" CREATE\_OBJ\_DIR=if not exist \$(OBJ\_DIR) mkdir \$(OBJ\_DIR) CLEAN\_OBJ\_DIR= if exist \$(OBJ\_DIR) \$(RMDIR) \$(OBJ\_DIR) !ELSE CREATE\_OBJ\_DIR= CLEAN\_OBJ\_DIR= !ENDIF

The \$OMContextMacros keyword expands several macros in the makefile. Each makefile macro has its own keyword. You can use these keywords separately to customize the makefile. The \$OMContextMacros variable modifies target-specific variables. Replace the \$OMContextMacros line in the MakeFileContent property with the following: FLAGSFILE=\$OMFlagsFile RULESFILE=\$OMRulesFile OMROOT=\$OMROOT C\_EXT=\$OMImplExt H\_EXT=\$OMSpecExt OBJ\_EXT=\$OMObjExt EXE\_EXT=\$OMExeExt LIB\_EXT=\$OMLibExt INSTRUMENTATION=\$OMInstrumentation TIME\_MODEL=\$OMTimeModel TARGET\_TYPE=\$OMTargetType TARGET\_NAME=\$OMTargetName \$OMAllDependencyRule TARGET\_MAIN=\$OMTargetMain LIBS=\$OMLibs INCLUDE\_PATH=\$OMIncludePath ADDITIONAL\_OBJS=\$OMAdditionalObjs OBJS= \$OMObjs

Predefined Macros The predefined macros section of the makefile contains other macros than the Rational Rhapsody-generated macros specified in the generated macros section. For example, the default predefined macros section of a C++ makefile for the Microsoft environment is as follows:

```
##### Predefined macros #####
##### $(OBJS) : $(INST_LIBS)
$(OXF_LIBS) LIB_POSTFIX= !IF "$ (BuildSet)"=="Release" LIB_POSTFIX=R !ENDIF !IF
"$(TARGET_TYPE)" == "Executable" LinkDebug=$(LinkDebug) /DEBUG
LinkRelease=$(LinkRelease) /OPT:NOREF !ELSEIF "$(TARGET_TYPE)" == "Library"
LinkDebug=$(LinkDebug) /DEBUGTYPE:CV !ENDIF !IF "$(INSTRUMENTATION)" == "Animation"
INST_FLAGS=/D "OMANIMATOR" INST_INCLUDES=/I $(OMROOT)\LangCpp\om /I
$(OMROOT)\LangCpp\om !IF "$(RPFrameWorkDll)" == "True" INST_LIBS=
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxanimdll$(LIB_POSTFIX) $(LIB_EXT)
!ELSE INST_LIBS= $(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomanim$(LIB_POSTFIX) $(LIB_EXT)
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfinst$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB=wssock32.lib !ELSEIF "$(INSTRUMENTATION)" == "Tracing" INST_FLAGS=/D
"OMTRACER" INST_INCLUDES=/I $(OMROOT)\LangCpp\om /I $(OMROOT)\LangCpp\om !IF
"$(RPFrameWorkDll)" == "True" INST_LIBS= OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfracedll$(LIB_POSTFIX) $(LIB_EXT) !ELSE
INST_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)tomtrace$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomtrace$(LIB_POSTFIX) $(LIB_EXT) OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfinst$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB=wssock32.lib !ELSEIF "$(INSTRUMENTATION)" == "None" INST_FLAGS=
INST_INCLUDES= INST_LIBS= !IF "$(RPFrameWorkDll)" == "True"
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfdll$(LIB_POSTFIX) $(LIB_EXT) !ELSE
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB= !ELSE !ERROR An invalid Instrumentation $(INSTRUMENTATION) is specified.
!ENDIF
```

Generated Dependencies The generated dependencies section of the makefile contains a variable that expands to Rational Rhapsody-generated dependencies and compilation instructions. For example, the generated dependencies section of a C++ makefile for the Microsoft environment is as follows:

```
##### Generated dependencies #####
##### SOMContextDependencies
$OMFileObjPath : $OMMainImplementationFile $(OBJS) $(CPP) $(ConfigurationCPPCompileSwitches)
```

/Fo"\$OMFileObjPath" \$OMMainImplementationFile

**Linking Instructions** The linking instructions section of the makefile contains the predefined linking instructions. For example, the default linking instructions section of a C++ makefile for the Microsoft environment is as follows: ##### Linking instructions #####

```
#####  
$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS) $OMFileObjPath  
$OMMakefileName $OMModelLibs @echo Linking $(TARGET_NAME)$(EXE_EXT) $(LINK_CMD)  
$OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \ $(OXF_LIBS) \  
$(SOCK_LIB) \ $(LINK_FLAGS) /out:$(TARGET_NAME)$(EXE_EXT)  
$(TARGET_NAME)$(LIB_EXT) : $(OBJS) $(ADDITIONAL_OBJS) $OMMakefileName @echo  
Building library $@ $(LIB_CMD) $(LIB_FLAGS) /out:$(TARGET_NAME)$(LIB_EXT) $(OBJS)  
$(ADDITIONAL_OBJS) clean: @echo Cleanup $OMCleanOBJS if exist $OMFileObjPath erase  
$OMFileObjPath if exist *$(OBJ_EXT) erase *$(OBJ_EXT) if exist $(TARGET_NAME).pdb erase  
$(TARGET_NAME).pdb if exist $(TARGET_NAME)$(LIB_EXT) erase  
$(TARGET_NAME)$(LIB_EXT) if exist $(TARGET_NAME).ilk erase $(TARGET_NAME).ilk if exist  
$(TARGET_NAME)$(EXE_EXT) erase $(TARGET_NAME)$(EXE_EXT) $(CLEAN_OBJ_DIR)
```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the [lang]\_CG::[environment]::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*Default = Blank*

## NoneIncludeDirectories

The NoneIncludeDirectories property is used to specify the directories that must be referenced in the makefile for projects where Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with INST\_INCLUDES.

*Default = Blank*

## NoneInstLibs

The NoneInstLibs property is used to specify the static libraries that are required when Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with INST\_LIBS.

*Default = Blank*

## NoneOxfLibs

The NoneOxflibs property is used to specify the framework libraries that are required when Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with OXF\_LIBS.

*Default = \$(OMROOT)\LangC\lib\\$(LIB\_PREFIX)oxf\$(LIB\_POSTFIX)\$(LIB\_EXT) winmm.lib*

## NonePreprocessor

The NonePreprocessor property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with INST\_FLAGS.

*Default = Blank*

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## ObjCleanCommand

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default = if exist \$OMFileObjPath erase \$OMFileObjPath*

## ObjectName

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = Blank*

## ObjectsDirectory

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = Blank*

## ObjExtension

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*Default = .obj*

## OSFileSystemCaseSensitive

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = False*

## ParseErrorDescript

The ParseErrorDescript property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = ([^()]+)([([0-9]+)]) [:] (error/warning/fatal error) (.\*)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning)

Along with the ErrorMessageTokensFormat property, the ParseErrorMessage property specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = ([^()]+)([([0-9]+)]) [:] (error/warning/fatal error)*

## Property ParseErrorMoreInfo

*Default = ^[( )]*

## ParseMakeError

The ParseMakeError property is used to define a regular expression that represents the format of make process or linker error messages. This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = (NMAKE/LINK)(.\*)(fatal error)*

## ParseSeverityError

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should



be displayed alongside the message on the Build tab of the Output window.

*Default = [:] (error/fatal error)*

## **ParseSeverityWarning**

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = [:] (warning)*

## **PathDelimiter**

The PathDelimiter property specifies an alternative path separator for code generation.

*Default = \*

## **QuoteOMROOT**

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = True*

## **RCCompileCommand**

The RCCompileCommand property specifies the compilation command for the resource file.

*Default = \$(RC) /Fo\"\$(TARGET\_MAIN).res\" \$(TARGET\_MAIN)\$OMRCEExtension*

## **RCEExtension**

The RCEExtension property specifies the extension for resource files.

*Default = .rc*

## **RemoteHost**

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running the product, whereas the target is the machine running the application. To run remotely, the UseRemoteHost property must be set to Checked.

If the UseRemoteHost property is set to Checked and the RemoteHost property is blank, the current host name is used for the remote host. You can use this as a workaround if you have problems running

animated applications on Windows 95.

You can leave the RemoteHost property blank if both the application and Rational Rhapsody are running on the same machine.

*Default = Blank*

## **SockLib**

The SockLib property represents the name of the socket library.

For the Cygwin environment, the value entered for this property is used in the generated makefile.

When using the integration with Visual Studio, the value of this property is included in the "additional dependencies" for the Visual Studio project.

*Default = wsock32.lib*

## **SpecExtension**

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*Default = .h*

## **TraceIncludeDirectories**

The TraceIncludeDirectories property is used to specify the directories that must be referenced in the makefile for projects where Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with INST\_INCLUDES.

*Default = \$(INCLUDE\_QUALIFIER)\$(OMROOT)\LangC\|aom  
\$(INCLUDE\_QUALIFIER)\$(OMROOT)\LangCpp\|tom*

## **TraceInstLibs**

The TraceInstLibs property is used to specify the static libraries that are required when Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with INST\_LIBS.

*Default = \$(OMROOT)\LangC\|lib\|\$(LIB\_PREFIX)aomtrace\$(LIB\_POSTFIX)\$(LIB\_EXT)  
\$(INST\_LIBS) \$(OMROOT)\LangCpp\|lib\|\$(LIB\_PREFIX)tomtraceRiC\$(LIB\_POSTFIX)\$(LIB\_EXT)*

## **TraceOxfLibs**

The TraceOxfLibs property is used to specify the framework libraries that are required when Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with

OXF\_LIBS.

```
Default = $(OMROOT)\LangC\lib\$(LIB_PREFIX)oxfinst$(LIB_POSTFIX)$(LIB_EXT)  
$(OMROOT)\LangC\lib\$(LIB_PREFIX)omcomappl$(LIB_POSTFIX)$(LIB_EXT)  
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$(LIB_POSTFIX)$(LIB_EXT)  
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omcomappl$(LIB_POSTFIX)$(LIB_EXT) winmm.lib
```

## TracePreprocessor

The TracePreprocessor property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with INST\_FLAGS.

```
Default = $(DEFINE_QUALIFIER)OMTRACER
```

## UpdateBuildSettingsInIDE

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

```
Default = True
```

## UseLIBCMT

If you want your application to use the statically-linked library libcmt.lib, rather than the dynamically-linked library msvcrt.lib, set the value of the property UseLIBCMT to True.

If the property is set to True, the makefile will include the option /MT. Otherwise, the makefile will include the option /MD.

```
Default = False
```

## UseNewBuildOutputWindow

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the CG::General::ShowLogViewAfterBuild property to True.

```
Default = True
```

## UseNonZeroStdInputHandle

The UseNonZeroStdInputHandle property is a Boolean value that specifies whether to use a non-zero standard input handle.

*Default = True*

## UseRemoteHost

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = False*

## Multi4Win32

The Multi4Win32 metaclass contains theEnvironment settings (Compiler, framework libraries, and so on) for Multi4Win32 compiler.

## AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

This property reflects the change in Version 4.1 where the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created.

This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files. To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

## AnimInstLibs

The AnimInstLibs property is used to specify the static libraries that are required when Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with INST\_LIBS.

*Default = -l\$(LibPrefix)OxfInst\$(BLDTarget)\$OMLibSuffix\$LibExtension  
-l\$(LibPrefix)AomAnim\$(BLDTarget)\$OMLibSuffix\$LibExtension  
-l\$(FrameworkLibPrefix)OmComAppl\$(BLDTarget)\$OMLibSuffix\$LibExtension  
-l\$(LibPrefix)OxfInst\$(BLDTarget)\$OMLibSuffix\$LibExtension*

## AnimPreprocessor

The AnimPreprocessor property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to Animation. In the makefile, these appear in the line that begins with INST\_FLAGS.

*Default = -D\_OMINSTRUMENT*

## BLDAdditionalOptions

The BLDAdditionalOptions property specifies additional compilation switches.

*Default = -I. -threading=multiple --exceptions --no\_implicit\_include --display\_error\_number --diag\_remark 14,161,837,817,815,47,69,830,550 -prelink.args=-r -prelink.args=-X7 -language=cxx*

## BLDIncludeAdditionalBLD

The BLDIncludeAdditionalBLD specifies additional build options.

*Default = Empty MultiLine*

## BLDTarget

The BLDTarget property specifies the target BSP. For example, ":target=Win32". This property also affects the names of the framework libraries used in the link.

## BuildArgumentsInIDE

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the C\_CG::[environment]::BuildCommandInIDE property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandInIDE

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the

application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration. To change this property, use the Configuration window in the browser - do not change it by using the Properties window or by modifying the site.prp file.

Note that this property also affects the names of the framework libraries used in the link. The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

The default value for MultiWin32 is DebugNoExp; for the other environments, the default value is Debug.

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\MultiWin32Make.bat\" MultiWin32Build.bat buildLibs "*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## COM

The COM property specifies whether the current component is a COM component.

By default, this property is set to Checked for all COM components (stereotypes COM DLL, COM EXE, and COM TLB).

If you set this property in the generated makefile for the component, the linker option /SUBSYSTEM is set to :windows.

*Default = Cleared*

## CompileDebug

The CompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

## CompileRelease

The CompileRelease property specifies additional compilation flags for a configuration set to Release mode.

## CompileSwitches

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property. The default values are as follows:

```
Environment Default Compile Switches Borland -I$OMDefaultSpecificationDirectory
-I$(BCROOT)\INCLUDE;,$(OMROOT)\LangCpp";
"$(OMROOT)\LangCpp\oxf";$(OMROOT)\LangCpp\omCom";
-D_RTLDLL;_AFXDLL;WIN32;_CONSOLE;_MBCS;WINDOWS;BORLAND;_BOOLEAN
$(INST_FLAGS) $(INCLUDE_PATH) $(INST_INCLUDES) $OMCPPCompileCommandSet -c Linux
-I -I$OMDefaultSpecificationDirectory -I$(OMROOT) -I$(OMROOT)\LangCpp
-I$(OMROOT)\LangCpp\oxf $(INST_FLAGS) $(INCLUDE_PATH) $(INST_INCLUDES)
-DUSE_Iostream $OMCPPCompileCommandSet -c Microsoft MicrosoftDLL /I . /I
$OMDefaultSpecificationDirectory /I $(OMROOT)\LangCpp /I $(OMROOT)\LangCpp\oxf /nologo /W3
/GX $OMCPPCompileCommandSet /D "_AFXDLL" /D "WIN32" /D "_CONSOLE" /D "_MBCS" /D
"_WINDOWS" $(INST_FLAGS) $(INCLUDE_PATH) $(INST_INCLUDES) /c MicrosoftWinCE /I . /I
$OMDefaultSpecificationDirectory /I $(OMROOT)\LangCpp /I $(OMROOT)\LangCpp\oxf /nologo /W3
/GX- /D _WIN32_WCE=$(CEVersion) /D "$(CEConfigName)" $(MACHINE_C_FLAGS) /D
"_OM_NO_Iostream" /D UNDER_CE=$(CEVersion) /D "UNICODE" /D
"_OM_UNICODE_ONLY" $OMCPPCompileCommandSet /D "_AFXDLL" /D "WIN32"
$(INST_FLAGS) $(INCLUDE_PATH) $(INST_INCLUDES) /c MicrosoftWinCE.NET /I . /I
$(OMROOT)\LangCpp /I $(OMROOT)\LangCpp\oxf /nologo /W3 /GX- /D
_WIN32_WCE=$(CEVersion) /D "$(CEConfigName)" $(MACHINE_C_FLAGS) /D
"_OM_NO_Iostream" /D UNDER_CE=$(CEVersion) /D "UNICODE" /D
"_OM_UNICODE_ONLY" $OMCPPCompileCommandSet /D "_AFXDLL" /D "WIN32"
$(INST_FLAGS) $(INCLUDE_PATH) $(INST_INCLUDES) /D "_X86_" /c MSStandardLibrary /I . /I
$OMDefaultSpecificationDirectory /I $(OMROOT)\LangCpp /I $(OMROOT)\LangCpp\oxf /nologo /W3
/GX $OMCPPCompileCommandSet /D "_AFXDLL" /D "WIN32" /D "_CONSOLE" /D "_MBCS" /D
"_WINDOWS" /D "OM_USE_STL" $(INST_FLAGS) $(INCLUDE_PATH) $(INST_INCLUDES) /c
MultiWin32 ${CPPCompileDebugNoExp} $CPPAdditionalCompileSwitches Nucleus PLUS-PPC -v -c
```

```
-DPLUS -DUSE_IOSTREAM -D__DIAB -t$(CPU) -I. -I$OMDefaultSpecificationDirectory
-I$(OMROOT)\LangCpp -I$(OMROOT)\LangCpp\oxf -I$(ATI_DIR) -Xmismatch-warning
-Xno-common $OMCPPCompileCommandSet $(INST_FLAGS) $(INCLUDE_PATH)
$(INST_INCLUDES) OsePPCDiab OseSfk -I. -I$OMDefaultSpecificationDirectory
-I$(OMROOT)/LangCpp $(INST_FLAGS) $(INCLUDE_PATH) $(INST_INCLUDES)
QNXNeutrinoCW -I. -I$OMDefaultSpecificationDirectory -I$(OMROOT) -I$(OMROOT)/LangCpp
-I$(OMROOT)/LangCpp/oxf $(INST_FLAGS) $(INCLUDE_PATH) $(INST_INCLUDES)
-DUSE_IOSTREAM $OMCPPCompileCommandSet -c QNXNeutrinoGCC Solaris2 Solaris2GNU -I.
-I$OMDefaultSpecificationDirectory -I$(OMROOT) -I$(OMROOT)/LangCpp
-I$(OMROOT)/LangCpp/oxf $(INST_FLAGS) $(INCLUDE_PATH) $(INST_INCLUDES)
-DUSE_IOSTREAM $OMCPPCompileCommandSet -c VxWorks -I$OMDefaultSpecificationDirectory
-I$(OMROOT) -I$(OMROOT)/LangCpp -I$(OMROOT)/LangCpp/oxf -DVxWorks $(INST_FLAGS)
$(INCLUDE_PATH) $OMCPPCompileCommandSet -c
```

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component. The default value is as follows:

```
-D_DEBUG -G
```

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

There is no default value.

## DebugLibSuffix

A suffix added to library names. OMLibSuffix keyword is replaced with this property or ReleaseLibSuffix according to the compilation to the build type: Release/Debug.

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction an IDE such as Eclipse, the EnableDebugIntegrationWithIDE property can be used to specify whether or not the IDE debugger should be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to True, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's exit() function is called automatically. The system function



calls the destructors of framework instances in the order it determines. The EndApplicationCode property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an endApplication() function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to endApplication(), it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The EntryPoint property specifies the name of the main program for a given environment.

The default value is "main."

See also the definition of the EntryPointDeclarationModifier property for more information.

## EnvironmentVarName

The EnvironmentVarName property specifies the name of the global variable that you must define in order to use the compiler. It is used by the MultiMakefileGenerator. The value replaces the "\$value of the EnvironmentVarName" keyword inside the value for the BLDAdditionalOptions property.

*(Default = MULTI\_ROOT)*

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property.

ErrorMessageTokens has three parameters, each with an integer value:

- TotalNumberOfTokens - The number of tokens in the regular expression
- FileTokenPosition - The position of the file name token in the expression
- LineTokenPosition - The position of the line number token in the expression

*(Default = TotalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2)*

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the C\_CG::<Environment>::ExeName property plus the value of the ExeExtension property.

*Default = .exe*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the C\_CG::<Environment>::ExeExtension property.

*(Default = Blank)*

## ExtraFilesToCopy

A list of file names (with full paths) separated with commas. The generator copies this list of files to the folder of the makefile only if the file does not already exist.

## FrameworkLibPrefix

The FrameworkLibPrefix property specifies the prefix of the Rational Rhapsody framework library linked with your application.

*(Default = Multi4Win32)*

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled).

During code generation, these libraries are added to the generated makefile. Note that if you select Release Build Set, these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

The default value is as follows:

```
$(LibPrefix)WebComponents$(BLDTarget)$OMLibSuffix$LibExtension ,  
$(OMRoot)/lib/$(FrameworkLibPrefix)WebServices$(BLDTarget)$OMLibSuffix$LibExtension
```

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment. The default values are as follows:

*(Default = .c)*

## InitTracingCppSupport

When using the tracing feature, C applications are linked to both C and C++ framework libraries. For this environment, if a program uses both C and C++, and the main() function is written in C, the main() function must include a call to \_main() in order to initialize C++ static instances. This property specifies the function call that must be included in main().

*Default = \_main();*

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*(Default = \$executable)*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

The InvokeMake default value is as follows:

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\Multi4Win32Make.bat\" $makefile $maketarget"
```

## InvokeMakeGenerator

The InvokeMakeGenerator specifies the path to the executable program for an external makefile generator. This external generator is started each time you request a makefile generation.

If the specified path is incorrect, Rational Rhapsody generates an error message. If you are using a

full-featured external code generator, this property setting is ignored.

*(Default = \$OMROOT/etc/MultiMakefileGenerator.exe)*

## **IsFileNameShort**

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format.

If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## **LibExtension**

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*(Default = .lib)*

## **LibPrefix**

Combines all the prefixes of library names.

## **LinkDebug**

The LinkDebug property specifies the special link switches used to link in debug mode.

There is no default.

## **LinkRelease**

The LinkRelease property specifies the special link switches used to link in release mode.

There is no default.

## **LinkSwitches**

The LinkSwitches property specifies the standard link switches used to link in any mode. The default values are as follows:

There is no default.

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the C\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*(Default = .gpj)*

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration. For example, the default content of the target type section of a C++ makefile for the Microsoft environment is as follows: ##### Target type (Debug/Release)

```
#####  
C++CompileDebug=$OMC++CompileDebug C++CompileRelease=$OMC++CompileRelease  
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease BuildSet=$OMBldSet  
SUBSYSTEM=$OMSubSystem COM=$OMCOM RPFramewOrkDll=$OMRPFramewOrkDll  
ConfigurationC++CompileSwitches= $OMReusableStatechartSwitches  
$OMConfigurationC++CompileSwitches !IF "$RPFramewOrkDll" == "True"  
ConfigurationC++CompileSwitches= $(ConfigurationC++CompileSwitches) /D "FRAMEWORK_DLL"  
!ENDIF !IF "$(COM)" == "True" SUBSYSTEM=/SUBSYSTEM:windows !ENDIF
```

**Compilation Flags** The compilation flags section of the makefile contains the default compilation flags stored in the CompileSwitches property. For example, the default content of the compilation flags section of a C++ makefile for the Microsoft environment is as follows: ##### Compilation flags

```
#####  
INCLUDE_QUALIFIER=/I LIB_PREFIX=MS
```

**Commands Definitions** The commands definition section of the makefile specifies programs to execute

from the makefile. For example, the default commands definition section of a C++ makefile for the Microsoft environment is as follows: ##### Commands definition #####  
##### RMDIR = rmdir LIB\_CMD=link.exe -lib  
LINK\_CMD=link.exe LIB\_FLAGS=\$OMConfigurationLinkSwitches  
LINK\_FLAGS=\$OMConfigurationLinkSwitches \$(SUBSYSTEM) / MACHINE:I386

**Generated Macros** The generated macros section of the makefile contains a variable that expands to the Rational Rhapsody-generated macros in the makefile. For example: ##### Generated macros #####  
##### \$OMContextMacros  
OBJ\_DIR=\$OMObjectsDir !IF "\$ (OBJ\_DIR)" != "" CREATE\_OBJ\_DIR=if not exist \$(OBJ\_DIR) mkdir  
\$(OBJ\_DIR) CLEAN\_OBJ\_DIR= if exist \$(OBJ\_DIR) \$(RMDIR) \$(OBJ\_DIR) !ELSE  
CREATE\_OBJ\_DIR= CLEAN\_OBJ\_DIR= !ENDIF

The \$OMContextMacros keyword expands several macros in the makefile. Each makefile macro has its own keyword. You can use these keywords separately to customize the makefile. The \$OMContextMacros variable modifies target-specific variables.

Replace the \$OMContextMacros line in the MakeFileContent property with the following:  
FLAGSFILE=\$OMFlagsFile RULESFILE=\$OMRulesFile OMROOT=\$OMROOT  
C\_EXT=\$OMImplExt H\_EXT=\$OMSpecExt OBJ\_EXT=\$OMObjExt EXE\_EXT=\$OMExeExt  
LIB\_EXT=\$OMLibExt INSTRUMENTATION=\$OMInstrumentation TIME\_MODEL=\$OMTimeModel  
TARGET\_TYPE=\$OMTargetType TARGET\_NAME=\$OMTargetName \$OMAllDependencyRule  
TARGET\_MAIN=\$OMTargetMain LIBS=\$OMLibs INCLUDE\_PATH=\$OMIncludePath  
ADDITIONAL\_OBJS=\$OMAdditionalObjs OBJS= \$OMObjs

**Predefined Macros** The predefined macros section of the makefile contains other macros than the Rational Rhapsody-generated macros specified in the generated macros section. For example, the default predefined macros section of a C++ makefile for the Microsoft environment is as follows:

##### Predefined macros #####  
##### \$(OBJS) : \$(INST\_LIBS)  
\$(OXF\_LIBS) LIB\_POSTFIX= !IF "\$ (BuildSet)" == "Release" LIB\_POSTFIX=R !ENDIF !IF  
"\$ (TARGET\_TYPE)" == "Executable" LinkDebug=\$(LinkDebug) /DEBUG  
LinkRelease=\$(LinkRelease) /OPT:NOREF !ELSEIF "\$ (TARGET\_TYPE)" == "Library"  
LinkDebug=\$(LinkDebug) /DEBUGTYPE:CV !ENDIF !IF "\$ (INSTRUMENTATION)" == "Animation"  
INST\_FLAGS=/D "OMANIMATOR" INST\_INCLUDES=/I \$(OMROOT)\LangCpp\aoom /I  
\$(OMROOT)\LangCpp\toom !IF "\$ (RPFrameWorkDll)" == "True" INST\_LIBS=  
OXF\_LIBS=\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)oxanimdll\$(LIB\_POSTFIX) \$(LIB\_EXT)  
!ELSE INST\_LIBS= \$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)aomanip\$(LIB\_POSTFIX) \$(LIB\_EXT)  
OXF\_LIBS=\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)oxfirst\$(LIB\_POSTFIX) \$(LIB\_EXT)  
\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)omComApp\$(LIB\_POSTFIX) \$(LIB\_EXT) !ENDIF  
SOCK\_LIB=wsock32.lib !ELSEIF "\$ (INSTRUMENTATION)" == "Tracing" INST\_FLAGS=/D  
"OMTRACER" INST\_INCLUDES=/I \$(OMROOT)\LangCpp\aoom /I \$(OMROOT)\LangCpp\toom !IF  
"\$ (RPFrameWorkDll)" == "True" INST\_LIBS= OXF\_LIBS=  
\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)oxframedll\$(LIB\_POSTFIX) \$(LIB\_EXT) !ELSE  
INST\_LIBS=\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)tomtrace\$(LIB\_POSTFIX) \$(LIB\_EXT)  
\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)aomanip\$(LIB\_POSTFIX) \$(LIB\_EXT) OXF\_LIBS=  
\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)oxfirst\$(LIB\_POSTFIX) \$(LIB\_EXT)  
\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)omComApp\$(LIB\_POSTFIX) \$(LIB\_EXT) !ENDIF  
SOCK\_LIB=wsock32.lib !ELSEIF "\$ (INSTRUMENTATION)" == "None" INST\_FLAGS=  
INST\_INCLUDES= INST\_LIBS= !IF "\$ (RPFrameWorkDll)" == "True"  
OXF\_LIBS=\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)oxfdll\$(LIB\_POSTFIX) \$(LIB\_EXT) !ELSE  
OXF\_LIBS=\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)oxf\$(LIB\_POSTFIX) \$(LIB\_EXT) !ENDIF  
SOCK\_LIB= !ELSE !ERROR An invalid Instrumentation \$(INSTRUMENTATION) is specified.  
!ENDIF

Generated Dependencies The generated dependencies section of the makefile contains a variable that expands to Rational Rhapsody-generated dependencies and compilation instructions. For example, the generated dependencies section of a C++ makefile for the Microsoft environment is as follows:

```
##### Generated dependencies #####
##### SOMContextDependencies
$OMFileObjPath : $OMMainImplementationFile $(OBJS) $(CPP) $(ConfigurationCPPCompileSwitches)
/Fo"$OMFileObjPath" $OMMainImplementationFile
```

Linking Instructions The linking instructions section of the makefile contains the predefined linking instructions. For example, the default linking instructions section of a C++ makefile for the Microsoft environment is as follows:

```
##### Linking instructions #####
#####
$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS) $OMFileObjPath
$OMMakefileName $OMModelLibs @echo Linking $(TARGET_NAME)$(EXE_EXT) $(LINK_CMD)
$OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \ $(OXF_LIBS) \
$(SOCK_LIB) \ $(LINK_FLAGS) /out:$(TARGET_NAME)$(EXE_EXT)
$(TARGET_NAME)$(LIB_EXT) : $(OBJS) $(ADDITIONAL_OBJS) $OMMakefileName @echo
Building library $@ $(LIB_CMD) $(LIB_FLAGS) /out:$(TARGET_NAME)$(LIB_EXT) $(OBJS)
$(ADDITIONAL_OBJS) clean: @echo Cleanup $OMCleanOBJS if exist $OMFileObjPath erase
$OMFileObjPath if exist *$(OBJ_EXT) erase *$(OBJ_EXT) if exist $(TARGET_NAME).pdb erase
$(TARGET_NAME).pdb if exist $(TARGET_NAME)$(LIB_EXT) erase
$(TARGET_NAME)$(LIB_EXT) if exist $(TARGET_NAME).ilk erase $(TARGET_NAME).ilk if exist
$(TARGET_NAME)$(EXE_EXT) erase $(TARGET_NAME)$(EXE_EXT) $(CLEAN_OBJ_DIR)
```

## MakeFileContentForExe1

This property is the content of the makefiles, in the case of an executable component type.

The default value is as follows:

```
#!gbuild primaryTarget=$PrimaryTarget [Project] -object_dir=$ObjectsDirectory
$MakeFileNameForExe2
```

## MakeFileContentForExe2

This property is the content of the makefiles, in the case of an executable component type.

The default value is as follows:

```
#!gbuild [Program] -o $OMTargetName$ExeExtension -object_dir=$ObjectsDirectory
$BLDAdditionalOptions $BLDIncludeAdditionalBLD -I$(OMRoot)/LangC -L$(OMRoot)/LangC/lib
$OMUserIncludePath $LinkSwitches $OMCompilationFlag $CompileSwitches
$OMInstrumentationFlags $OMInstrumentationLibs $BLDAdditionalDefines $OMUserLibs
$OMMainFiles$ImpExtension $OMSrcFiles
```

## MakeFileContentForLib1

This property is the content of the makefiles, in the case of a library component type.

The default value is as follows:

```
#!/gbuild primaryTarget=$PrimaryTarget [Project] -object_dir=$ObjectsDirectory  
$MakeFileNameForLib2
```

## MakeFileContentForLib2

This property is the content of the makefiles, in case of library component type.

The default value is as follows:

```
#!/gbuild [Library] -o $OMTargetName$LibExtension -object_dir=$ObjectsDirectory  
$BLDAdditionalOptions $BLDIncludeAdditionalBLD -I$(OMRoot)/LangC $OMUserIncludePath  
$OMCompilationFlag $CompileSwitches $OMInstrumentationFlags $BLDAdditionalDefines  
$OMSrcFiles
```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the C\_CG::::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## MakeFileNameForExe1

This property contains the name of the makefiles, in the case of an executable component type.

The default value is as follows:

```
$(OMTargetName)$MakeExtension
```

## MakeFileNameForExe2

This property contains the name of the makefiles, in the case of an executable component type.

The default value is as follows:

```
$(OMTargetName)_program$MakeExtension
```

## MakeFileNameForLib1



This property contains the name of the makefiles, in the case of a library component type.

The default value is as follows:

`$(OMTargetName)$MakeExtension`

## **MakeFileNameForLib2**

The name of the makefiles, in case of library component type.

The default value is as follows:

`$(OMTargetName)_library$MakeExtension`

## **NetAndSocketLibs**

A list of library names that is added to OMWebLibs keyword if web-enabling flag is on or to OMInstrumentationFlags keyword if the instrumentation is in animation mode.

## **NoneInstLibs**

The NoneInstLibs property is used to specify the static libraries that are required when Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with INST\_LIBS.

*Default = -l\$(LibPrefix)Ox\$(BLDTarget)\$OMLibSuffix\$LibExtension*

## **NonePreprocessor**

The NonePreprocessor property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to None. In the makefile, these appear in the line that begins with INST\_FLAGS.

*Default = Blank*

## **NullValue**

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## **ObjCleanCommand**

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build. The default values are as follows:

Environment Clean Command Borland if exist \$OMFileObjPath erase \$OMFileObjPath INTEGRITY IntegrityESTL Microsoft MicrosoftDLL MicrosoftWinCE.NET MSStandardLibrary GNAT Empty string MultiWin32 JDK if exist \$OMFileObjPath del \$OMFileObjPath Linux \$(RM) \$OMFileObjPath OsePPCDiab OseSfk QNXNeutrinoCW QNXNeutrinoGCC Solaris2 Solaris2GNU VxWorks NucleusPLUS-PPC @if exist \$OMFileObjPath \$(RM) \$OMFileObjPath

## ObjectName

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = empty string*

## ObjectsDirectory

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*(Default = work)*

## ObjExtension

The ObjExtension property specifies the extension appended to compiled object files for a given environment. The default values are as follows:

*(Default = .obj)*

## OSFileSystemCaseSensitive

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive. The default values are as follows:

Environment Default Value Borland Cleared GNAT INTEGRITY IntegrityESTL JDK Microsoft MicrosoftDLL MicrosoftWinCE.NET MSStandardLibrary MultiWin32 NucleusPLUS-PPC OseSfk OsePPCDiab RAVEN\_PPC SPARK VxWorks Linux Checked QNXNeutrinoCW QNXNeutrinoGCC Solaris2 Solaris2GNU

## ParseErrorDescript

The ParseErrorDescript property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = ([^"]+)", line ([0-9]+)[:] (error/warning) (.\*)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

The default values are as follows:

Environment Default Value Borland PsosX86  
ToTalNumberOfTokens=5,FileTokenPosition=4,LineTokenPosition=5 GNAT  
ToTalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2 JDK Linux QNXNeutrinoCW  
QNXNeutrinoGCC Solaris2GNU SPARK VxWorks IntegrityESTL  
ToTalNumberOfTokens=3,FileTokenPosition=1,LineTokenPosition=2 Microsoft MicrosoftDLL  
MicrosoftWinCE.NET MSSStandardLibrary NucleusPLUS-PPC OsePPCDiab OseSfk Solaris2

## ParseMakeError

The ParseMakeError property is used to define a regular expression that represents the format of make process or linker error messages. This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = (Error)[:] (build failed)*

## ParseSeverityError

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^"]+)", line ([0-9]+)[:] (error)*

## ParseSeverityWarning

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^"]+)", line ([0-9]+)[:] (warning)*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation.

*(Default = /)*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

The default value is Checked.

## RCCompileCommand

The RCCompileCommand property is a string that specifies the compilation command for the resource file.

In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = empty string*

## RCExtension

The RCExtension property is a string that specifies the extension for resource files.

In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*(Default = .rc)*

## ReleaseLibSuffix

A suffix added to library names. OMLibSuffix keyword is replaced with this property or DebugLibSuffix according to the compilation to the build type: Release/Debug.

*(Default = R)*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely.

In such configurations, the host is the machine running the product, whereas the target is the machine running the application.

To run remotely, the UseRemoteHost property must be set to Checked. If UseRemoteHost is Checked and RemoteHost is blank, the current host name is used for the remote host. You can use this as a workaround if you have problems running animated applications on Windows 95. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = empty string*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*(Default = .h)*

## SubSystem

The SubSystem property is a string that defines the type of the program for the Microsoft linker.

The possible values are as follows:

- **CONSOLE** - Used for a Win32 character-mode application
- **WINDOWS** - Used for an application that does not require a console
- **NATIVE** - Applies device drivers for Windows NT
- **POSIX** - Creates an application that runs with the POSIX subsystem in Windows NT

*(Default = /SUBSYSTEM:console)*

## TraceInstLibs

The TraceInstLibs property is used to specify the static libraries that are required when Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with INST\_LIBS.

*Default = -l\$(LibPrefix)OxfInst\$(BLDTarget)\$OMLibSuffix\$LibExtension  
-l\$(LibPrefix)AomTrace\$(BLDTarget)\$OMLibSuffix\$LibExtension  
-l\$(FrameworkLibPrefix)OmComAppl\$(BLDTarget)\$OMLibSuffix\$LibExtension  
-l\$(LibPrefix)OxfInst\$(BLDTarget)\$OMLibSuffix\$LibExtension  
-l\$OMRoot/LangCpp/lib/\$(FrameworkLibPrefix)TomTraceRiC\$(BLDTarget)\$OMLibSuffix\$LibExtension  
-l\$OMRoot/LangCpp/lib/\$(FrameworkLibPrefix)Oxf\$(BLDTarget)\$OMLibSuffix\$LibExtension  
-l\$OMRoot/LangCpp/lib/\$(FrameworkLibPrefix)OmComAppl\$(BLDTarget)\$OMLibSuffix\$LibExtension  
-lwssock32.lib*

## TracePreprocessor

The TracePreprocessor property is used to specify conditions that should be used for conditional compilation for projects where Instrumentation Mode is set to Tracing. In the makefile, these appear in the line that begins with INST\_FLAGS.

*Default = -DOMTRACER -DRIC\_APP*

## UpdateBuildSettingsInIDE

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build

settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## UseActorsCode

The UseActorsCode property specifies whether code is generated for actors. The value of the property should be synchronized with the configuration Generate Code For Actors check mark (located in the configuration Initialization tab).

*Default = Cleared*

## UseNewBuildOutputWindow

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the CG::General::ShowLogViewAfterBuild property to True.

*Default = Checked*

## UseRemoteHost

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

The default value for the following environments is Cleared:

Borland GNAT INTEGRITY IntegrityESTL Microsoft MicrosoftDLL MSStandardLibrary MultiWin32

The default value for the following environments is Checked:

Linux MicrosoftWinCE.NET NucleusPLUS-PPC OsePPCDiab OseSfk QNXNeutrinoCW  
QNXNeutrinoGCC Solaris2 Solaris2GNU VxWorks

## NucleusPLUS-PPC

The NucleusPLUS-PPC metaclass contains the Environment settings (Compiler, framework libraries, and so on) for NucleusPLUS-PPC compiler.

## AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*(Default = \$(OMROOT)/LangC/osconfig/Nucleus)*

## BuildArgumentsInIDE

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the C\_CG::[environment]::BuildCommandInIDE property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandInIDE

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release

version of the configuration. To change this property, use the Configuration window in the browser - do not change it by using the Properties window or by modifying the site.prp file.

Note that this property also affects the names of the framework libraries used in the link. The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*(Default = Debug)*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\numake.bat" nubuild.mak buildLibs \*  
*"CPU=\$CPU\" \"BUILD\_SET=\$BuildCommandSet\" \" \"*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## CompileSwitches

The CompileSwitches property specifies the compiler switches.

The default value is as follows:

*-v -c -DUSE\_STDIO -DPLUS -t\$(CPU) -I. -I\$OMDefaultSpecificationDirectory -I\$(OMROOT)/LangC  
-I\$(OMROOT)/LangC/oxf -I\$(ATI\_DIR) -I\$(ATI\_DIR)/plus -Xoptimized-debug-on -XO -Xsize-opt  
-Xmismatch-warning -Xno-common \$OMCPPCompileCommandSet \$(INST\_FLAGS)  
\$(INCLUDE\_PATH) \$(INST\_INCLUDES)*



## CPPAdditionalReservedWords

The CPPAdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use.

In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody. The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

The default value is as follows:

```
__asm __finally naked __based __inline __single_inheritance __cdecl __int8 __stdcall __declspec __int16  
dllexport __int32 __try dllimport __int64 __virtual_inheritance __except __leave __fastcall  
__multiple_inheritance
```

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The default values are as follows:

```
@echo Compiling $OMFileImpPath $(CREATE_OBJ_DIR) @$$(CC) $OMFileCPPCompileSwitches -o  
$OMFileObjPath $OMFileImpPath
```

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

The default value is -g -D\_DEBUG -DASSERT\_DEBUG.

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

The default value is -DNDEBUG.

## CPU

The CPU property represents the target CPU for projects using the NucleusPLUS-PPC environment. The value of this property is referred to in the value of the buildFrameworkCommand property, which defines the command that is run to build the framework for the environment when you select Code > Build Framework from the main Rational Rhapsody menu.

*Default = PPC860ES*

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: `$OMFileObjPath : $OMFileImpPath "*.bmp" "*.ico" "*.rc2"`

The default value is `$OMFileObjPath : $OMFileImpPath $OMFileSpecPath $(CREATE_OBJ_DIR) $OMFileDependencies`.

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction an IDE such as Eclipse, the EnableDebugIntegrationWithIDE property can be used to specify whether or not the IDE debugger should be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to True, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's exit() function is called automatically. The system function calls the destructors of framework instances in the order it determines. The EndApplicationCode property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an endApplication() function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to endApplication(), it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The EntryPoint property specifies the name of the main program for a given environment.

The default value is "numain."

See also the definition of the EntryPointDeclarationModifier property for more information.

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens has three parameters, each with an integer value:

- TotalNumberOfTokens - The number of tokens in the regular expression
- FileTokenPosition - The position of the file name token in the expression
- LineTokenPosition - The position of the line number token in the expression

The default value is "TotalNumberOfTokens=3,FileTokenPosition=1,LineTokenPosition=2."

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the C\_CG::<Environment>::ExeName property plus the value of the ExeExtension property.

*Default = .elf*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the C\_CG::<Environment>::ExeExtension property.

*(Default = Blank)*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements. The file inclusions are generated in the makefile.

The C default value is \$OMSpecIncludeInElements \$OMImpIncludeInElements.

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled).

During code generation, these libraries are added to the generated makefile. Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries). The default values are as follows:

The default value is as follows:

```
$(OMROOT)\LangC\lib\$(LIB_PREFIX)WebComponents$(LIB_POSTFIX)$(LIB_EXT),  
$(OMROOT)\lib\$(LIB_PREFIX)WebServices$(LIB_POSTFIX)$(LIB_EXT)
```

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment. The default values are as follows:

( *C Default* = .c )

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

(*Default* = .INCLUDE:)

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

(*Default* = "\$executable")

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$.

(*Default* = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\numake.bat\" \"\$makefile \$maketarget")

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format.

If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*(Default = .lib)*

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

*Default = empty string*

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

*(Default = \$OMLinkCommandSet)*

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the C\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

(Default = .mak)

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration. For example, the default content of the target type section of a C++ makefile for the Microsoft environment is as follows: ##### Target type (Debug/Release)

```
#####  
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease  
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease BuildSet=$OMBUILDSet  
SUBSYSTEM=$OMSubSystem COM=$OMCOM RPFramEWorkDll=$OMRPFramEWorkDll  
ConfigurationCPPCompileSwitches= $OMReusableStatechartSwitches  
$OMConfigurationCPPCompileSwitches !IF "$ (RPFramEWorkDll)" == "True"  
ConfigurationCPPCompileSwitches= $(ConfigurationCPPCompileSwitches) /D "FRAMEWORK_DLL"  
!ENDIF !IF "$ (COM)" == "True" SUBSYSTEM=/SUBSYSTEM:windows !ENDIF
```

**Compilation Flags** The compilation flags section of the makefile contains the default compilation flags stored in the CompileSwitches property. For example, the default content of the compilation flags section of a C++ makefile for the Microsoft environment is as follows: ##### Compilation flags

```
#####  
INCLUDE_QUALIFIER=/I LIB_PREFIX=MS
```

**Commands Definitions** The commands definition section of the makefile specifies programs to execute from the makefile. For example, the default commands definition section of a C++ makefile for the Microsoft environment is as follows: ##### Commands definition #####

```
##### RMDIR = rmdir LIB_CMD=link.exe -lib  
LINK_CMD=link.exe LIB_FLAGS=$OMConfigurationLinkSwitches  
LINK_FLAGS=$OMConfigurationLinkSwitches $(SUBSYSTEM) / MACHINE:I386
```

**Generated Macros** The generated macros section of the makefile contains a variable that expands to the Rational Rhapsody-generated macros in the makefile. For example: ##### Generated macros

```
##### $OMContextMacros  
OBJ_DIR=$OMObjectsDir !IF "$ (OBJ_DIR)" != "" CREATE_OBJ_DIR=if not exist $(OBJ_DIR) mkdir  
$(OBJ_DIR) CLEAN_OBJ_DIR= if exist $(OBJ_DIR) $(RMDIR) $(OBJ_DIR) !ELSE  
CREATE_OBJ_DIR= CLEAN_OBJ_DIR= !ENDIF
```

The \$OMContextMacros keyword expands several macros in the makefile. Each makefile macro has its own keyword. You can use these keywords separately to customize the makefile. The \$OMContextMacros variable modifies target-specific variables. Replace the \$OMContextMacros line in the MakeFileContent property with the following: `FLAGSFILE=$OMFlagsFile`  
`RULESFILE=$OMRulesFile OMROOT=$OMROOT C_EXT=$OMImplExt H_EXT=$OMSpecExt`  
`OBJ_EXT=$OMObjExt EXE_EXT=$OMExeExt LIB_EXT=$OMLibExt`  
`INSTRUMENTATION=$OMInstrumentation TIME_MODEL=$OMTimeModel`  
`TARGET_TYPE=$OMTargetType TARGET_NAME=$OMTargetName $OMAllDependencyRule`  
`TARGET_MAIN=$OMTargetMain LIBS=$OMLibs INCLUDE_PATH=$OMIncludePath`  
`ADDITIONAL_OBJS=$OMAdditionalObjs OBJS= $OMObjs`

**Predefined Macros** The predefined macros section of the makefile contains other macros than the Rational Rhapsody-generated macros specified in the generated macros section. For example, the default predefined macros section of a C++ makefile for the Microsoft environment is as follows:

```
##### Predefined macros #####
##### $(OBJS) : $(INST_LIBS)
$(OXF_LIBS) LIB_POSTFIX= !IF "$(BuildSet)"=="Release" LIB_POSTFIX=R !ENDIF !IF
"$(TARGET_TYPE)"=="Executable" LinkDebug=$(LinkDebug) /DEBUG
LinkRelease=$(LinkRelease) /OPT:NOREF !ELSEIF "$(TARGET_TYPE)"=="Library"
LinkDebug=$(LinkDebug) /DEBUGTYPE:CV !ENDIF !IF "$(INSTRUMENTATION)"=="Animation"
INST_FLAGS=/D "OMANIMATOR" INST_INCLUDES=/I $(OMROOT)\LangCpp\om /I
$(OMROOT)\LangCpp\tom !IF "$(RPFrameWorkDll)"=="True" INST_LIBS=
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfanimdll$(LIB_POSTFIX) $(LIB_EXT)
!ELSE INST_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomanim$(LIB_POSTFIX) $(LIB_EXT)
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfinst$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)"=="Tracing" INST_FLAGS=/D
"OMTRACER" INST_INCLUDES=/I $(OMROOT)\LangCpp\om /I $(OMROOT)\LangCpp\tom !IF
"$(RPFrameWorkDll)"=="True" INST_LIBS= OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxftracedll$(LIB_POSTFIX) $(LIB_EXT) !ELSE
INST_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)tomtrace$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomtrace$(LIB_POSTFIX) $(LIB_EXT) OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfinst$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)"=="None" INST_FLAGS=
INST_INCLUDES= INST_LIBS= !IF "$(RPFrameWorkDll)"=="True"
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfdll$(LIB_POSTFIX) $(LIB_EXT) !ELSE
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB= !ELSE !ERROR An invalid Instrumentation $(INSTRUMENTATION) is specified.
!ENDIF
```

**Generated Dependencies** The generated dependencies section of the makefile contains a variable that expands to Rational Rhapsody-generated dependencies and compilation instructions. For example, the generated dependencies section of a C++ makefile for the Microsoft environment is as follows:

```
##### Generated dependencies #####
##### $OMContextDependencies
$OMFileObjPath : $OMMainImplementationFile $(OBJS) $(CPP) $(ConfigurationCPPCompileSwitches)
/Fo"$OMFileObjPath" $OMMainImplementationFile
```

**Linking Instructions** The linking instructions section of the makefile contains the predefined linking instructions. For example, the default linking instructions section of a C++ makefile for the Microsoft environment is as follows: ##### Linking instructions #####

```
#####
$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS) $OMFileObjPath
```

```

$OMMakefileName $OMModelLibs @echo Linking $(TARGET_NAME)$(EXE_EXT) $(LINK_CMD)
$OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \ $(OXF_LIBS) \
$(SOCK_LIB) \ $(LINK_FLAGS) /out:$(TARGET_NAME)$(EXE_EXT)
$(TARGET_NAME)$(LIB_EXT) : $(OBJS) $(ADDITIONAL_OBJS) $OMMakefileName @echo
Building library $@ $(LIB_CMD) $(LIB_FLAGS) /out:$(TARGET_NAME)$(LIB_EXT) $(OBJS)
$(ADDITIONAL_OBJS) clean: @echo Cleanup $OMCleanOBJS if exist $OMFileObjPath erase
$OMFileObjPath if exist *$(OBJ_EXT) erase *$(OBJ_EXT) if exist $(TARGET_NAME).pdb erase
$(TARGET_NAME).pdb if exist $(TARGET_NAME)$(LIB_EXT) erase
$(TARGET_NAME)$(LIB_EXT) if exist $(TARGET_NAME).ilk erase $(TARGET_NAME).ilk if exist
$(TARGET_NAME)$(EXE_EXT) erase $(TARGET_NAME)$(EXE_EXT) $(CLEAN_OBJ_DIR)

```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the C.CG::<Environment>::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## ObjCleanCommand

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*(Default = @if exist \$OMFileObjPath \$(RM) \$OMFileObjPath)*

## ObjectName

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = empty string*

## ObjectsDirectory



The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

## ObjExtension

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*(Default = .o)*

## OSFileSystemCaseSensitive

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Cleared*

## ParseErrorDescript

The ParseErrorDescript property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = ([^()]+)([()([0-9]+)]\*)[:] (error|warning|fatal error) (.\*)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

The default is *([^()]+)([()([0-9]+)]\*)[:] (error|warning|fatal error).*

## ParseSeverityError

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^()]+)([()([0-9]+)]\*)[:] (error|fatal error)*

## ParseSeverityWarning

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^()]+)(/[0-9]+)[/:] (warning)*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Cleared*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running the product, whereas the target is the machine running the application. To run remotely, the UseRemoteHost property must be set to True. If UseRemoteHost is True and RemoteHost is blank, the current host name is used for the remote host. You can use this as a workaround if you have problems running animated applications on Windows 95.

The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = empty string*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*(Default = .h)*

## UpdateBuildSettingsInIDE

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## UseNewBuildOutputWindow

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log

tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the `CG::General::ShowLogViewAfterBuild` property to True.

*Default = Checked*

## UseRemoteHost

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Checked*

## Operation

The Operation metaclass contains properties that control operations.

### ActiveOperationExtractMeCall

Configures the function call that extracts the “me” pointer from the operating system (OS) context variable.

*Default*

`=<isConditionalProperty><Rte_Pim_$(Model::AR_BMT::PerInstanceMemoryName)>($<C_CG::Operation::OSContextName>)`

### ActivityReferenceToAttributes

The ActivityReferenceToAttributes property specifies whether Rational Rhapsody should generate references in the functor object, thereby giving you direct access to the attributes of the class that owns the modeled operation (without the need for `this_`). See the section on activity diagrams in the Rational Rhapsody help for information about modeled operations and functor classes.

*Default = Checked*

### AnimAllowInvocation

The AnimAllowInvocation property specifies whether primitive and triggered operations can be called during instrumentation. If an operation is called during animation, its return value is displayed in the output window; if it is traced, the return value is displayed in the console. The possible values are as follows:

- All - Enable all operation calls, regardless of visibility.
- None - Do not enable operation calls.
- Public - Enable calls to public operations only.
- Protected - Enable calls to protected operations only.

*(Default = None)*

## **AnimateTriggeredOperationReturnValue**

The AnimateTriggeredOperationReturnValue property allows you to specify that the return values of triggered operations should be displayed in animated sequence diagrams.

*Default = Checked*

## **DeclarationModifier**

The DeclarationModifier property is used to allow Rational Rhapsody to reverse engineer non-standard keywords that appear in operation declarations. Keywords that appear between the return type and the operation name are stored as the value of this property, and the property is then used during code generation to re-create the original code.

Since this is a code generation property, it can also be used to add non-standard keywords to code even when reverse engineering is not used.

This property is used in conjunction with the PreDeclarationModifier and PostDeclarationModifier properties.

*Default = Blank*

## **DescriptionInImplementation**

By default, descriptions for operations are generated only in the specification files. Using the properties GenerateDescriptionInImplementation and DescriptionInImplementation, you can have a descriptive comment generated in the implementation file as well.

When the value of the property GenerateDescriptionInImplementation is set to "UseDescriptionInImplementationProperty", the text entered for the property DescriptionInImplementation is generated as a comment before the operation definition.

If the value of the property GenerateDescriptionInImplementation is set to "UseDescriptionInImplementationProperty" and the value of the property DescriptionInImplementation is left blank, no comment is generated before the operation definition.

Note that the text that was specified for the DescriptionInImplementation property is always generated as a comment before the operation definition, whether the definition appears in the specification file or in the implementation file. Similarly, the text that was specified on the Description tab is always generated as a comment before the operation declaration, whether the declaration appears in the specification file or in the implementation file.

During roundtripping, any changes that were made to the comment that precedes the operation definition are ignored.

*Default = Blank*

## DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- `$Name` - the element name
- `$FullName` - the full path of the element (P1::P2::C.a)
- `$Description` - the text provided on the Description tab of the Features window for the element
- `$Requirements` - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the `$Requirements` keyword in the DescriptionTemplate property to get such comments generated for specific model elements.
- `$Login` - the login name of the user who generated the file
- `$CodeGeneratedDate` - the date on which the code was generated
- `$CodeGeneratedTime` - the time at which the code was generated
- `$RhapsodyVersion` - the version of Rhapsody that was used to generate the file
- For operations: `$Signature` - the operation signature, `$Arguments` - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: `$Type` - the type of the attribute
- For arguments: `$Type` - the argument type, `$Direction` - the argument direction (In, Out, or InOut)
- For events: `$Arguments` - the text provided on the Description tab of the Features window for the event arguments
- For association ends: `$Target` - the other end of the association
- For requirements: `$ID` - the ID specified for the requirement, `$Specification` - the specification text for the requirement
- The values of tags can be included by using the syntax `$tag_name`. For example, if you use a tag called `author` for your classes, you can use the keyword `$author` in the value of DescriptionTemplate in order to have the value of the `author` tag included in the element description.

- The values of properties can be included by using the syntax `$property_name`. For example, you can use the keyword `$Animate` in the value of `DescriptionTemplate` in order to have the value of the `Animate` property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as `$Name`)
- Tag keywords
- Property keywords

If the value of a keyword is a `MultiLine` value, then each new line (except the first one) starts with the value of the `<lang>_CG::Configuration::DescriptionBeginLine` property, and each line ends with the value of the `<lang>_CG::Configuration::DescriptionEndLine` property.

*Default = Blank*

## DisableAutoGeneratedInitializer

When Rational Rhapsody generates code, certain code is automatically generated in the initializer of the constructor. The `DisableAutoGeneratedInitializer` property allows you to disable the generation of this auto-generated initializer code for class elements in the constructor.

Note that when this initializer-generation feature is enabled, any changes you make to the initializer code is not brought into your model during roundtripping.

If you want to have changes made to the initializer code brought into the model during roundtripping, set the value of this property to `Checked`.

If you reverse engineer a class, the value of this property is changed to `Checked` for the class.

*Default = Cleared (default is Checked in code-centric settings)*

## EntryCondition

The `EntryCondition` property specifies the task guard.

*Default = empty string*

## ExecutableOperationBeginCode

The `ExecutableOperationBeginCode` property specifies the code used for the beginning of the executable operation. It is used when the executable operation is inlined.

*Default = Empty MultiLine*

## ExecutableOperationEndCode

The ExecutableOperationEndCode property specifies the code used for the end of the executable operation. It is used when the executable operation is inlined.

*Default = Empty MultiLine*

## GenerateDescriptionForFlowchartActions

Beginning in release 8.0.6, when code is generated for flowcharts, comments are generated to represent the descriptions entered for actions and transitions. To preserve the previous code generation behavior for pre-8.0.6 models, the [lang]\_CG::Operation::GenerateDescriptionForFlowchartActions and [lang]\_CG::Operation::GenerateDescriptionForFlowchartFlows properties were added to the backward compatibility settings for C and C++ with a value of False. If you want to use the new code generation behavior with pre-8.0.6 models, change the value of these properties to True and reload the model.

*Default = True*

## GenerateDescriptionForFlowchartFlows

Beginning in release 8.0.6, when code is generated for flowcharts, comments are generated to represent the descriptions entered for actions and transitions. To preserve the previous code generation behavior for pre-8.0.6 models, the [lang]\_CG::Operation::GenerateDescriptionForFlowchartActions and [lang]\_CG::Operation::GenerateDescriptionForFlowchartFlows properties were added to the backward compatibility settings for C and C++ with a value of False. If you want to use the new code generation behavior with pre-8.0.6 models, change the value of these properties to True and reload the model.

*Default = True*

## GenerateDescriptionInImplementation

By default, descriptions for operations are generated only in the specification files. Using the properties GenerateDescriptionInImplementation and DescriptionInImplementation, you can have a descriptive comment generated in the implementation file as well.

The possible values for GenerateDescriptionInImplementation are:

- UseSpecificationText - the text from the Description tab will be generated before the operation definition as well as before the operation declaration
- UseDescriptionInImplementationProperty - the text specified for the property DescriptionInImplementation will be generated as a comment before the operation definition

Note that the text that was specified for the DescriptionInImplementation property is always generated as a comment before the operation definition, whether the definition appears in the specification file or in the implementation file. Similarly, the text that was specified on the Description tab is always generated as a comment before the operation declaration, whether the declaration appears in the specification file or in the implementation file.

*Default = UseDescriptionInImplementationProperty*

## **GenerateImplementation**

The GenerateImplementation property specifies whether to generate the body for the operation.

To generate Import pragmas, add the "pragma..." declaration in the C\_CG::Operation::SpecificationEpilog property.

*Default = Checked*

## **GenerateReturnStatementInVoidOperations**

When code is generated for flowcharts, all operations include a return statement, even operations declared as void. If you do not want to have a return statement generated for void operations, set the value of GenerateReturnStatementInVoidOperations to False.

*Default = True*

## **ImplementActivity Diagram**

The ImplementActivity Diagram property enables or disables code generation for activity diagrams.

*Default = Cleared*

## **ImplementationEpilog**

The ImplementationEpilog property enables you to add any code that you want to be added as verbatim text (to be ignored by Rational Rhapsody) to the end of the definition of a model element.. For example, you could wrap a section of code with an #ifdef-#endif pair, add compiler-specific keywords, or add a #pragma statement.

For example, to specify that an operation is available only when the code is compiled with \_DEBUG, set the following properties for the operation:

- Set SpecificationProlog to #ifdef \_DEBUG cr.
- Set SpecificationEpilog to #endif.
- Set ImplementationProlog to #ifdef \_DEBUG cr.
- Set ImplementationEpilog to #endif.

The following table lists whether leading and trailing linefeeds are generated.

Metaclass Leading Linefeed Added? Class Yes Package No

*Default = Empty MultiLine*



## ImplementFlowchart

ImplementFlowchart property that specifies whether or not code should be generated for the flow charts created by the user. It can be set at the individual operation level or at higher levels, such as class or package.

*Default = Checked*

## ImplementationName

The ImplementationName property gives an operation one model name and generate it with another name. It is introduced as a workaround that generates const and non-const operations with the same name. For example:

- Create a class A.
- Add a non-const operation f().
- Add a const operation f\_const().
- Set the C\_CG::Operation::ImplementationName property for f\_const() to “f.”
- Generate the code.

The resulting code is as follows: `class A { ... void f(); /* the non const f */ ... void f() const; /* actually f_const() */ ... };` The creation of two operations with the same signature, differing only in whether it is a const, is a common practice in C++, especially for STL users.

*Default = empty string*

## ImplementationProlog

The ImplementationProlog property enables you to add any code that you want to be added as verbatim text (to be ignored by the software) to the beginning of the definition of a model element. For example, you could wrap a section of code with an `#ifdef-#endif` pair, add compiler-specific keywords, or add a `#pragma` statement.

For example, to specify that an operation is available only when the code is compiled with `_DEBUG`, set the following properties for the operation:

- Set SpecificationProlog to `#ifdef _DEBUG cr.`
- Set SpecificationEpilog to `#endif.`
- Set ImplementationProlog to `#ifdef _DEBUG cr.`
- Set ImplementationEpilog to `#endif.`

The following table lists whether leading and trailing linefeeds are generated.

Metaclass Trailing Linefeed Added? Class No Package Yes

*Default = Empty MultiLine*

## Inline

The Inline property specifies how inline operations are generated. Which operations are affected by the Inline property depends on the metaclass:

- Attribute - Applies only to operations that handle attributes (such as accessors and mutators)
- Operation - Applies to all operations
- Relation - Applies only to operations that handle relations

Inlining in Rational Rhapsody Developer for C You can inline attribute and relation accessors and mutators for increased code performance. The visibility of the inlined operations can be either public or private. The accessor can contain only a return statement, or more than just a return statement. For Rational Rhapsody Developer for C, there are two possible settings for this property:

- none - The operation is not generated inline. For example: `/* Mutator of Tank::ItsDishwasher relation */ void Tank_setItsDishwasher(struct Tank_t* const me, struct Dishwasher_t* p_Dishwasher) { if(p_Dishwasher != NULL) Dishwasher__setItsTank(p_Dishwasher, me); Tank__setItsDishwasher(me, p_Dishwasher); } /* Accessor to Tank::ItsDishwasher relation */ struct Dishwasher_t* Tank_getItsDishwasher( const struct Tank_t* const me) { return (struct Dishwasher_t*)me-itsDishwasher; }`
- in\_header - The operation is generated inline, as follows:
- Mutators are defined as macro definitions. For example: `/* Inline Mutator of Tank::ItsDishwasher relation */ #define Tank_setItsDishwasher(me, p_Dishwasher) \ { \ if((p_Dishwasher) != NULL) \ Dishwasher__setItsTank((p_Dishwasher), (me)); \ Tank__setItsDishwasher((me), (p_Dishwasher)); \ }`
- Accessors that contain only return statements are defined as macros (the return statement and the semicolon at the end of expression are omitted); other accessors are generated as operations. For example: `/* Inline Accessor to Tank::ItsDishwasher relation */ #define Tank_getItsDishwasher(me) ((me)-itsDishwasher)`
- If the attribute visibility is defined as Private, the macro definitions are placed in the implementation (.c) file.
- If the attribute visibility is defined as Public, the macro definitions are placed in the specification (.h) file.

Note the following:

- Each instance of the parameters for the macro s parenthesized.
- You cannot inline an accessor if it contains statements other than the return statement. For example, accessors to relations implemented by using RiCCollection cannot be generated as function-like macros.
- You cannot set the Inline property separately for specific helpers (for example, only mutators) - this property affects all helpers of the attribute or relation.
- If a multi-lined mutator macro is called as the body of the “then” part of an “if ...else” statement, you must enclose it in parentheses or it generates a compilation error. For example: `// Erroneous code: If (itsDishwasher != NULL) Tank_setItsDishwasher(me, itsDishwasher); Else Return; // Correct code: If (itsDishwasher != NULL) { Tank_setItsDishwasher(me, itsDishwasher); } Else Return;`

## IsEntry

The IsEntry property indicates whether the operation is a task entry or a regular operation in AdaTask and AdaTaskType classes.

*Default = Cleared*

## IsExplicit

The IsExplicit property is a Boolean value that allows you to specify that a constructor is an explicit constructor.

*Default = Cleared*

## IsNative

The IsNative property specifies whether the Java modifier “native” should be added to an operation in the source file. The body of such operations, if specified, is ignored by the code generator.

*Default = Cleared*

## Kind

The Kind property specifies the kind of operation that should be generated for an element. The kind of operations that can be generated is language-dependent (for example, virtual and abstract exist only in C++ and Java). In Java, Kind can be defined only for attributes and operations, but not for relations.

This property affects class operations, in addition to accessors and mutators for relations and attributes.

The possible values are as follows:

- common - Class operations and accessor/mutator are non-virtual.
- virtual - Class operations and accessor/mutator are virtual. This type is valid for C++ and Java only.
- abstract - Class operations and accessor/mutator are pure virtual. This type is valid for C++ and Java only.

*Default = common*

## LocalVariablesDeclaration

The LocalVariablesDeclaration property specifies variables that you want to appear in the declaration of the entrypoint or operation.

*Default = empty string*

## MarkPrologEpilogInAnnotations

The MarkPrologEpilogInAnnotations property specifies whether to generate ignore annotations for the

Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties so they are ignored during roundtrip. When you insert code element declarations (variables, types, functions, and so on) in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, after a full roundtrip those elements are added to the model and are duplicated on the next code generation.

When you use the `MarkPrologEpilogInAnnotations` property, you can have Rational Rhapsody automatically ignore the information specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties instead of adding the ignore annotations manually. The possible values for the `MarkPrologEpilogInAnnotations` property are:

- **None** - Rational Rhapsody does not generate any annotations. Any models created before Version 4.1 automatically have this property setting.
- **Ignore** - Rational Rhapsody generates the `///  
[ ignore` annotation before the code specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, and generates the `///  
]` annotation after the code specified in those properties.
- **Auto** - If the code in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties is one line (it does not contain any newline characters (`\n`)), no annotations are generated (the same behavior has the **None** setting). If there is more than one line, Rational Rhapsody generates the `///  
[ ignore` annotation before the code specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, and generates the `///  
]` annotation after the code specified in those properties (the same behavior as the **Ignore** setting).

During roundtrip, any ignore annotations in the comments of the element are not included in its description. Because the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties are generated between the annotation and its declaration for the element, you cannot rename those elements on roundtrip. If you change the name of an element, it is removed from the model and added with the new name. Some model information (for example, property settings) might be lost.

*Default = Auto*

## Me

*The Me property specifies the name of the first argument to operations generated in C. (Default = me)*

## MeDeclType

The `MeDeclType` property is a string that specifies the type of the first argument to operations generated in C, as a pointer to an object or object type. The default value is as follows: `$objectName* const` The variable `$objectName` is replaced with the name of the object or object type.

## MultiLineArgumentList

The `MultiLineArgumentList` property specifies how the list of arguments for an operation should be formatted.

If set to `False`, all of the arguments appear on the same line as the operation name.

If set to True, each argument appears on a separate line.

*Default = False*

## OpeningBraceStyle

The OpeningBraceStyle property controls where the opening brace of the code block is positioned - on the same line as the element name (SameLine) or on the following line (NewLine).

*Default = SameLine*

## OSContextName

Configures the name of the operating system (OS) context parameter that may be passed to operation.

*Default = Self*

## OSContextDeclType

Configures the type name of the operating system (OS) context parameter that can be added as a parameter to operation.

*Default = Rte\_Instance*

## PostDeclarationModifier

The PostDeclarationModifier property is used to allow Rational Rhapsody to reverse engineer non-standard keywords that appear in operation declarations. Keywords that appear after the operation argument list are stored as the value of this property, and the property is then used during code generation to re-create the original code.

Since this is a code generation property, it can also be used to add non-standard keywords to code even when reverse engineering is not used.

This property is used in conjunction with the PreDeclarationModifier and DeclarationModifier properties.

*Default = Blank*

## PreDeclarationModifier

The PreDeclarationModifier property is used to allow Rational Rhapsody to reverse engineer non-standard keywords that appear in operation declarations. Keywords that appear before the return type are stored as the value of this property, and the property is then used during code generation to re-create the original code.

Since this is a code generation property, it can also be used to add non-standard keywords to code even

when reverse engineering is not used.

This property is used in conjunction with the DeclarationModifier and PostDeclarationModifier properties.

*Default = Blank*

## PrivateQualifier

*The PrivateQualifier property specifies the qualifier that is printed at the beginning of a private operation declaration or definition. You can set this property to an empty string to prevent the generation of the static qualifier in the private function declaration or definition. (Default = static)*

## ProtectedName

The ProtectedName property specifies the pattern used to generate names of private operations in C. The default value is as follows: \$opName

The \$opName variable specifies the name of the operation. For example, the generated name of a private operation go() of an object A is generated as: go()

## PublicName

The PublicName property specifies the pattern used to generate names of public operations.

*Default = \$ObjectName\_\$opName*

The \$ObjectName variable specifies the name of the object; the \$opName variable specifies the name of the operation. For example, the generated name of a public operation go() of an object A is generated as: A\_go()

## PublicQualifier

The PublicQualifier property specifies the qualifier that is printed at the beginning of a public operation declaration or definition. Note that the Static check mark in the operation window UI is disabled in Rational Rhapsody Developer for C because the check mark is associated with class-wide semantics that are not supported by Rational Rhapsody Developer for C.

When loading models from previous versions, the Static check mark is cleared; if the operation is public, the C\_CG::Operation::PublicQualifier property value is set to Static in order to generate the same code.

*Default = empty string*

## RefactorRenameRegularExpression

When you use the Refactor:Rename feature, Rhapsody searches the user-defined code in the model for all occurrences of the name of the specified element, including places where the element name appears as

part of the name of elements whose name is derived from the element that is being renamed.

To locate these elements, Rhapsody uses a set of regular expressions.

The regular expressions used for this search are defined by the property `RefactorRenameRegularExpression`.

Different values are used for this property, depending on what type of element you are renaming. For example, you will see that this property has a different value under `Class` than it does under `Attribute`. For some types of elements, Rhapsody uses a more generic expression, taken from the value of the property under `ModelElement`.

In general, the value of this property is a regular expression that consists of a number of "Or" components. For the renaming to work properly, each such component must contain three groupings, with the middle grouping using the variable `$keyword`, which represents the name of the element that is being renamed. In some cases, the expression takes this information from the value of another property rather than using the `$keyword` variable.

```
Default = ([^A-Za-z0-9_]|^)($keyword)([^A-Za-z0-9_]|$)
|(^$<C.CG::Operation::PublicName>[^A-Za-z0-9_]|$)
|([A-Za-z0-9_]$<C.CG::Operation::PublicName>[^A-Za-z0-9_]|$)
|(^$<C.CG::Operation::ProtectedName>[^A-Za-z0-9_]|$)
|([A-Za-z0-9_]$<C.CG::Operation::ProtectedName>[^A-Za-z0-9_]|$)
```

## Renames

The `Renames` property enables one element to rename another element of the same type. You can also rename an element by using a `renames` dependency. In the case of a conflict, the dependency has precedence. Note the following:

- For attributes, this property works only for static attributes in a class or for attributes in a package.
- For operations, this property contains the name of the operation this operation is renaming. The signatures of the two operations must match.

*Default = empty string*

## RenamesKind

*The `RenamesKind` property specifies whether the renaming of the operation designated in the `C.CG::Operation::Renames` property is “as specification” or “as body.” (Default = *Specification*)*

## ReturnTypeByAccess

The `ReturnTypeByAccess` property determines whether the return type is generated as an access type or a regular type. Note that this property is applicable only to classes for which an access type is generated.

*Default = Cleared*

## RunnableOperationImplementationTemplate

The RunnableOperationImplementationTemplate property specifies the structure of the implementation of a runnable operation. It can include both user code and auto-generated code.

*Default = \$UserImplementation \$GeneratedImplementation*

## Simplify

If you are using the Rational Rhapsody customizable code generation mechanism, the Simplify property can be used to change the way specific types of elements are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - The element is ignored.
- Copy - The element is copied from the original to the simplified model. It is not modified in any way.
- Default - Uses the standard simplification for this item, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for the element has been applied.

*Default = "Default"*

## SimplifyTriggeredOperation

If you are using the Rational Rhapsody customizable code generation mechanism, the SimplifyTriggeredOperation property can be used to change the way triggered operations are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - Triggered operations are ignored.
- Copy - Triggered operations are copied from the original to the simplified model. They are not modified in any way.
- Default - Uses the standard simplification for triggered operations, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for triggered operations has been applied.

*Default = "Default"*

## SpecificationEpilog



The SpecificationEpilog property allows you to add code to the end of the declaration of a model element.

For example, you can use this property to add an #ifdef to indicate that an operation is available only when the code is compiled with \_DEBUG, by setting the following properties:

- For SpecificationProlog, enter the value #ifdef \_DEBUG and a new line.
- For SpecificationEpilog, enter the value #endif
- For ImplementationProlog, enter the value #ifdef \_DEBUG and a new line.
- For ImplementationEpilog, enter the value #endif

*Default = Blank*

## SpecificationProlog

The SpecificationProlog property allows you to add code to the beginning of the declaration of a model element.

For example, you can use this property to add an #ifdef to indicate that an operation is available only when the code is compiled with \_DEBUG, by setting the following properties:

- For SpecificationProlog, enter the value #ifdef \_DEBUG and a new line.
- For SpecificationEpilog, enter the value #endif
- For ImplementationProlog, enter the value #ifdef \_DEBUG and a new line.
- For ImplementationEpilog, enter the value #endif

*Default = Blank*

## TaskDefaultScheme

The TaskDefaultScheme property sets the task default entry scheme. The possible values are as follows:

- Conditional
- Timed
- None

*(Default = None)*

## TaskDefaultSchemeDelayStatement

The TaskDefaultScheme property sets the task default entry statement for timed entry schemes.

*Default = Empty MultiLine*

## ThisByAccess

*The ThisByAccess property specifies whether to pass the this parameter as an access mode parameter for*

*a non-static operation. Default = Cleared*

## **ThisName**

*The ThisName property specifies the name of the this parameter, which specifies the instance. (Default = this)*

## **ThrowExceptions**

The ThrowExceptions property specifies the exceptions that an operation can throw. Separate multiple exceptions with commas.

*Default = empty string*

## **UsePre82Flowchart**

Prior to release 8.2, in cases where an "else" guard contained blank spaces, code was not generated correctly for flowcharts. This issue was corrected in 8.2. To preserve the previous code generation behavior for pre-8.2 models, the property [lang]\_CG::Operation::UsePre82Flowchart was added to the backward compatibility settings for C and C++ with a value of True.

## **UsePre82GuardedOperationDescriptionLocation**

Prior to release 8.2., when C code was generated, the descriptions for guarded operations were generated in the implementation file rather than in the specification file. This issue was corrected in 8.2. To preserve the previous code generation behavior for pre-8.2 models, the property C\_CG::Operation::UsePre82GuardedOperationDescriptionLocation was added to the backward compatibility settings for C with a value of True.

## **UsePre821DescriptionInImplementation**

Prior to release 8.2.1, if you used the property DescriptionInImplementation to define a description to be generated in the implementation of an operation, and you also modified the value of the property TemplateDescription (for the specification description), the comment for the implementation description was not generated correctly. This issue was corrected in 8.2.1. To preserve the previous code generation behavior for pre-8.2.1 models, the property [lang]\_CG::Operation::UsePre821DescriptionInImplementation was added to the backward compatibility settings for C and CPP with a value of True.

## **UsePre821GuardedOperationDescriptionLocation**

Prior to release 8.2., when C code was generated, the descriptions for guarded operations were generated in the implementation file rather than in the specification file. This issue was corrected in 8.2, however,

the parts of the description that are taken from arguments and requirements were not included in the fix for the problem. In release 8.2.1, this remaining issue was corrected. To preserve the previous code generation behavior for pre-8.2.1 models, the property `C_CG::Operation::UsePre821GuardedOperationDescriptionLocation` was added to the backward compatibility settings for C with a value of `True`.

## UseProtectedNameAndPublicNameInFile

The values of the properties `C_CG::Operation::ProtectedName` and `C_CG::Operation::PublicName` are ordinarily used to determine the names of operations belonging to a class in the model. However, the values of these properties do not affect the names used for functions in Files.

The property `UseProtectedNameAndPublicNameInFile` can be used to specify that the values of these properties should also be used to determine the names of ordinary stand-alone functions. This mechanism can be used in cases where you would like to add specific prefixes or suffixes to the names of functions, for example, adding the file name as a prefix to the names of the functions in a file.

Note that for certain auto-generated operations such as `Init()` and `Cleanup()`, the values of the properties `ProtectedName` and `PublicName` always affect the names used, regardless of the value of `UseProtectedNameAndPublicNameInFile`.

*Default = False*

## VirtualMethodGenerationScheme

The `VirtualMethodGenerationScheme` property enables compatibility with earlier versions for methods of interface and abstract classes. The possible values are as follows:

- `Default` - The class type is class-wide, but the `this` parameters are not.
- `ClassWideOperations` - The class type is not class-wide, but the `this` parameters are.

*Default = Default*

## OptimizedTopDownStatechart

The `OptimizedTopDownStatechart` metaclass contains properties that control the optimization of top-down code generation for statecharts. To enable the use of the properties in this metaclass, set `C_CG::Configuration::StatechartImplementation` to `"OptimizedTopDown"`.

## AddCodeDocumentation

The `AddCodeDocumentation` property enables `OptimizedTopDownStatechart` code to include comments for statechart implementation functions, state transitions, and static reactions in a state.

*Default = Cleared*

Note: To enable the use of the properties in the `OptimizedTopDownStatechart` metaclass, set `C_CG::Configuration::StatechartImplementation` to "OptimizedTopDown".

## AllowCodeOptimization

The `AllowCodeOptimization` property enables the `OptimizedTopDownStatechart` code to be optimized. The various optimizations are each controlled by the following properties:

- `C_CG::OptimizedTopDownStatechart::ClutchEntranceToStateHierarchy`
- `C_CG::OptimizedTopDownStatechart::EmptyOverlappingTestsForStateHierarchy`
- `C_CG::OptimizedTopDownStatechart::InlineEnteringExitingReactions`
- `C_CG::OptimizedTopDownStatechart::InlineMaxTranstionsCriteria`
- `C_CG::OptimizedTopDownStatechart::InlineRootStateDefaultTransitionTest`
- `C_CG::OptimizedTopDownStatechart::MergeStateSequenceWithNoGuard`

*Default = Checked*

Note: To enable the use of the properties in the `OptimizedTopDownStatechart` metaclass, set `C_CG::Configuration::StatechartImplementation` to "OptimizedTopDown".

## ClutchEntranceToStateHierarchy

The `ClutchEntranceToStateHierarchy` property enables the `OptimizedTopDownStatechart` code to be optimized to enter the innermost state in a state hierarchy, wherever possible. (Type of optimization gained when this property is set to `Checked`: ROM, run-time optimization.)

*Default = Checked*

See also:

- `C_CG::OptimizedTopDownStatechart::EmptyOverlappingTestsForStateHierarchy`
- `C_CG::OptimizedTopDownStatechart::InlineEnteringExitingReactions`
- `C_CG::OptimizedTopDownStatechart::InlineMaxTranstionsCriteria`
- `C_CG::OptimizedTopDownStatechart::InlineRootStateDefaultTransitionTest`
- `C_CG::OptimizedTopDownStatechart::MergeStateSequenceWithNoGuard`

Note: To enable the use of the properties in the `OptimizedTopDownStatechart` metaclass, set `C_CG::Configuration::StatechartImplementation` to "OptimizedTopDown".

## EmptyOverlappingTestsForStateHierarchy

The `EmptyOverlappingTestsForStateHierarchy` property enables the `OptimizedTopDownStatechart` code to be optimized to skip overlapping comparisons when there is a state hierarchy, so control cannot be in the inner state if it has not entered the surrounding state. (Type of optimization gained when this property is set to `Checked`: ROM optimization, but this might reduce run-time efficiency.)

*Default = Cleared*

See also:

- C.CG::OptimizedTopDownStatechart::ClutchEntranceToStateHierarchy
- C.CG::OptimizedTopDownStatechart::InlineEnteringExitingReactions
- C.CG::OptimizedTopDownStatechart::InlineMaxTranstionsCriteria
- C.CG::OptimizedTopDownStatechart::InlineRootStateDefaultTransitionTest
- C.CG::OptimizedTopDownStatechart::MergeStateSequenceWithNoGuard

Note: To enable the use of the properties in the OptimizedTopDownStatechart metaclass, set C.CG::Configuration::StatechartImplementation to "OptimizedTopDown".

## InlineEnteringExitingReactions

The InlineEnteringExitingReactions property enables the OptimizedTopDownStatechart code to be optimized to try to inline the entering and exiting reactions of states, in order to avoid generating the entering and exiting reaction functions. (Type of optimization gained when this property is set to Checked: RAM, ROM optimization.)

*Default = Checked*

See also:

- C.CG::OptimizedTopDownStatechart::ClutchEntranceToStateHierarchy
- C.CG::OptimizedTopDownStatechart::EmptyOverlappingTestsForStateHierarchy
- C.CG::OptimizedTopDownStatechart::InlineMaxTranstionsCriteria
- C.CG::OptimizedTopDownStatechart::InlineRootStateDefaultTransitionTest
- C.CG::OptimizedTopDownStatechart::MergeStateSequenceWithNoGuard

Note: To enable the use of the properties in the OptimizedTopDownStatechart metaclass, set C.CG::Configuration::StatechartImplementation to "OptimizedTopDown".

## InlineMaxTranstionsCriteria

The InlineMaxTranstionsCriteria property enables the OptimizedTopDownStatechart code to be optimized as follows: At the end of the statechart code, there is a section that tests whether a transition was made in the current step. The test of whether another step is needed uses a state variable that stores the information about the state that is being entered. When this option is selected, the code flagging the need for another step will be put inline in the transition code, eliminating the need for the state variable.

When selected, the user can enter the maximum number of transitions he is willing to tolerate. If the number of actual transitions will be greater than this number, the optimization will not be performed. (Type of optimization gained when this property is set to used: RAM, ROM optimization.)

*Default = 999*

See also:

- C\_CG::OptimizedTopDownStatechart::ClutchEntranceToStateHierarchy
- C\_CG::OptimizedTopDownStatechart::EmptyOverlappingTestsForStateHierarchy
- C\_CG::OptimizedTopDownStatechart::InlineEnteringExitingReactions
- C\_CG::OptimizedTopDownStatechart::InlineRootStateDefaultTransitionTest
- C\_CG::OptimizedTopDownStatechart::MergeStateSequenceWithNoGuard

Note: To enable the use of the properties in the OptimizedTopDownStatechart metaclass, set C\_CG::Configuration::StatechartImplementation to "OptimizedTopDown".

## InlineRootStateDefaultTransitionTest

The InlineRootStateDefaultTransitionTest property enables the OptimizedTopDownStatechart code to be optimized to inline the test on default transitions into testing of other transitions. (Type of optimization gained when this property is set to Checked: RAM, ROM optimization.)

*Default = Checked*

See also:

- C\_CG::OptimizedTopDownStatechart::ClutchEntranceToStateHierarchy
- C\_CG::OptimizedTopDownStatechart::EmptyOverlappingTestsForStateHierarchy
- C\_CG::OptimizedTopDownStatechart::InlineEnteringExitingReactions
- C\_CG::OptimizedTopDownStatechart::InlineMaxTranstionsCriteria
- C\_CG::OptimizedTopDownStatechart::MergeStateSequenceWithNoGuard

Note: To enable the use of the properties in the OptimizedTopDownStatechart metaclass, set C\_CG::Configuration::StatechartImplementation to "OptimizedTopDown".

## MergeStateSequenceWithNoGuard

The MergeStateSequenceWithNoGuard property enables the OptimizedTopDownStatechart code to be optimized to merge state sequences with no guards on transition into a single state wherever possible. (Type of optimization gained when this property is set to Checked: RAM, ROM, run-time optimization.)

*Default = Checked*

See also:

- C\_CG::OptimizedTopDownStatechart::ClutchEntranceToStateHierarchy
- C\_CG::OptimizedTopDownStatechart::EmptyOverlappingTestsForStateHierarchy
- C\_CG::OptimizedTopDownStatechart::InlineEnteringExitingReactions
- C\_CG::OptimizedTopDownStatechart::InlineMaxTranstionsCriteria
- C\_CG::OptimizedTopDownStatechart::InlineRootStateDefaultTransitionTest

Note: To enable the use of the properties in the OptimizedTopDownStatechart metaclass, set C\_CG::Configuration::StatechartImplementation to "OptimizedTopDown".

## OSEK21NT

A list of file names (with full paths) separated with commas. The generator copies this list of files to the folder of the makefile only if the file does not already exist.

### ExtraFilesToCopy

A list of file names (with full paths) separated with commas. The generator copies this list of files to the folder of the makefile only if the file does not already exist.

## OSEK21HC12

A list of file names (with full paths) separated with commas. The generator copies this list of files to the folder of the makefile only if the file does not already exist.

### ExtraFilesToCopy

A list of file names (with full paths) separated with commas. The generator copies this list of files to the folder of the makefile only if the file does not already exist.

## Package

The Package metaclass contains properties that affect packages.

### Animate

The Animate property specifies whether animation code is generated for an element. You can specify your own animation function by using the `CG::Attribute::AnimSerializeOperation` property. The semantics of the Animate property is always in favor of the owner settings:

- If a package Animate property is set to False, all the classes owned by the package are not animated, regardless of the class Animate settings.
- If a class Animate property is set to False, all the elements in the class (attributes, operations, relations, and so on) are not animated.
- If an operation Animate property is set to False, all the arguments are not animated.
- If the AnimateArguments property is set to False, all the arguments are not animated, regardless of the specific argument Animate property settings.

*Default = Checked*

## ContributesToNamespace

The ContributesToNamespace property specifies whether the packages contained in this package is declared as children packages of this package. Regardless of the setting, a directory is created for the current package to hold its contained elements.

*Default = Checked*

## DefineNameSpace

The DefineNameSpace property specifies whether a package defines a namespace. A namespace is a declarative region that attaches an additional identifier to any names declared inside it.

*Default = Cleared*

## DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- `$Name` - the element name
- `$FullName` - the full path of the element (P1::P2::C.a)
- `$Description` - the text provided on the Description tab of the Features window for the element
- `$Requirements` - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the `$Requirements` keyword in the DescriptionTemplate property to get such comments generated for specific model elements.
- `$Login` - the login name of the user who generated the file
- `$CodeGeneratedDate` - the date on which the code was generated
- `$CodeGeneratedTime` - the time at which the code was generated
- `$RhapsodyVersion` - the version of Rhapsody that was used to generate the file



- For operations: \$Signature - the operation signature, \$Arguments - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: \$Type - the type of the attribute
- For arguments: \$Type - the argument type, \$Direction - the argument direction (In, Out, or InOut)
- For events: \$Arguments - the text provided on the Description tab of the Features window for the event arguments
- For association ends: \$Target - the other end of the association
- For requirements: \$ID - the ID specified for the requirement, \$Specification - the specification text for the requirement
- The values of tags can be included by using the syntax \$tag\_name. For example, if you use a tag called author for your classes, you can use the keyword \$author in the value of DescriptionTemplate in order to have the value of the author tag included in the element description.
- The values of properties can be included by using the syntax \$property\_name. For example, you can use the keyword \$Animate in the value of DescriptionTemplate in order to have the value of the Animate property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as \$Name)
- Tag keywords
- Property keywords

If the value of a keyword is a MultiLine value, then each new line (except the first one) starts with the value of the <lang>\_CG::Configuration::DescriptionBeginLine property, and each line ends with the value of the <lang>\_CG::Configuration::DescriptionEndLine property.

*Default = Blank*

## EventsBaseID

All events are assigned an ID number that is used when code is generated.

If you would like the numbering of events in a package to start at a number different than the default start number used by Rational Rhapsody, you can use the EventsBaseID property to specify your own start number.

*Default = 1*

## GenerateDirectory

The GenerateDirectory property is used to specify that the code files for classes and files in a package should be generated in a separate directory that has the same name as the package.

If the property is set to False, the code files will be generated in a single directory that contains the generated files for all packages for which this property is set to False.

Note that if this property is set to False and your model contains classes with the same name in different packages, the generated files for these classes will overwrite the previous file with the same name. This will leave you with only one generated file even though the model contains a number of classes with that name.

*Default = False*

## ImplIncludes

The ImplIncludes property specifies the names (including full paths) of header files to be included at the top of implementation files generated for classes, objects or object types, or packages. Separate multiple file names by using commas, without spaces.

*Default = empty string*

## ImplementationEpilog

The ImplementationEpilog property enables you to add any code that you want to be added as verbatim text (to be ignored by Rational Rhapsody) to the end of the definition of a model element.. For example, you could wrap a section of code with an #ifdef-#endif pair, add compiler-specific keywords, or add a #pragma statement.

For example, to specify that an operation is available only when the code is compiled with \_DEBUG, set the following properties for the operation:

- Set SpecificationProlog to #ifdef \_DEBUG cr.
- Set SpecificationEpilog to #endif.
- Set ImplementationProlog to #ifdef \_DEBUG cr.
- Set ImplementationEpilog to #endif.

The following table lists whether leading and trailing linefeeds are generated.

Metaclass	Leading Linefeed Added?	Class	Yes	Package	No
-----------	-------------------------	-------	-----	---------	----

*Default = Empty MultiLine*

## ImplementationPragmas

The ImplementationPragmas property specifies the user-defined pragmas to generate in the body.

*Default = Empty MultiLine*

## ImplementationPragmasInContextClause

The ImplementationPragmasInContextClause property specifies the user-defined pragmas to generate in the context clause of the body.

*Default = Empty MultiLine*

## ImplementationProlog

The ImplementationProlog property enables you to add any code that you want to be added as verbatim text (to be ignored by the software) to the beginning of the definition of a model element. For example, you could wrap a section of code with an #ifdef-#endif pair, add compiler-specific keywords, or add a #pragma statement.

For example, to specify that an operation is available only when the code is compiled with \_DEBUG, set the following properties for the operation:

- Set SpecificationProlog to #ifdef \_DEBUG cr.
- Set SpecificationEpilog to #endif.
- Set ImplementationProlog to #ifdef \_DEBUG cr.
- Set ImplementationEpilog to #endif.

The following table lists whether leading and trailing linefeeds are generated.

Metaclass	Trailing Linefeed Added?	Class	No	Package	Yes
-----------	--------------------------	-------	----	---------	-----

*Default = Empty MultiLine*

## InitializationCode

The InitializationCode property specifies the user-defined initialization code to add to the package body.

*Default = Empty MultiLine*

## IsNested

The IsNested property specifies whether to generate the class or package as nested.

*Default = Cleared*

## IsPrivate

The IsPrivate property specifies whether to generate the class or package as private.

*Default = Cleared*

## NestingVisibility

*The NestingVisibility property specifies the visibility of the generated specification of the nested class or package. (Default = Public)*

## MarkPrologEpilogInAnnotations

The MarkPrologEpilogInAnnotations property specifies whether to generate ignore annotations for the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties so they are ignored during roundtrip. When you insert code element declarations (variables, types, functions, and so on) in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, after a full roundtrip those elements are added to the model and are duplicated on the next code generation.

When you use the MarkPrologEpilogInAnnotations property, you can have Rational Rhapsody automatically ignore the information specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties instead of adding the ignore annotations manually. The possible values for the MarkPrologEpilogInAnnotations property are:

- None - Rational Rhapsody does not generate any annotations. Any models created before Version 4.1 automatically have this property setting.
- Ignore - Rational Rhapsody generates the `///  
ignore` annotation before the code specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, and generates the `///  
]` annotation after the code specified in those properties.
- Auto - If the code in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties is one line (it does not contain any newline characters (`\n`)), no annotations are generated (the same behavior has the None setting). If there is more than one line, Rational Rhapsody generates the `///  
ignore` annotation before the code specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, and generates the `///  
]` annotation after the code specified in those properties (the same behavior as the Ignore setting).

During roundtrip, any ignore annotations in the comments of the element are not included in its description. Because the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties are generated between the annotation and its declaration for the element, you cannot rename those elements on roundtrip. If you change the name of an element, it is removed from the model and added with the new name. Some model information (for example, property settings) might be lost.

*Default = Auto*

## PackageClassNamePolicy

The PackageClassNamePolicy property specifies the naming policy for classes generated by Rational Rhapsody. The possible values are as follows:

- Default - Use the default naming style (the package class name is the same as the package name).
- WithSuffix - Add a suffix to the class name. The suffix is “\_pkgClass”.

*Default = Default*

## PackageEventIdRange

*The PackageEventIdRange property specifies the maximum number of events allowed in a package. This*

*property is set on the component level. (Default = 200)*

## Renames

The Renames property enables one element to rename another element of the same type. You can also rename an element by using a renames dependency. In the case of a conflict, the dependency has precedence. Note the following:

- For attributes, this property works only for static attributes in a class or for attributes in a package.
- For operations, this property contains the name of the operation this operation in renaming. The signatures of the two operations must match.

*Default = empty string*

## Simplify

If you are using the Rational Rhapsody customizable code generation mechanism, the Simplify property can be used to change the way specific types of elements are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - The element is ignored.
- Default - Uses the standard simplification for this item, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for the element has been applied.

*Default = "Default"*

## SpecificationEpilog

The SpecificationEpilog property allows you to add code to the end of the declaration of a model element.

For example, you can use this property to add an `#ifdef` to indicate that an operation is available only when the code is compiled with `_DEBUG`, by setting the following properties:

- For SpecificationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For SpecificationEpilog, enter the value `#endif`
- For ImplementationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For ImplementationEpilog, enter the value `#endif`

*Default = Blank*

## SpecificationPragmas

The SpecificationPragmas property specifies the user-defined pragmas to generate in the specification.

*Default = Empty MultiLine*

## SpecificationPragmasInContextClause

The SpecificationPragmasInContextClause property specifies the user-defined pragmas to generate in the context clause of the specification.

*Default = Empty MultiLine*

## SpecificationProlog

The SpecificationProlog property allows you to add code to the beginning of the declaration of a model element.

For example, you can use this property to add an `#ifdef` to indicate that an operation is available only when the code is compiled with `_DEBUG`, by setting the following properties:

- For SpecificationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For SpecificationEpilog, enter the value `#endif`
- For ImplementationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For ImplementationEpilog, enter the value `#endif`

*Default = Blank*

## SpecIncludes

The SpecIncludes property specifies the names (including full paths) of header files to be included at the top of specification files generated for classes (C++ and Java), objects or object types (C), and packages. Separate multiple file names by using commas, without spaces.

*Default = empty string*

## Port

The Port metaclass controls whether code is generated for ports.

## DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- `$Name` - the element name
- `$FullName` - the full path of the element (P1::P2::C.a)
- `$Description` - the text provided on the Description tab of the Features window for the element
- `$Requirements` - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the `$Requirements` keyword in the `DescriptionTemplate` property to get such comments generated for specific model elements.
- `$Login` - the login name of the user who generated the file
- `$CodeGeneratedDate` - the date on which the code was generated
- `$CodeGeneratedTime` - the time at which the code was generated
- `$RhapsodyVersion` - the version of Rhapsody that was used to generate the file
- For operations: `$Signature` - the operation signature, `$Arguments` - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: `$Type` - the type of the attribute
- For arguments: `$Type` - the argument type, `$Direction` - the argument direction (In, Out, or InOut)
- For events: `$Arguments` - the text provided on the Description tab of the Features window for the event arguments
- For association ends: `$Target` - the other end of the association
- For requirements: `$ID` - the ID specified for the requirement, `$Specification` - the specification text for the requirement
- The values of tags can be included by using the syntax `$tag_name`. For example, if you use a tag called `author` for your classes, you can use the keyword `$author` in the value of `DescriptionTemplate` in order to have the value of the `author` tag included in the element description.
- The values of properties can be included by using the syntax `$property_name`. For example, you can use the keyword `$Animate` in the value of `DescriptionTemplate` in order to have the value of the `Animate` property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as `$Name`)
- Tag keywords
- Property keywords

If the value of a keyword is a MultiLine value, then each new line (except the first one) starts with the value of the <lang>\_CG::Configuration::DescriptionBeginLine property, and each line ends with the value of the <lang>\_CG::Configuration::DescriptionEndLine property.

*Default = Blank*

## Generate

The Generate property specifies whether to generate code for a particular type of element.

*Default = Checked*

## OptimizeCode

Code generation for ports and flow ports was optimized in version 7.5.3 of Rhapsody, relative to the code generated in previous versions. A new property named OptimizeCode was added with a default value of True. If the value of this property is set to False, the old code generation mechanism will be used for ports.

## Simplify

If you are using the Rational Rhapsody customizable code generation mechanism, the Simplify property can be used to change the way specific types of elements are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - The element is ignored.
- Copy - The element is copied from the original to the simplified model. It is not modified in any way.
- Default - Uses the standard simplification for this item, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for the element has been applied.

*Default = "Default"*

## SupportMulticast

Use this property to control the multicasting ability of a port. Use the property to enable data or events to be sent from one sender port to many.

The possible values are:

- Always - Rational Rhapsody always generates multicasting ability to each port in the model.
- Smart - Rational Rhapsody identifies ports that are connected to more than 1 port and generates code to



support multicasting to those ports only.

- **Never** - Rational Rhapsody never generates code supporting multicasting of data/event through ports. This is the value for models created before Rational Rhapsody 7.5, so the old behavior is the same.

*Default = Smart*

## QNXNeutrinoMomentics

The QNXNeutrinoMomentics metaclass contains the Environment settings (compiler, framework libraries, and so on) for the QNXNeutrinoMomentics environment.

### EndApplicationCode

When the main function ends, the system's `exit()` function is called automatically. The system function calls the destructors of framework instances in the order it determines. The `EndApplicationCode` property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an `endApplication()` function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to `endApplication()`, it should come after any additional code that you have specified.

*Default = Blank*

### EntryPoint

The `EntryPoint` property specifies the name of the main program for a given environment.

*Default = main*

### QuoteOMROOT

*Default = True*

### UseShortPathNameForRoot

If your project uses the QNXNeutrinoMomentics environment, and the path to the Rhapsody "Share" directory contains one or more spaces, then the compiler will have trouble locating files from this directory that are referenced in the makefile. The `UseShortPathNameForRoot` property is used to resolve this problem. When the value of the property is set to `True`, a DOS-style path is used for the "Share" directory in the makefile, rather than the full path.

*Default = True*

# Relation

The Relation metaclass contains properties that affect relations.

## Add

The Add property specifies the command used to add an item to a container.

*(Default = Add\_\$target:c)*

## AddGenerate

The AddGenerate property specifies whether to generate an Add() operation for relations.

*Default = Checked*

## Animate

The Animate property specifies whether animation code is generated for an element. You can specify your own animation function by using the CG::Attribute::AnimSerializeOperation property. The semantics of the Animate property is always in favor of the owner settings:

- If a package Animate property is set to False, all the classes owned by the package are not animated, regardless of the class Animate settings.
- If a class Animate property is set to False, all the elements in the class (attributes, operations, relations, and so on) are not animated.
- If an operation Animate property is set to False, all the arguments are not animated.
- If the AnimateArguments property is set to False, all the arguments are not animated, regardless of the specific argument Animate property settings.

*Default = Checked*

## Clear

The Clear property specifies the name of an operation that removes all items from a relation.

*(Default = Clear\_\$target:c)*

## ClearGenerate

The ClearGenerate property specifies whether to generate a Clear() operation for relations.

*Default = Checked*

## CreateComponent

*The CreateComponent property specifies the name of an operation that creates a new component in a composite class. (Default = New\_\$target:c)*

## CreateComponentGenerate

The CreateComponentGenerate property specifies whether to generate a CreateComponent operation for composite objects. Setting this property to False is one way to optimize your code for size.

*Default = Checked*

## DataMemberVisibility

The DataMemberVisibility property specifies the visibility of the relation data member. For example, if the relation is implemented as a pointer, this property determines whether the pointer data member is declared as public, private, or protected. The default value for Ada and C is Private; the default value for C++ and Java is Protected.

## DeleteComponent

*The DeleteComponent property specifies the name of an operation that deletes a component from a composite class. (Default = Delete\_\$target:c)*

## DeleteComponentGenerate

The DeleteComponentGenerate property specifies whether to generate a DeleteComponent() operation for composite objects.

*Default = Checked*

## DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text

generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- `$Name` - the element name
- `$FullName` - the full path of the element (`P1::P2::C.a`)
- `$Description` - the text provided on the Description tab of the Features window for the element
- `$Requirements` - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the `$Requirements` keyword in the `DescriptionTemplate` property to get such comments generated for specific model elements.
- `$Login` - the login name of the user who generated the file
- `$CodeGeneratedDate` - the date on which the code was generated
- `$CodeGeneratedTime` - the time at which the code was generated
- `$RhapsodyVersion` - the version of Rhapsody that was used to generate the file
- For operations: `$Signature` - the operation signature, `$Arguments` - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: `$Type` - the type of the attribute
- For arguments: `$Type` - the argument type, `$Direction` - the argument direction (In, Out, or InOut)
- For events: `$Arguments` - the text provided on the Description tab of the Features window for the event arguments
- For association ends: `$Target` - the other end of the association
- For requirements: `$ID` - the ID specified for the requirement, `$Specification` - the specification text for the requirement
- The values of tags can be included by using the syntax `$tag_name`. For example, if you use a tag called `author` for your classes, you can use the keyword `$author` in the value of `DescriptionTemplate` in order to have the value of the `author` tag included in the element description.
- The values of properties can be included by using the syntax `$property_name`. For example, you can use the keyword `$Animate` in the value of `DescriptionTemplate` in order to have the value of the `Animate` property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as `$Name`)
- Tag keywords
- Property keywords

If the value of a keyword is a `MultiLine` value, then each new line (except the first one) starts with the value of the `<lang>_CG::Configuration::DescriptionBeginLine` property, and each line ends with the value of the `<lang>_CG::Configuration::DescriptionEndLine` property.

*Default = Blank*

## Find

*The Find property specifies the name of an operation that locates an item among relational objects. (Default = Find\_\$target:c)*

## FindGenerate

The FindGenerate property specifies whether to generate a Find() operation for relations.

*Default = Cleared*

## Get

*The Get property specifies the name of an operation that retrieves the relation currently pointed to by the iterator. (Default = Get\_\$target:c)*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation by using an index. The ContainerTypes>::Relationtype::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the at() operation for the container to retrieve the item at the indexed position:  
\$cname-at(\$index)

*(Default = get\$cname:cAt)*

## GetAtGenerate

The GetAtGenerate property specifies whether to generate a getAt() operation for relations. The possible values are as follows:

- Checked - Generate a getAt() operation for relations.
- Cleared - Do not generate a getAt() operation for relations. Setting the GetAtGenerate property to False is one way to optimize your code for size.

*Default = Cleared*

## GetEnd

*The GetEnd property specifies the name of an operation that points the iterator to the last item in a collection. (Default = Get\_\$target:cEnd)*

## GetEndGenerate

The GetEndGenerate property specifies whether to generate a GetEnd() operation for relations.

*Default = Checked*

## GetGenerate

The GetGenerate property specifies whether to generate accessor operations for relations.

*Default = Checked*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key. For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps: \$cname-operator[](\$keyName)

*(Default = get\$cname:c\_Key)*

## GetKeyGenerate

The GetKeyGenerate property specifies whether to generate getKey() operations for relations. Setting this property to False is one way to optimize your code for size.

*Default = Checked*

## ImplementationEpilog

The ImplementationEpilog property enables you to add any code that you want to be added as verbatim text (to be ignored by Rational Rhapsody) to the end of the definition of a model element.. For example, you could wrap a section of code with an #ifdef-#endif pair, add compiler-specific keywords, or add a #pragma statement. For example, to specify that an operation is available only when the code is compiled with \_DEBUG, set the following properties for the operation:

- Set SpecificationProlog to #ifdef \_DEBUG cr.
- Set SpecificationEpilog to #endif.
- Set ImplementationProlog to #ifdef \_DEBUG cr.
- Set ImplementationEpilog to #endif.

The following table lists whether leading and trailing linefeeds are generated.

Metaclass	Leading Linefeed	Added?	Class	Yes	Package	No
-----------	------------------	--------	-------	-----	---------	----

*Default = Empty MultiLine*

## ImplementationProlog

The ImplementationProlog property enables you to add any code that you want to be added as verbatim text (to be ignored by the software) to the beginning of the definition of a model element. For example, you could wrap a section of code with an #ifdef-#endif pair, add compiler-specific keywords, or add a #pragma statement. For example, to specify that an operation is available only when the code is compiled with \_DEBUG, set the following properties for the operation:

- Set SpecificationProlog to #ifdef \_DEBUG cr.
- Set SpecificationEpilog to #endif.
- Set ImplementationProlog to #ifdef \_DEBUG cr.
- Set ImplementationEpilog to #endif.

The following table lists whether leading and trailing linefeeds are generated.

Metaclass	Trailing Linefeed Added?	Class	No	Package	Yes
-----------	--------------------------	-------	----	---------	-----

*Default = Empty MultiLine*

## ImplementWithStaticArray

The ImplementWithStaticArray property specifies whether to implement relations as static arrays. The possible values are as follows:

- Default - Rational Rhapsody provides the appropriate implementation for all fixed and bounded relations.
- FixedAndBounded - All fixed and bounded relations are generated into static arrays.

*(Default = FixedAndBounded)*

## InitializeComposition

The InitializeComposition property controls how a composition relation is initialized. The possible values are as follows:

- InInitializer
- InRecordType
- None

*(Default = InInitializer)*

## Inline

The Inline property specifies how inline operations are generated. Which operations are affected by the Inline property depends on the metaclass:

- Attribute - Applies only to operations that handle attributes (such as accessors and mutators)

- Operation - Applies to all operations
- Relation - Applies only to operations that handle relations

Inlining in Rational Rhapsody Developer for C Beginning with Rational Rhapsody Developer for C Version 4.2, you can inline attribute and relation accessors and mutators for increased code performance. The visibility of the inlined operations can be either public or private. The accessor can contain only a return statement, or more than just a return statement. For Rational Rhapsody Developer for C, there are two possible settings for this property:

- none - The operation is not generated inline. For example: `/* Mutator of Tank::ItsDishwasher relation */ void Tank_setItsDishwasher(struct Tank_t* const me, struct Dishwasher_t* p_Dishwasher) { if(p_Dishwasher != NULL) Dishwasher__setItsTank(p_Dishwasher, me); Tank__setItsDishwasher(me, p_Dishwasher); } /* Accessor to Tank::ItsDishwasher relation */ struct Dishwasher_t* Tank_getItsDishwasher( const struct Tank_t* const me) { return (struct Dishwasher_t*)me-itsDishwasher; }`
- in\_header - The operation is generated inline, as follows:
- Mutators are defined as macro definitions. For example: `/* Inline Mutator of Tank::ItsDishwasher relation */ #define Tank_setItsDishwasher(me, p_Dishwasher) \ { \ if((p_Dishwasher) != NULL) \ Dishwasher__setItsTank((p_Dishwasher), (me)); \ Tank__setItsDishwasher((me), (p_Dishwasher)); \ }`
- Accessors that contain only return statements are defined as macros (the return statement and the semicolon at the end of expression are omitted); other accessors are generated as operations. For example: `/* Inline Accessor to Tank::ItsDishwasher relation */ #define Tank_getItsDishwasher(me) ((me)-itsDishwasher)`
- If the attribute visibility is defined as Private, the macro definitions are placed in the implementation (.c) file.
- If the attribute visibility is defined as Public, the macro definitions are placed in the specification (.h) file.

Note the following:

- Each instance of the parameters for the macro is parenthesized.
- You cannot inline an accessor if it contains statements other than the return statement. For example, accessors to relations implemented by using RiCCollection cannot be generated as function-like macros.
- You cannot set the Inline property separately for specific helpers (for example, only mutators) - this property affects all helpers of the attribute or relation.
- If a multi-lined mutator macro is called as the body of the “then” part of an “if ...else” statement, you must enclose it in parentheses or it generates a compilation error. For example: `// Erroneous code: If (itsDishwasher != NULL) Tank_setItsDishwasher(me, itsDishwasher); Else Return; // Correct code: If (itsDishwasher != NULL) { Tank_setItsDishwasher(me, itsDishwasher); } Else Return;`

## IsAliased

The IsAliased property is a Boolean value that specifies whether attributes are aliased.

*Default = Cleared*

## Kind



The Kind property specifies the kind of operation that should be generated for an element. The kind of operations that can be generated is language-dependent (for example, virtual and abstract exist only in C++ and Java). In Java, Kind can be defined only for attributes and operations, but not for relations. This property affects class operations, in addition to accessors and mutators for relations and attributes. The possible values are as follows:

- common - Class operations and accessor/mutator are non-virtual.
- virtual - Class operations and accessor/mutator are virtual. This type is valid for C++ and Java only.
- abstract - Class operations and accessor/mutator are pure virtual. This type is valid for C++ and Java only.

*Default = common*

## MarkPrologEpilogInAnnotations

The MarkPrologEpilogInAnnotations property specifies whether to generate ignore annotations for the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties so they are ignored during roundtrip. When you insert code element declarations (variables, types, functions, and so on) in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, after a full roundtrip those elements are added to the model and are duplicated on the next code generation.

When you use the MarkPrologEpilogInAnnotations property, you can have Rational Rhapsody automatically ignore the information specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties instead of adding the ignore annotations manually. The possible values for the MarkPrologEpilogInAnnotations property are:

- None - Rational Rhapsody does not generate any annotations. Any models created before Version 4.1 automatically have this property setting.
- Ignore - Rational Rhapsody generates the `/// ignore annotation before the code specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, and generates the /// annotation after the code specified in those properties.`
- Auto - If the code in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties is one line (it does not contain any newline characters (`\n`)), no annotations are generated (the same behavior has the None setting). If there is more than one line, Rational Rhapsody generates the `/// ignore annotation before the code specified in the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties, and generates the /// annotation after the code specified in those properties (the same behavior as the Ignore setting).`

During roundtrip, any ignore annotations in the comments of the element are not included in its description. Because the Specification Prolog, Implementation Prolog, Specification Epilog, and Implementation Epilog properties are generated between the annotation and its declaration for the element, you cannot rename those elements on roundtrip. If you change the name of an element, it is removed from the model and added with the new name. Some model information (for example, property settings) might be lost.

*Default = Auto*

## ObjectInitialization

The ObjectInitialization property specifies what kind of initialization occurs for the initial instances of a configuration. The possible values are as follows:

- Full - Instances are initialized and their behavior is started.
- Creation - Instances are initialized but their behavior is not started.
- None - Instances are not initialized and their behavior is not started.

*(Default = Full)*

## Remove

The Remove property specifies the name of an operation that removes an item from a relation. (Default = Remove\_\$target:c)

## RemoveGenerate

The RemoveGenerate property specifies whether to generate a Remove() operation for relations.

*Default = Checked*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: \$cname-erase(\$keyName)

*(Default = remove\$cname:c\_Key)*

## RemoveKeyGenerate

The RemoveKeyGenerate property specifies whether to generate a removeKey() operation for qualified relations. Setting this property to False is one way to optimize your code for size.

*Default = Checked*

## RemoveKeyHelpersGenerate

The RemoveKeyHelpersGenerate property controls the generation of the relation helper methods (for example, \_removeItsX() and \_\_removeItsX()). The possible values are as follows:

- True - Generate the helpers whenever code generation analysis determines that the methods are needed.
- False - Never generate the helpers.
- FromModifier - Generate the helpers based on the value of the C\_CG::Relation::RemoveKey property.

*Default = Checked*

## SafeInitScalar

The SafeInitScalar property specifies whether to initialize scalar relations as null pointers.

*Default = Cleared*

## Set

*The Set property specifies the name of the mutator generated for scalar relations. (Default = Set\_\$target:c)*

## SetGenerate

The SetGenerate property specifies whether to generate mutators for relations.

*Default = Checked*

## Simplify

If you are using the Rational Rhapsody customizable code generation mechanism, the Simplify property can be used to change the way specific types of elements are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - The element is ignored.
- Copy - The element is copied from the original to the simplified model. It is not modified in any way.
- Default - Uses the standard simplification for this item, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for the element has been applied.

*Default = "Default"*

## SpecificationEpilog

The SpecificationEpilog property allows you to add code to the end of the declaration of a model element.

For example, you can use this property to add an `#ifdef` to indicate that an operation is available only when the code is compiled with `_DEBUG`, by setting the following properties:

- For SpecificationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For SpecificationEpilog, enter the value `#endif`
- For ImplementationProlog, enter the value `#ifdef _DEBUG` and a new line.

- For ImplementationEpilog, enter the value #endif

*Default = Blank*

## SpecificationProlog

The SpecificationProlog property adds code to the beginning of the declaration of a model element (such as a configuration or class). For example, to create an abstract class in Java, you can set the SpecificationProlog property for the class to "abstract."

You must include the space after the word "abstract." If the visibility for the class is set to default, the following class declaration is generated in the .java file: abstract class classname { ... }

The SpecificationProlog property allows you to add compiler-specific keywords, add a #pragma statement, or wrap a section of code with an #ifdef-#endif pair. For example, to specify that an operation is available only when the code is compiled with \_DEBUG, set the following properties for the operation:

- Set SpecificationProlog to #ifdef \_DEBUG cr.
- Set SpecificationEpilog to #endif.
- Set ImplementationProlog to #ifdef \_DEBUG cr.
- Set ImplementationEpilog to #endif.

The following table lists whether leading and trailing linefeeds are generated.

Metaclass	Leading Linefeed Added?	Trailing Linefeed Added?	Class	Yes	No	Package	Yes	Yes
(empty MultiLine)								

## Static

The Static property is a Boolean value that determines whether class-wide relations are enabled. Class-wide members of a class are shared between all instances of that class and are mapped as static. When a relation is tagged as static:

- The data member is generated as static (with the static keyword).
- The relation accessors are generated as static.
- The mutators of directional relations are generated as static. The mutators of symmetric relations are generated as common (non-static) operations.

Note the following behavior and restrictions:

- If there are links between instances based on static relations, code generation initializes all the valid links. In case of a limited relation size, the last initialization is preserved.
- When you generate instrumented code (animation or tracing), relation NOTIFY calls are not added to static relation mutators.
- Animation associates static relations with the class instances, not the class itself.
- In an instrumented application (animation or tracing), the static relations names appear in each instance node; however, the values of directional static relations are not visible.

You might also want to use the "Filter" facility in this window to refer to the definitions of these properties:

CG::Relation::Containment

Containertype::Relationtype::CreateStatic

Containertype::Relationtype::InitStatic

*Default = Cleared*

## Visibility

The Visibility property specifies the visibility of that kind of model element. Code generation maps the visibility specified for an element to the same visibility in the generated language. (Default = Public)

## Requirement

The Requirement metaclass controls whether code is generated for requirements.

## DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- `$Name` - the element name
- `$FullName` - the full path of the element (P1::P2::C.a)
- `$Description` - the text provided on the Description tab of the Features window for the element
- `$Requirements` - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the `$Requirements` keyword in the DescriptionTemplate property to get such comments generated for specific model elements.

- \$Login - the login name of the user who generated the file
- \$CodeGeneratedDate - the date on which the code was generated
- \$CodeGeneratedTime - the time at which the code was generated
- \$RhapsodyVersion - the version of Rhapsody that was used to generate the file
- For operations: \$Signature - the operation signature, \$Arguments - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: \$Type - the type of the attribute
- For arguments: \$Type - the argument type, \$Direction - the argument direction (In, Out, or InOut)
- For events: \$Arguments - the text provided on the Description tab of the Features window for the event arguments
- For association ends: \$Target - the other end of the association
- For requirements: \$ID - the ID specified for the requirement, \$Specification - the specification text for the requirement
- The values of tags can be included by using the syntax \$tag\_name. For example, if you use a tag called author for your classes, you can use the keyword \$author in the value of DescriptionTemplate in order to have the value of the author tag included in the element description.
- The values of properties can be included by using the syntax \$property\_name. For example, you can use the keyword \$Animate in the value of DescriptionTemplate in order to have the value of the Animate property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as \$Name)
- Tag keywords
- Property keywords

If the value of a keyword is a MultiLine value, then each new line (except the first one) starts with the value of the <lang>\_CG::Configuration::DescriptionBeginLine property, and each line ends with the value of the <lang>\_CG::Configuration::DescriptionEndLine property.

*Default = Realizes requirement \$Name #\$ID*

## DescriptionTemplateForImplementation

If you are generating C or C++ code from your model, you can specify that requirements should be generated as comments in the implementation files, next to the operations that realize the requirements. There is also an option to have requirements generated as comments in both the specification and implementation files.

If you use one of these two options, you use the DescriptionTemplateForImplementation property to specify the text of the comments that are generated in the implementation files.

*Default = Realizes requirement \$Name[[ #\$ID]][[: \$Specification]]*

## Generate

The Generate property can be used to control whether comments are generated in the code to reflect the fulfillment of a requirement by specific model elements.

In general, the policy for generating requirement comments in code is controlled by the `CG::Configuration::IncludeRequirementsAsComment` property. The Generate property can be used to override the global policy for specific groups of requirements, for example high-level requirements.

You can set the value of the Generate property to False for a stereotype and then apply the stereotype to requirements that do not have to be reflected in the generated code.

Note that if `IncludeRequirementsAsComment` is set to False, comments will not be generated for requirements even if the Generate property is set to True.

*Default = True*

## Simplify

If you are using the Rational Rhapsody customizable code generation mechanism, the Simplify property can be used to change the way specific types of elements are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - The element is ignored.
- Copy - The element is copied from the original to the simplified model. It is not modified in any way.
- Default - Uses the standard simplification for this item, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for the element has been applied.

*Default = "Default"*

## Solaris2

Environment settings (Compiler, framework libraries, and so on) for Solaris 2, by using the Sun compiler.

## AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the `AdaptorSearchPath` property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*(Default = \$(OMROOT)/LangC/osconfig/Solaris2)*

## **AdditionalReservedWords**

The `AdditionalReservedWords` property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use.

In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

## **BuildArgumentsInIDE**

The `BuildArgumentsInIDE` property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the `C_CG::[environment]::BuildCommandInIDE` property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the `BuildInIDE` property is set to `True`.

*Default = Blank*

## **BuildCommandInIDE**

The `BuildCommandInIDE` property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.



Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration. To change this property, use the Configuration window in the browser - do not change it by using the Properties window or by modifying the site.prp file.

Note that this property also affects the names of the framework libraries used in the link. The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*(Default = Debug)*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## CompileSwitches

The CompileSwitches property specifies the compiler switches.

*(Default = -I. -I\$OMDefaultSpecificationDirectory -I\$(OMROOT) -I\$(OMROOT)/LangC -I\$(OMROOT)/LangC/oxf \$(INST\_FLAGS) \$(INCLUDE\_PATH) \$(INST\_INCLUDES) -DSolaris2 \$OMCPPCompileCommandSet -c)*

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The default value is as follows:

```
@echo Compiling $OMFileImpPath @$ (CC) $OMFileCPPCompileSwitches -o $OMFileObjPath  
$OMFileImpPath
```

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component. The default values are as follows:

*(Default = -g)*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*(Default = -O)*

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

The default value is as follows:

```
$OMFileObjPath : $OMFileImpPath $OMFileSpecPath $OMFileDependencies
```

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: `$OMFileObjPath : $OMFileImpPath "*.bmp" "*.ico" "*.rc2"`

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction an IDE such as Eclipse, the EnableDebugIntegrationWithIDE property can be used to specify whether or not the IDE debugger should be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to True, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's exit() function is called automatically. The system function calls the destructors of framework instances in the order it determines. The EndApplicationCode property

provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an `endApplication()` function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to `endApplication()`, it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The `EntryPoint` property specifies the name of the main program for a given environment.

*(Default = main)*

You might also want to use the "Filter" facility in this window to refer to the definition of the `EntryPointDeclarationModifier` property for more information.

## ErrorMessageTokensFormat

The `ErrorMessageTokensFormat`, working with the `ParseErrorMessage` property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

`ErrorMessageTokensFormat` defines the number and location of tokens within the regular expression defined by the `ParseErrorMessage` property. `ErrorMessageTokens` has three parameters, each with an integer value:

- `TotalNumberOfTokens` - The number of tokens in the regular expression
- `FileTokenPosition` - The position of the file name token in the expression
- `LineTokenPosition` - The position of the line number token in the expression

The default value is as follows:

`TotalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2`

## ExeExtension

The `ExeExtension` property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the `C_CG::<Environment>::ExeName` property plus the value of the `ExeExtension` property.

*Default = Empty string*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the C\_CG::<Environment>::ExeExtension property.

*(Default = Blank)*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements. The file inclusions are generated in the makefile.

This default value is \$OMSpecIncludeInElements \$OMImpIncludeInElements.

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile.

Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

This default value is \$(OMROOT)/LangC/lib/sol2WebComponents\$(LIB\_EXT), \$(OMROOT)/lib/sol2WebServices\$(LIB\_EXT), -lsocket -lnsl.

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment. The default values are as follows:

*(Default = .c)*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

The C default value is "include."

## InvokeAnimatedExecutable

*Default = xterm -e \$executable -port \$port*

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*(Default = xterm -e \$executable)*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

*(Default = \$OMROOT/etc/sol2make \$makefile \$maketarget)*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*(Default = .a)*

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

*(Default = -g)*

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

*(Default = -O)*

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode. The default values are as follows:

*(Default = \$OMLinkCommandSet)*

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the C\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*(Default = .mak)*

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration. For example, the default content of the target type section of a C++ makefile for the Microsoft environment is as follows: ##### Target type (Debug/Release)

```
#####  
C++CompileDebug=$OMC++CompileDebug C++CompileRelease=$OMC++CompileRelease  
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease BuildSet=$OMBuildSet  
SUBSYSTEM=$OMSubSystem COM=$OMCOM RPFramWorkDll=$OMRPFramWorkDll  
ConfigurationC++CompileSwitches= $OMReusableStatechartSwitches  
$OMConfigurationC++CompileSwitches !IF "$ (RPFramWorkDll)" == "True"  
ConfigurationC++CompileSwitches= $(ConfigurationC++CompileSwitches) /D "FRAMEWORK_DLL"  
!ENDIF !IF "$ (COM)" == "True" SUBSYSTEM=/SUBSYSTEM:windows !ENDIF
```

**Compilation Flags** The compilation flags section of the makefile contains the default compilation flags stored in the CompileSwitches property. For example, the default content of the compilation flags section of a C++ makefile for the Microsoft environment is as follows: ##### Compilation flags

```
#####  
INCLUDE_QUALIFIER=/I LIB_PREFIX=MS
```

**Commands Definitions** The commands definition section of the makefile specifies programs to execute from the makefile. For example, the default commands definition section of a C++ makefile for the Microsoft environment is as follows: ##### Commands definition #####

```
##### RMDIR = rmdir LIB_CMD=link.exe -lib  
LINK_CMD=link.exe LIB_FLAGS=$OMConfigurationLinkSwitches  
LINK_FLAGS=$OMConfigurationLinkSwitches $(SUBSYSTEM) / MACHINE:I386
```

**Generated Macros** The generated macros section of the makefile contains a variable that expands to the Rational Rhapsody-generated macros in the makefile. For example: ##### Generated macros

```
##### $OMContextMacros  
OBJ_DIR=$OMObjectsDir !IF "$ (OBJ_DIR)" != "" CREATE_OBJ_DIR=if not exist $(OBJ_DIR) mkdir  
$(OBJ_DIR) CLEAN_OBJ_DIR= if exist $(OBJ_DIR) $(RMDIR) $(OBJ_DIR) !ELSE  
CREATE_OBJ_DIR= CLEAN_OBJ_DIR= !ENDIF
```

The \$OMContextMacros keyword expands several macros in the makefile. Each makefile macro has its own keyword. You can use these keywords separately to customize the makefile. The \$OMContextMacros variable modifies target-specific variables.

Replace the \$OMContextMacros line in the MakeFileContent property with the following:

```
FLAGSFILE=$OMFlagsFile RULESFILE=$OMRulesFile OMROOT=$OMROOT  
C_EXT=$OMImplExt H_EXT=$OMSpecExt OBJ_EXT=$OMObjExt EXE_EXT=$OMExeExt  
LIB_EXT=$OMLibExt INSTRUMENTATION=$OMInstrumentation TIME_MODEL=$OMTimeModel  
TARGET_TYPE=$OMTargetType TARGET_NAME=$OMTargetName $OMAllDependencyRule  
TARGET_MAIN=$OMTargetMain LIBS=$OMLibs INCLUDE_PATH=$OMIncludePath  
ADDITIONAL_OBJS=$OMAdditionalObjs OBJS= $OMObjs
```

**Predefined Macros** The predefined macros section of the makefile contains other macros than the Rational Rhapsody-generated macros specified in the generated macros section. For example, the default predefined macros section of a C++ makefile for the Microsoft environment is as follows:

```
##### Predefined macros #####  
##### $(OBJS) : $(INST_LIBS)  
$(OXF_LIBS) LIB_POSTFIX= !IF "$ (BuildSet)" == "Release" LIB_POSTFIX=R !ENDIF !IF  
"$ (TARGET_TYPE)" == "Executable" LinkDebug=$(LinkDebug) /DEBUG
```

```

LinkRelease=$(LinkRelease) /OPT:NOREF !ELSEIF "$(TARGET_TYPE)" == "Library"
LinkDebug=$(LinkDebug) /DEBUGTYPE:CV !ENDIF !IF "$(INSTRUMENTATION)" == "Animation"
INST_FLAGS=/D "OMANIMATOR" INST_INCLUDES=/I $(OMROOT)\LangCpp\om /I
$(OMROOT)\LangCpp\tom !IF "$(RPFrameWorkDll)" == "True" INST_LIBS=
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfanimdll$(LIB_POSTFIX)$(LIB_EXT)
!ELSE INST_LIBS= $(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomanim$(LIB_POSTFIX)$(LIB_EXT)
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfirst$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX)$(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)" == "Tracing" INST_FLAGS=/D
"OMTRACER" INST_INCLUDES=/I $(OMROOT)\LangCpp\om /I $(OMROOT)\LangCpp\tom !IF
"$(RPFrameWorkDll)" == "True" INST_LIBS= OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfracedll$(LIB_POSTFIX)$(LIB_EXT) !ELSE
INST_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)tomtrace$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomtrace$(LIB_POSTFIX)$(LIB_EXT) OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfirst$(LIB_POSTFIX)$(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX)$(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)" == "None" INST_FLAGS=
INST_INCLUDES= INST_LIBS= !IF "$(RPFrameWorkDll)" == "True"
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfdll$(LIB_POSTFIX)$(LIB_EXT) !ELSE
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$(LIB_POSTFIX)$(LIB_EXT) !ENDIF
SOCK_LIB= !ELSE !ERROR An invalid Instrumentation $(INSTRUMENTATION) is specified.
!ENDIF

```

Generated Dependencies The generated dependencies section of the makefile contains a variable that expands to Rational Rhapsody-generated dependencies and compilation instructions. For example, the generated dependencies section of a C++ makefile for the Microsoft environment is as follows:

```

##### Generated dependencies #####
##### SOMContextDependencies
$OMFileObjPath : $OMMainImplementationFile $(OBJS) $(CPP) $(ConfigurationCPPCompileSwitches)
/Fo"$OMFileObjPath" $OMMainImplementationFile

```

Linking Instructions The linking instructions section of the makefile contains the predefined linking instructions. For example, the default linking instructions section of a C++ makefile for the Microsoft environment is as follows:

```

##### Linking instructions #####
#####
$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS) $OMFileObjPath
$OMMakefileName $OMModelLibs @echo Linking $(TARGET_NAME)$(EXE_EXT) $(LINK_CMD)
$OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \ $(OXF_LIBS) \
$(SOCK_LIB) \ $(LINK_FLAGS) /out:$(TARGET_NAME)$(EXE_EXT)
$(TARGET_NAME)$(LIB_EXT) : $(OBJS) $(ADDITIONAL_OBJS) $OMMakefileName @echo
Building library @@ $(LIB_CMD) $(LIB_FLAGS) /out:$(TARGET_NAME)$(LIB_EXT) $(OBJS)
$(ADDITIONAL_OBJS) clean: @echo Cleanup $OMCleanOBJS if exist $OMFileObjPath erase
$OMFileObjPath if exist *$(OBJ_EXT) erase *$(OBJ_EXT) if exist $(TARGET_NAME).pdb erase
$(TARGET_NAME).pdb if exist $(TARGET_NAME)$(LIB_EXT) erase
$(TARGET_NAME)$(LIB_EXT) if exist $(TARGET_NAME).ilk erase $(TARGET_NAME).ilk if exist
$(TARGET_NAME)$(EXE_EXT) erase $(TARGET_NAME)$(EXE_EXT) $(CLEAN_OBJ_DIR)

```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.



Note that this property only specifies the first part of the filename. The extension is specified by using the C\_CG::<Environment>::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## **NullValue**

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## **ObjCleanCommand**

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*(Default = \$(RM) \$OMFileObjPath)*

## **ObjectName**

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = empty string*

## **ObjectsDirectory**

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile. The default values are as follows:

*Default = empty string*

## **ObjExtension**

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*(Default = .o)*

## **OSFileSystemCaseSensitive**

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Checked*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning)

Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*(Default = "[^:]+)("[^"]\*"|"[^"]\*"|line ([0-9]+)[:])*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation.

*(Default = /)*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Checked*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running the product, whereas the target is the machine running the application.

To run remotely, the UseRemoteHost property must be set to True. If UseRemoteHost is True and RemoteHost is blank, the current host name is used for the remote host. You can use this as a workaround if you have problems running animated applications on Windows 95. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = empty string)*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

The extension ".h" is the default for C.

## UnixLineTerminationStyle

The UnixLineTerminationStyle property specifies whether generated files use the UNIX end-of-line style.

If this property is set to Cleared, the end-of-line style depends on the host type (for example, DOS style on Windows machines, and UNIX style on Solaris machines).

*Default = Checked*

## UnixPathNameForOMROOT

The UnixPathNameForOMROOT property specifies whether the makefile must include UNIXstyle path names.

The pRISM compilers do not tolerate DOS-style path conventions. If you do not set this property correctly, there might be many compilation problems.

*Default = Checked*

## UpdateBuildSettingsInIDE

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## UseRemoteHost

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Checked*

## Solaris2GNU

Environment settings (Compiler, framework libraries, and so on) for Solaris 2, by using the GCC compiler.

## AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is

added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the `AdaptorSearchPath` property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*(Default = \$(OMROOT)/LangC/osconfig/Solaris2)*

## **AdditionalReservedWords**

The `AdditionalReservedWords` property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

## **BuildArgumentsInIDE**

The `BuildArgumentsInIDE` property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the `C_CG::[environment]::BuildCommandInIDE` property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the `BuildInIDE` property is set to `True`.

*Default = Blank*

## **BuildCommandInIDE**

The `BuildCommandInIDE` property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration. To change this property, use the Configuration window in the browser - do not change it by using the Properties window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link.

The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*(Default = Debug)*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## CompileSwitches

The CompileSwitches property specifies the compiler switches.

*(Default = -I. -I\$OMDefaultSpecificationDirectory -I\$(OMROOT) -I\$(OMROOT)/LangC -I\$(OMROOT)/LangC/oxf \$(INST\_FLAGS) \$(INCLUDE\_PATH) \$(INST\_INCLUDES) -DSolaris2 \$OMCPPCompileCommandSet -c)*

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The default value is as follows:

```
@echo Compiling $OMFileImpPath @$ (CC) $OMFileCPPCompileSwitches -o $OMFileObjPath  
$OMFileImpPath
```

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component. The default values are as follows:

*(Default = -g)*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*(Default = -O)*

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

The default value is as follows:

```
$OMFileObjPath : $OMFileImpPath $OMFileSpecPath $OMFileDependencies
```

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: `$OMFileObjPath : $OMFileImpPath "*.bmp" "*.ico" "*.rc2"`

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction an IDE such as Eclipse, the EnableDebugIntegrationWithIDE property can be used to specify whether or not the IDE debugger should be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to True, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's exit() function is called automatically. The system function calls the destructors of framework instances in the order it determines. The EndApplicationCode property

provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an `endApplication()` function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to `endApplication()`, it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The EntryPoint property specifies the name of the main program for a given environment.

*(Default = main)*

You might also want to use the "Filter" facility in this window to refer to the definition of the `EntryPointDeclarationModifier` property for more information.

## ErrorMessageTokensFormat

The `ErrorMessageTokensFormat`, working with the `ParseErrorMessage` property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. `ErrorMessageTokensFormat` defines the number and location of tokens within the regular expression defined by the `ParseErrorMessage` property.

`ErrorMessageTokens` has three parameters, each with an integer value:

- `TotalNumberOfTokens` - The number of tokens in the regular expression
- `FileTokenPosition` - The position of the file name token in the expression
- `LineTokenPosition` - The position of the line number token in the expression

The default value is as follows:

`TotalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2`

## ExeExtension

The `ExeExtension` property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the `C_CG::<Environment>::ExeName` property plus the value of the `ExeExtension` property.

*Default = Empty string*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the C\_CG::<Environment>::ExeExtension property.

*(Default = Blank)*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements. The file inclusions are generated in the makefile.

This default value is \$OMSpecIncludeInElements \$OMImpIncludeInElements.

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile.

Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

This default value is \$(OMROOT)/LangC/lib/sol2WebComponents\$(LIB\_EXT), \$(OMROOT)/lib/sol2WebServices\$(LIB\_EXT), -lsocket -lnsl.

## HasIDEInterface

The HasIDEInterface property is a Boolean value that specifies whether IDE support is enabled.

If IDE support is enabled (Checked), the IDEInterfaceDLL property points to an IDE adapter that provides connection to the IDE. If the property is set to Cleared, IDE support is disabled and IDE services are not attempted. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The default value is Checked.

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated



implementation files for a given language and environment. The default values are as follows:

*(Default = .c)*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

The C default value is "include."

## InvokeAnimatedExecutable

The AnimationPortRange value in the rhapsody.ini file makes it possible to use animation with multiple instances of Rhapsody running simultaneously.

Ordinarily, when you run an application with animation, the executable is launched using the command specified by the InvokeExecutable property. However, if you have specified a value greater than zero for AnimationPortRange in your rhapsody.ini file, the executable is launched using the command specified by the InvokeAnimatedExecutable property. By default, the value of InvokeAnimatedExecutable includes a reference to the port to use for animation.

*Default = xterm -e \$executable -port \$port*

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*(Default = xterm -e \$executable)*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT/etc/vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

*(Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT/etc/vxmake.bat\" \$makefile \$maketarget")*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*(Default = .a)*

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

*(Default = -g)*

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

*Default = empty string*

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode. The default values are as follows:

*(Default = \$OMLinkCommandSet)*

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the C.CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

(Default = .mak)

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration. For example, the default content of the target type section of a C++ makefile for the Microsoft environment is as follows: ##### Target type (Debug/Release)

```
#####  
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease  
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease BuildSet=$OMBuildSet  
SUBSYSTEM=$OMSubSystem COM=$OMCOM RPFrameWorkDll=$OMRPFrameWorkDll  
ConfigurationCPPCompileSwitches= $OMReusableStatechartSwitches  
$OMConfigurationCPPCompileSwitches !IF "$(RPFrameWorkDll)" == "True"  
ConfigurationCPPCompileSwitches= $(ConfigurationCPPCompileSwitches) /D "FRAMEWORK_DLL"  
!ENDIF !IF "$(COM)" == "True" SUBSYSTEM=/SUBSYSTEM:windows !ENDIF
```

**Compilation Flags** The compilation flags section of the makefile contains the default compilation flags stored in the CompileSwitches property. For example, the default content of the compilation flags section of a C++ makefile for the Microsoft environment is as follows: ##### Compilation flags

```
#####  
INCLUDE_QUALIFIER=/I LIB_PREFIX=MS
```

**Commands Definitions** The commands definition section of the makefile specifies programs to execute from the makefile. For example, the default commands definition section of a C++ makefile for the Microsoft environment is as follows: ##### Commands definition #####

```
##### RMDIR = rmdir LIB_CMD=link.exe -lib  
LINK_CMD=link.exe LIB_FLAGS=$OMConfigurationLinkSwitches  
LINK_FLAGS=$OMConfigurationLinkSwitches $(SUBSYSTEM) / MACHINE:I386
```

**Generated Macros** The generated macros section of the makefile contains a variable that expands to the Rational Rhapsody-generated macros in the makefile. For example: ##### Generated macros

```
##### $OMContextMacros  
OBJ_DIR=$OMObjectsDir !IF "$(OBJ_DIR)"!="" CREATE_OBJ_DIR=if not exist $(OBJ_DIR) mkdir  
$(OBJ_DIR) CLEAN_OBJ_DIR= if exist $(OBJ_DIR) $(RMDIR) $(OBJ_DIR) !ELSE
```

```
CREATE_OBJ_DIR= CLEAN_OBJ_DIR= !ENDIF
```

The \$OMContextMacros keyword expands several macros in the makefile. Each makefile macro has its own keyword. You can use these keywords separately to customize the makefile. The \$OMContextMacros variable modifies target-specific variables. Replace the \$OMContextMacros line in the MakeFileContent property with the following: FLAGSFILE=\$OMFlagsFile  
RULESFILE=\$OMRulesFile OMROOT=\$OMROOT C\_EXT=\$OMImplExt H\_EXT=\$OMSpecExt  
OBJ\_EXT=\$OMObjExt EXE\_EXT=\$OMExeExt LIB\_EXT=\$OMLibExt  
INSTRUMENTATION=\$OMInstrumentation TIME\_MODEL=\$OMTimeModel  
TARGET\_TYPE=\$OMTargetType TARGET\_NAME=\$OMTargetName \$OMAllDependencyRule  
TARGET\_MAIN=\$OMTargetMain LIBS=\$OMLibs INCLUDE\_PATH=\$OMIncludePath  
ADDITIONAL\_OBJS=\$OMAdditionalObjs OBJS= \$OMObjs

**Predefined Macros** The predefined macros section of the makefile contains other macros than the Rational Rhapsody-generated macros specified in the generated macros section. For example, the default predefined macros section of a C++ makefile for the Microsoft environment is as follows:

```
##### Predefined macros #####
##### $(OBJS) : $(INST_LIBS)
$(OXF_LIBS) LIB_POSTFIX= !IF "$$(BuildSet)"=="Release" LIB_POSTFIX=R !ENDIF !IF
"$(TARGET_TYPE)"=="Executable" LinkDebug=$(LinkDebug) /DEBUG
LinkRelease=$(LinkRelease) /OPT:NOREF !ELSEIF "$(TARGET_TYPE)"=="Library"
LinkDebug=$(LinkDebug) /DEBUGTYPE:CV !ENDIF !IF "$(INSTRUMENTATION)"=="Animation"
INST_FLAGS=/D "OMANIMATOR" INST_INCLUDES=/I $(OMROOT)\LangCpp\aoom /I
$(OMROOT)\LangCpp\tom !IF "$(RPFrameWorkDll)"=="True" INST_LIBS=
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxanimdll$(LIB_POSTFIX) $(LIB_EXT)
!ELSE INST_LIBS= $(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomanim$(LIB_POSTFIX) $(LIB_EXT)
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfinst$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)"=="Tracing" INST_FLAGS=/D
"OMTRACER" INST_INCLUDES=/I $(OMROOT)\LangCpp\aoom /I $(OMROOT)\LangCpp\tom !IF
"$(RPFrameWorkDll)"=="True" INST_LIBS= OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfracedll$(LIB_POSTFIX) $(LIB_EXT) !ELSE
INST_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)tomtrace$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomtrace$(LIB_POSTFIX) $(LIB_EXT) OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfinst$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)"=="None" INST_FLAGS=
INST_INCLUDES= INST_LIBS= !IF "$(RPFrameWorkDll)"=="True"
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfdll$(LIB_POSTFIX) $(LIB_EXT) !ELSE
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB= !ELSE !ERROR An invalid Instrumentation $(INSTRUMENTATION) is specified.
!ENDIF
```

**Generated Dependencies** The generated dependencies section of the makefile contains a variable that expands to Rational Rhapsody-generated dependencies and compilation instructions. For example, the generated dependencies section of a C++ makefile for the Microsoft environment is as follows:

```
##### Generated dependencies #####
##### $OMContextDependencies
$OMFileObjPath : $OMMainImplementationFile $(OBJS) $(CPP) $(ConfigurationCPPCompileSwitches)
/Fo"$OMFileObjPath" $OMMainImplementationFile
```

**Linking Instructions** The linking instructions section of the makefile contains the predefined linking instructions. For example, the default linking instructions section of a C++ makefile for the Microsoft environment is as follows: ##### Linking instructions #####

```
#####
$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS) $OMFileObjPath
$OMMakefileName $OMModelLibs @echo Linking $(TARGET_NAME)$(EXE_EXT) $(LINK_CMD)
$OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \ $(OXF_LIBS) \
$(SOCK_LIB) \ $(LINK_FLAGS) /out:$(TARGET_NAME)$(EXE_EXT)
$(TARGET_NAME)$(LIB_EXT) : $(OBJS) $(ADDITIONAL_OBJS) $OMMakefileName @echo
Building library $@ $(LIB_CMD) $(LIB_FLAGS) /out:$(TARGET_NAME)$(LIB_EXT) $(OBJS)
$(ADDITIONAL_OBJS) clean: @echo Cleanup $OMCleanOBJS if exist $OMFileObjPath erase
$OMFileObjPath if exist *$(OBJ_EXT) erase *$(OBJ_EXT) if exist $(TARGET_NAME).pdb erase
$(TARGET_NAME).pdb if exist $(TARGET_NAME)$(LIB_EXT) erase
$(TARGET_NAME)$(LIB_EXT) if exist $(TARGET_NAME).ilk erase $(TARGET_NAME).ilk if exist
$(TARGET_NAME)$(EXE_EXT) erase $(TARGET_NAME)$(EXE_EXT) $(CLEAN_OBJ_DIR)
```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the C\_CG::<Environment>::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## ObjCleanCommand

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*(Default = \$(RM) \$OMFileObjPath)*

## ObjectName

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = empty string*

## ObjectsDirectory

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile. The default values are as follows:

*Default = empty string*

## ObjExtension

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*(Default = .o)*

## OSFileSystemCaseSensitive

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Cleared*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning)

Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*(Default = ([^:]+):([0-9]+):)*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation.

*(Default = /)*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Checked*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running the product, whereas the target is the machine running the application. To run remotely, the UseRemoteHost property must be set to True.

If UseRemoteHost is True and RemoteHost is blank, the current host name is used for the remote host. You can use this as a workaround if you have problems running animated applications on Windows 95. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = empty string)*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

The extension ".h" is the default for C.

## UpdateBuildSettingsInIDE

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## UseRemoteHost

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Checked*

## State

The State metaclass contains code generation properties related to states.

## DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be

generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- `$Name` - the element name
- `$FullName` - the full path of the element (P1::P2::C.a)
- `$Description` - the text provided on the Description tab of the Features window for the element
- `$Requirements` - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the `$Requirements` keyword in the `DescriptionTemplate` property to get such comments generated for specific model elements.
- `$Login` - the login name of the user who generated the file
- `$CodeGeneratedDate` - the date on which the code was generated
- `$CodeGeneratedTime` - the time at which the code was generated
- `$RhapsodyVersion` - the version of Rhapsody that was used to generate the file
- For operations: `$Signature` - the operation signature, `$Arguments` - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: `$Type` - the type of the attribute
- For arguments: `$Type` - the argument type, `$Direction` - the argument direction (In, Out, or InOut)
- For events: `$Arguments` - the text provided on the Description tab of the Features window for the event arguments
- For association ends: `$Target` - the other end of the association
- For requirements: `$ID` - the ID specified for the requirement, `$Specification` - the specification text for the requirement
- The values of tags can be included by using the syntax `$tag_name`. For example, if you use a tag called `author` for your classes, you can use the keyword `$author` in the value of `DescriptionTemplate` in order to have the value of the `author` tag included in the element description.
- The values of properties can be included by using the syntax `$property_name`. For example, you can use the keyword `$Animate` in the value of `DescriptionTemplate` in order to have the value of the `Animate` property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as `$Name`)
- Tag keywords



- Property keywords

If the value of a keyword is a MultiLine value, then each new line (except the first one) starts with the value of the `<lang>_CG::Configuration::DescriptionBeginLine` property, and each line ends with the value of the `<lang>_CG::Configuration::DescriptionEndLine` property.

*Default = State \$Name [[Description: \$Description]]*

## EncloseFlowchartActionDescriptionTemplate

Ordinarily, when you specify the text to use as the value of the property `DescriptionTemplate` for a type of element, it is your responsibility to specify the comment symbols to use at the beginning and end of the text by providing values for the configuration-level properties `DescriptionBeginLine` and `DescriptionEndLine`.

Prior to release 8.1.3, the descriptions for actions in flowcharts were an exception to this rule - comment symbols were added automatically by Rhapsody. This behavior was changed in 8.1.3 to align with the behavior for other types of elements. To make it possible for users to restore the previous code generation behavior for actions in flowcharts, the property `EncloseFlowchartActionDescriptionTemplate` was added in release 8.1.3 with a default value of `False` (value is `True` in the backward compatibility settings).

*Default = False (True in the backward compatibility settings)*

## Statechart

The Statechart metaclass contains the properties that control statechart code generation.

### AddAnnotationToAllTransitions

Prior to version 8.0, the code for certain transitions did not have Rhapsody annotations preceding it, resulting in problems with actions like roundtripping. Beginning in 8.0, all transition code is preceded by an appropriate annotation. To preserve the previous code generation behavior for pre-8.0 models, the `lang_CG::Statechart::AddAnnotationToAllTransitions` property was added to the backward compatibility settings for C and C++ with a value of `False`.

*Default = False*

### AddDescriptionToActions

Prior to version 8.0, if no description was provided for an action in an activity, the generated code included a comment that reflected the description provided for the class that the action belongs to. Beginning in 8.0, the generated code no longer reflects the description of the owner class in such cases. To preserve the previous code generation behavior for pre-8.0 models, the `lang_CG::Statechart::AddDescriptionToActions` property was added to the backward compatibility

settings for C and C++ with a value of True.

*Default = True*

## GenerateActionOnExitOrderForNestedStatechartOldWay

Before version 7.5.3, the code generated for actions on exit was not put in the correct location in the generated code. This was corrected in version 7.5.3. In order to maintain the previous code generation behavior for older models, a property called [lang]\_CG::Statechart::GenerateActionOnExitOrderForNestedStatechartOldWay was added to the C, CPP, and Java backward compatibility profiles for 7.5.3 with a value of True.

## Simplify

If you are using the Rational Rhapsody customizable code generation mechanism, the Simplify property can be used to change the way specific types of elements are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- None - The element is ignored.
- Copy - The element is copied from the original to the simplified model. It is not modified in any way.
- Default - Uses the standard simplification for this item, as defined in Rational Rhapsody.
- ByUser - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- ByUserPostDefault - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for the element has been applied.

*Default = "Default"*

## StatechartImplementation

Prior to version 7.3 of Rational Rhapsody, the transition-handling code that is generated by Rational Rhapsody used a switch statement to represent the possible states. Beginning with version 7.3, this code uses an if/else structure. To allow older models to use the previous code generation behavior, a property called StatechartImplementation was added to the Pre73 profiles for compatibility with earlier versions. The possible values for the property are:

- SwitchOnly - transition-handling code uses a switch statement to represent the possible states
- Default - the transition-handling code uses an if/else structure to represent the possible states

*Default = SwitchOnly*

## StatechartStateOperations

This property determines whether the code is generated for this feature.

Rational Rhapsody provides a mechanism for serialization of reactive instances. By setting a number of Rational Rhapsody properties, you can have methods added to the generated code, which you can then use to implement serialization.

The possible values for this property are as follows:

- None - code is not generated for the feature
- WithoutReactive - Rational Rhapsody does not generate calls to OMReactive
- WithReactive - Rational Rhapsody generates calls to OMReactive

*(Default = None)*

The other C properties used in the serialization methods are as follows:

- C\_CG::Framework::ReactiveGetStateCall
- C\_CG::Framework::ReactiveSetStateCall
- C\_CG::Framework::ReactiveStateType

## Transition

The Transition metaclass contains code generation properties related to transitions.

### DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- \$Name - the element name
- \$FullName - the full path of the element (P1::P2::C.a)
- \$Description - the text provided on the Description tab of the Features window for the element
- \$Requirements - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the \$Requirements keyword in the DescriptionTemplate property to get such comments generated for specific model

elements.

- `$Login` - the login name of the user who generated the file
- `$CodeGeneratedDate` - the date on which the code was generated
- `$CodeGeneratedTime` - the time at which the code was generated
- `$RhapsodyVersion` - the version of Rhapsody that was used to generate the file
- For operations: `$Signature` - the operation signature, `$Arguments` - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: `$Type` - the type of the attribute
- For arguments: `$Type` - the argument type, `$Direction` - the argument direction (In, Out, or InOut)
- For events: `$Arguments` - the text provided on the Description tab of the Features window for the event arguments
- For association ends: `$Target` - the other end of the association
- For requirements: `$ID` - the ID specified for the requirement, `$Specification` - the specification text for the requirement
- The values of tags can be included by using the syntax `$tag_name`. For example, if you use a tag called `author` for your classes, you can use the keyword `$author` in the value of `DescriptionTemplate` in order to have the value of the `author` tag included in the element description.
- The values of properties can be included by using the syntax `$property_name`. For example, you can use the keyword `$Animate` in the value of `DescriptionTemplate` in order to have the value of the `Animate` property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as `$Name`)
- Tag keywords
- Property keywords

If the value of a keyword is a `MultiLine` value, then each new line (except the first one) starts with the value of the `<lang>_CG::Configuration::DescriptionBeginLine` property, and each line ends with the value of the `<lang>_CG::Configuration::DescriptionEndLine` property.

*Default = `[[Description: $Description]]`*

## Type

The `Type` metaclass contains a property that affects the visibility of data types.

### AnimEnumerationTypeImage

The `AnimEnumerationTypeImage` property is a Boolean value that determines whether the `Image` attribute is used for enumerated types when by using animation.

*Default = Cleared*

## AnimSerializeOperation

The AnimSerializeOperation property specifies the name of an external function used to animate all attributes and arguments that are of that type. Rational Rhapsody can animate (display) the values of simple types and one-dimensional arrays without any problem. To display the current values of such attributes during an animation session, run the features window for the instance.

However, if you want to animate a more complex type, such as a date, the type must be converted to a string (char \*) for Rational Rhapsody to display it. This is generally done by writing a global function, an instrumentation function, that takes one argument of the type you want to display, and returns a char \*. You must disable animation of the instrumentation function itself (by using the Animate and AnimateArguments properties for the function).

For example, you can have a type tDate, defined as follows: `typedef struct date { int day; int month; int year; } %s;` You can have an object with an attribute count of type int, and an attribute date of type tDate. The object can have an initializer with the following body: `me-date.month = 5; me-date.day = 12; me-date.year = 2000;` If you want to animate the date attribute, the AnimSerializeOperation property for date must be set to the name of a function that converts the type tDate to char \*. For example, you can set the property to a function named showDate. This function name must be entered without any parentheses. It must take an attribute of type tDate and return a char \*. The Animate and AnimateArguments properties for the showDate function must be set to False.

The implementation of the showDate function might be as follows: `showDate(tDate aDate) { char* buff; buff = (char*) malloc(sizeof(char) * 20); sprintf(buff,"%d %d %d", aDate.month,aDate.day,aDate.year); return buff; }`

When you run this model with animation, instances of this object displays a value of 5 12 2000 for the date attribute in the browser. If the showDate function is defined in the same class that the attribute belongs to and the function is not static, the AnimSerializeOperation property value should be similar to the following:

`myReal-showDate`

This value shows that the function is called from the serializeAttributes function, located in the class OMAAnimatedclassname. The showDate function must allocate memory for the returned string by way of the malloc/alloc/calloc function in C, or the new operator in C++. Otherwise, the system shuts down unexpectedly.

*Default = empty string*

## AnimUnserializeOperation

The AnimUnserializeOperation property converts a string to the value of an element (the opposite of the AnimSerializeOperation property). Unserialize functions are used for event generation or operation invocation by using the Animation toolbar to convert the string (received from the user) to the value of the event or operation before the event generation or operation invocation.

For example, your serialization operation might look similar to the following:

```
char* myX2String(const Rec f) { char* cS = new char[OutputStringLength]; /* conversion from the Rec
type to string */ return (cS); }
```

The unserialization operation would be:

```
Rec myString2X (char* C, Rec T) { T = new Trc; /* conversion of the string C to the Rec type */ delete C;
return (T); }
```

*Default = empty string*

## AnimUseMultipleSerializationFunctions

The AnimSerializeOperation and AnimUnserializeOperation properties are used to specify user-provided functions for serialization/unserialization of objects to allow inclusion of such objects in animation.

Because Rational Rhapsody allows you to fine-tune code generation of arguments by using the In, Out, InOut, and TriggerArgument properties, you might need to provide multiple serialization/unserialization functions to handle these different types of arguments. You can use the AnimUseMultipleSerializationFunctions property to instruct Rational Rhapsody to use multiple user-provided serialization/unserialization functions.

If you set the value of this property to Checked, Rational Rhapsody searches for user-provided serialization functions whose names consist of the string entered for the AnimSerializeOperation property and the suffixes "In", "Out", "InOut", and "TriggerArgument".

The same is true for unserialization functions. However, for unserialization, Rational Rhapsody cannot handle Out arguments, so the relevant suffixes are "In", "InOut", and "TriggerArgument".

Since the AnimSerializeOperation and AnimUnserializeOperation properties exist under both the Type metaclass and the Class metaclass, the AnimUseMultipleSerializationFunctions property also exists under both these metaclasses.

*Default = Cleared*

## DeclarationModifier

The DeclarationModifier property is used to enable Rational Rhapsody to reverse engineer non-standard keywords that appear in type declarations. Keywords that appear between the type of the type (structure, enumeration, or union) and the type name are stored as the value of this property, and the property is then used during code generation to re-create the original code.

Since this is a code generation property, it can also be used to add non-standard keywords to code even when reverse engineering is not used.

This property is used in conjunction with the PreDeclarationModifier and PostDeclarationModifier properties.

*Default = Blank*

## DeclarationPosition

The DeclarationPosition property specifies where the type declaration is displayed. The possible values are as follows:

- BeforeClassRecord - The type declaration is displayed before the class record (CR) declaration if CR has a visibility set to public, and before the class record forward declaration if CR has a visibility set to private.
- AfterClassRecord - The type declaration is displayed after the class record declaration if CR has a visibility set to public, and after the class record forward declaration if CR has a visibility set to private.
- StartOfDeclaration - The type declaration is displayed among the first declarations (together with other types having the same settings) in the public section if CR has a visibility set to public, and among the first declarations in the private section (together with other types having the same settings) if CR has a visibility set to private.
- EndOfDeclaration - The type declaration is displayed among the last declarations (together with other types having the same settings) in the public section if CR has a visibility set to public, and among the last declarations in the private section (together with other types having the same settings) if CR has a visibility set to private.

*If the C\_CG::Type::Visibility property is set to "Body", no matter the settings of C\_CG::Type::DeclarationPosition property, the type declaration still displays in the package body. (Default = BeforeClassRecord)*

## DefaultValue

In generated C code, there are cases where operations that belong to an interface can end up returning an uninitialized variable.

The DefaultValue property allows you to define a value, such as null, that can be returned in such situations.

Providing a value for this property also prevents problematic return values in situations where your application tries to call an operation that is part of a required interface for a port, but the service is not available.

*Default = Blank*

## DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- `$Name` - the element name
- `$FullName` - the full path of the element (`P1::P2::C.a`)
- `$Description` - the text provided on the Description tab of the Features window for the element
- `$Requirements` - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the `$Requirements` keyword in the `DescriptionTemplate` property to get such comments generated for specific model elements.
- `$Login` - the login name of the user who generated the file
- `$CodeGeneratedDate` - the date on which the code was generated
- `$CodeGeneratedTime` - the time at which the code was generated
- `$RhapsodyVersion` - the version of Rhapsody that was used to generate the file
- For operations: `$Signature` - the operation signature, `$Arguments` - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: `$Type` - the type of the attribute
- For arguments: `$Type` - the argument type, `$Direction` - the argument direction (In, Out, or InOut)
- For events: `$Arguments` - the text provided on the Description tab of the Features window for the event arguments
- For association ends: `$Target` - the other end of the association
- For requirements: `$ID` - the ID specified for the requirement, `$Specification` - the specification text for the requirement
- The values of tags can be included by using the syntax `$tag_name`. For example, if you use a tag called `author` for your classes, you can use the keyword `$author` in the value of `DescriptionTemplate` in order to have the value of the `author` tag included in the element description.
- The values of properties can be included by using the syntax `$property_name`. For example, you can use the keyword `$Animate` in the value of `DescriptionTemplate` in order to have the value of the `Animate` property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as `$Name`)
- Tag keywords
- Property keywords

If the value of a keyword is a `MultiLine` value, then each new line (except the first one) starts with the value of the `<lang>_CG::Configuration::DescriptionBeginLine` property, and each line ends with the value of the `<lang>_CG::Configuration::DescriptionEndLine` property.

*Default = Blank*



## EnumerationAsTypedef

The EnumerationAsTypedef property specifies whether the generated enum should be wrapped by a typedef. This property is applicable to enumeration types in C and C++.

*Default = Checked*

## ImplementationEpilog

The ImplementationEpilog property enables you to add any code that you want to be added as verbatim text (to be ignored by Rational Rhapsody) to the end of the definition of a model element.

For example, you could wrap a section of code with an #ifdef-#endif pair, add compiler-specific keywords, or add a #pragma statement. For example, to specify that an operation is available only when the code is compiled with \_DEBUG, set the following properties for the operation:

- Set SpecificationProlog to #ifdef \_DEBUG cr.
- Set SpecificationEpilog to #endif.
- Set ImplementationProlog to #ifdef \_DEBUG cr.
- Set ImplementationEpilog to #endif.

The following table lists whether leading and trailing linefeeds are generated.

Metaclass	Leading Linefeed	Added?	Class	Yes	Package	No
-----------	------------------	--------	-------	-----	---------	----

*Default = Empty MultiLine*

## ImplementationName

The ImplementationName property enables you to give a type one model name and generate it with another name.

*Default = Empty string*

## ImplementationProlog

The ImplementationProlog property enables you to add any code that you want to be added as verbatim text (to be ignored by the software) to the beginning of the definition of a model element.

For example, you could wrap a section of code with an #ifdef-#endif pair, add compiler-specific keywords, or add a #pragma statement. For example, to specify that an operation is available only when the code is compiled with \_DEBUG, set the following properties for the operation:

- Set SpecificationProlog to #ifdef \_DEBUG cr.
- Set SpecificationEpilog to #endif.
- Set ImplementationProlog to #ifdef \_DEBUG cr.

- Set ImplementationEpilog to #endif.

The following table lists whether leading and trailing linefeeds are generated.

Metaclass Trailing Linefeed Added? Class No Package Yes

*Default = Empty MultiLine)*

## In

The In property specifies how code is generated when the type is used with an argument that has the modifier In. The C value "const \$type\*" is the default.

## InOut

The InOut property specifies how code is generated when the type is used with an argument that has the modifier "InOut."

## IsLimited

The IsLimited property determines whether the class or record type is generated as limited.

*Default = Cleared*

## GenerateSerializationFunctions

Generates the serialization functions.

## LanguageMap

The LanguageMap property (specifies the Ada declaration for Rational Rhapsody language-independent types.

*Default = empty string*

## OpeningBraceStyle

The OpeningBraceStyle property controls where the opening brace of the code block is positioned - on the same line as the element name (SameLine) or on the following line (NewLine).

## Out

The Out property specifies how code is generated when the type is used with an argument that has the

modifier Out. The default value is \$type\*\*.

## PostDeclarationModifier

The PostDeclarationModifier property is used to enable Rational Rhapsody to reverse engineer non-standard keywords that appear in type declarations. Keywords that appear after the type name are stored as the value of this property, and the property is then used during code generation to re-create the original code.

Since this is a code generation property, it can also be used to add non-standard keywords to code even when reverse engineering is not used.

This property is used in conjunction with the PreDeclarationModifier and DeclarationModifier properties.

*Default = Blank*

## PreDeclarationModifier

The PreDeclarationModifier property is used to enable Rational Rhapsody to reverse engineer non-standard keywords that appear in type declarations. Keywords that appear before the type of the type (structure, enumeration, or union) are stored as the value of this property, and the property is then used during code generation to re-create the original code.

Since this is a code generation property, it can also be used to add non-standard keywords to code even when reverse engineering is not used.

This property is used in conjunction with the DeclarationModifier and PostDeclarationModifier properties.

*Default = Blank*

## PrivateName

*The PrivateName property specifies the pattern used to generate names of private operations in C. (Default = \$typeName)*

## PublicName

*The PublicName property specifies the pattern used to generate names of public operations in C. (Default = \$objectName\_\$typeName)*

## RefactorRenameRegularExpression

When you use the Refactor:Rename feature, Rhapsody searches the user-defined code in the model for all occurrences of the name of the specified element, including places where the element name appears as part of the name of elements whose name is derived from the element that is being renamed.

To locate these elements, Rhapsody uses a set of regular expressions.

The regular expressions used for this search are defined by the property `RefactorRenameRegularExpression`.

Different values are used for this property, depending on what type of element you are renaming. For example, you will see that this property has a different value under `Class` than it does under `Attribute`. For some types of elements, Rhapsody uses a more generic expression, taken from the value of the property under `ModelElement`.

In general, the value of this property is a regular expression that consists of a number of "Or" components. For the renaming to work properly, each such component must contain three groupings, with the middle grouping using the variable `$keyword`, which represents the name of the element that is being renamed. In some cases, the expression takes this information from the value of another property rather than using the `$keyword` variable.

```
Default = ([^A-Za-z0-9_]/^)($keyword)([^A-Za-z0-9_]/$)
/($<C.CG::Type::PublicName>[^A-Za-z0-9_]/$)
/([A-Za-z0-9_]($<C.CG::Type::PublicName>[^A-Za-z0-9_]/$)
/($<C.CG::Type::PrivateName>[^A-Za-z0-9_]/$)
/([A-Za-z0-9_]($<C.CG::Type::PrivateName>[^A-Za-z0-9_]/$)
```

## ReferenceImplementationPattern

*The ReferenceImplementationPattern property specifies how the Reference option for attribute/typedefs (composite types) is mapped to code. See the Rational Rhapsody help for information about composite types. (Default = "")*

## ReturnType

*The ReturnType property specifies how code is generated when the type is used as a return type. (Default = \$type\*)*

## Simplify

If you are using the Rational Rhapsody customizable code generation mechanism, the `Simplify` property can be used to change the way specific types of elements are handled by Rational Rhapsody when it transforms the model into a simplified model.

The property can take any of the following values:

- `None` - The element is ignored.
- `Copy` - The element is copied from the original to the simplified model. It is not modified in any way.
- `Default` - Uses the standard simplification for this item, as defined in Rational Rhapsody.
- `ByUser` - Uses the customized simplification provided by the user. (For details, see the section on User-Provided Simplification in the Rational Rhapsody help.)
- `ByUserPostDefault` - Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for the element has been applied.

*Default = "Default"*

## SpecificationEpilog

The SpecificationEpilog property allows you to add code to the end of the declaration of a model element.

For example, you can use this property to add an `#ifdef` to indicate that a type is available only when the code is compiled with `_DEBUG`, by setting the following properties:

- For SpecificationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For SpecificationEpilog, enter the value `#endif`
- For ImplementationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For ImplementationEpilog, enter the value `#endif`

*Default = Blank*

## SpecificationProlog

The SpecificationProlog property allows you to add code to the beginning of the declaration of a model element.

For example, you can use this property to add an `#ifdef` to indicate that a type is available only when the code is compiled with `_DEBUG`, by setting the following properties:

- For SpecificationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For SpecificationEpilog, enter the value `#endif`
- For ImplementationProlog, enter the value `#ifdef _DEBUG` and a new line.
- For ImplementationEpilog, enter the value `#endif`

*Default = Blank*

## StructAsTypedef

The StructAsTypedef property specifies whether the generated struct should be wrapped by a typedef. This property is applicable to structure types in C and C++.

*Default = Checked*

## TriggerArgument

The TriggerArgument property is used for mapping event and triggered operation arguments to code instead of the In, Out, and Out properties. A different property is required because of code generation limitations related to event arguments. You might also want to use the "Filter" facility in this window to refer to these definitions:

- In

- InOut
- Out

*(Default = \$type)*

## UnionAsTypedef

The UnionAsTypedef property specifies whether the generated union should be wrapped by a typedef. This property is applicable to union types in C and C++.

*Default = Checked*

## Visibility

The Visibility property specifies the visibility of the model element. Code generation maps the visibility specified for an element to the same visibility in the generated language. The possible values are as follows:

- Public - The model element is public.
- Protected - The model element is protected.
- Private - The element is private.

*(Default = Public)*

## VxWorks

The VxWorks metaclass contains the Environment settings (Compiler, framework libraries, and so on) for VxWorks compiler.

## AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*(Default = \$(OMROOT)/LangC/osconfig/VxWorks)*

## **AdditionalReservedWords**

The AdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use.

In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = empty string*

## **BSP**

The BSP property specifies the board support package (BSP) for the system. If you need to change the value of the CPU, you can simply reset the value in this property instead of changing the value in the MakeFileContent property.

*(Default = PENTIUM)*

## **BuildArgumentsInIDE**

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the C\_CG::[environment]::BuildCommandInIDE property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## **BuildCommandInIDE**

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration. To change this property, use the Configuration window in the browser - do not change it by using the Properties window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link.

The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*(Default = Debug)*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\vxmake.bat" vxbuild.mak build 5.5  
\"CPU=\$BSP\" \"BUILD=\$BuildCommandSet\" \" \"*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## CodeTestSettings

The CodeTestSettings property specifies the compiler command settings. This property supports



integration with Applied Microsystems Corporation CodeTest.

*(Default = CC = \$(AMC\_HOME)\bin\ctcc)*

## CompileSwitches

The CompileSwitches property specifies the compiler switches.

The default value is:

`-$OMDefaultSpecificationDirectory -I$(OMROOT)/LangC -I$(OMROOT)/LangC/oxf -DVxWorks $(INST_FLAGS) $(INCLUDE_PATH) $OMCPPCompileCommandSet -Wno-unused -c`

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The default value is:

`@echo Compiling $OMFileImpPath $(CREATE_OBJ_DIR) @$ (CC) $(CFLAGS) $OMFileCPPCompileSwitches -o $OMFileObjPath $OMFileImpPath`

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*(Default = -O0 -g)*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*(Default = empty string )*

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: `$OMFileObjPath : $OMFileImpPath "*.bmp" "*.ico" "*.rc2"`

The default value is \$OMFileObjPath : \$OMFileImpPath \$OMFileSpecPath \$OMFileDependencies.

## DuplicateLibsListInMakeFile

The DuplicateLibsListInMakeFile property is a Boolean value that specifies whether Rational Rhapsody should duplicate the libraries list in the generated makefile link command. This property supports linkers that are sensitive to library order in the link command.

*Default = Checked*

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction an IDE such as Eclipse, the EnableDebugIntegrationWithIDE property can be used to specify whether or not the IDE debugger should be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to True, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's exit() function is called automatically. The system function calls the destructors of framework instances in the order it determines. The EndApplicationCode property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an endApplication() function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to endApplication(), it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The EntryPoint property specifies the name of the main program for a given environment.

*(Default = vxmain)*

Use the "Filter" facility in this window to see the definition of the EntryPointDeclarationModifier property for more information.

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected

format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens has three parameters, each with an integer value:

- TotalNumberOfTokens - The number of tokens in the regular expression
- FileTokenPosition - The position of the file name token in the expression
- LineTokenPosition - The position of the line number token in the expression

*(Default = TotalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2)*

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the C\_CG::<Environment>::ExeName property plus the value of the ExeExtension property.

*Default = .out*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the C\_CG::<Environment>::ExeExtension property.

*(Default = Blank)*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements.

The file inclusions are generated in the makefile.

The default value is \$OMSpecIncludeInElements \$OMImpIncludeInElements.

## GetConnectedRuntimeLibraries

The `GetConnectedRuntimeLibraries` property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile.

Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

This default value is `$(OMROOT)/LangC/lib/vxWebComponents$(CPU)$(LIB_EXT)`, `$(OMROOT)/lib/vxWebServices$(CPU)$(LIB_EXT)`.

## HasIDEInterface

The `HasIDEInterface` property is a Boolean value that specifies whether IDE support is enabled.

If IDE support is enabled (Checked), the `IDEInterfaceDLL` property points to an IDE adapter that provides connection to the IDE. If the property is set to Cleared, IDE support is disabled and IDE services are not attempted. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = Checked*

## IDEInterfaceDLL

The `IDEInterfaceDLL` property is a string that points to the IDE adapter DLL. You should not have any reason to modify this property. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*(Default = \$OMROOT/DLLs/TornadoIDE.dll)*

## ImpExtension

The `ImpExtension` property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment. The default values are:

*(Default = .c)*

## Include

The `Include` property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*(Default = include)*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP gnu"
```

The default value is:

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vxmake.bat\" $makefile $maketarget"
```

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*(Default = .a)*

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

(Default = \$OMLinkCommandSet)

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the C\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

(Default = .mak)

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration. For example, the default content of the target type section of a C++ makefile for the Microsoft environment is: ##### Target type (Debug/Release)

```
#####  
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease  
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease BuildSet=$OMBUILDSET  
SUBSYSTEM=$OMSubSystem COM=$OMCOM RPFramewOrkDll=$OMRPFramewOrkDll  
ConfigurationCPPCompileSwitches= $OMReusableStatechartSwitches  
$OMConfigurationCPPCompileSwitches !IF "$RPFramewOrkDll" == "True"  
ConfigurationCPPCompileSwitches= $(ConfigurationCPPCompileSwitches) /D "FRAMEWORK_DLL"  
!ENDIF !IF "$(COM)" == "True" SUBSYSTEM=/SUBSYSTEM:windows !ENDIF
```

**Compilation Flags** The compilation flags section of the makefile contains the default compilation flags stored in the CompileSwitches property. For example, the default content of the compilation flags section of a C++ makefile for the Microsoft environment is: ##### Compilation flags

```
#####  
INCLUDE_QUALIFIER=/I LIB_PREFIX=MS
```

**Commands Definitions** The commands definition section of the makefile specifies programs to execute

from the makefile. For example, the default commands definition section of a C++ makefile for the Microsoft environment is: ##### Commands definition #####  
##### RMDIR = rmdir LIB\_CMD=link.exe -lib  
LINK\_CMD=link.exe LIB\_FLAGS=\$OMConfigurationLinkSwitches  
LINK\_FLAGS=\$OMConfigurationLinkSwitches \$(SUBSYSTEM) / MACHINE:I386

Generated Macros The generated macros section of the makefile contains a variable that expands to the Rational Rhapsody-generated macros in the makefile. For example: ##### Generated macros #####  
##### \$OMContextMacros  
OBJ\_DIR=\$OMObjectsDir !IF "\$(OBJ\_DIR)"!=" CREATE\_OBJ\_DIR=if not exist \$(OBJ\_DIR) mkdir  
\$(OBJ\_DIR) CLEAN\_OBJ\_DIR= if exist \$(OBJ\_DIR) \$(RMDIR) \$(OBJ\_DIR) !ELSE  
CREATE\_OBJ\_DIR= CLEAN\_OBJ\_DIR= !ENDIF

The \$OMContextMacros keyword expands several macros in the makefile. Each makefile macro has its own keyword. You can use these keywords separately to customize the makefile. The \$OMContextMacros variable modifies target-specific variables. Replace the \$OMContextMacros line in the MakeFileContent property with the following: FLAGSFILE=\$OMFlagsFile  
RULESFILE=\$OMRulesFile OMROOT=\$OMROOT C\_EXT=\$OMImplExt H\_EXT=\$OMSpecExt  
OBJ\_EXT=\$OMObjExt EXE\_EXT=\$OMExeExt LIB\_EXT=\$OMLibExt  
INSTRUMENTATION=\$OMInstrumentation TIME\_MODEL=\$OMTimeModel  
TARGET\_TYPE=\$OMTargetType TARGET\_NAME=\$OMTargetName \$OMAllDependencyRule  
TARGET\_MAIN=\$OMTargetMain LIBS=\$OMLibs INCLUDE\_PATH=\$OMIncludePath  
ADDITIONAL\_OBJS=\$OMAdditionalObjs OBJS= \$OMObjs

Predefined Macros The predefined macros section of the makefile contains other macros than the Rational Rhapsody-generated macros specified in the generated macros section. For example, the default predefined macros section of a C++ makefile for the Microsoft environment is: #####  
Predefined macros #####  
##### \$(OBJS) : \$(INST\_LIBS)  
\$(OXF\_LIBS) LIB\_POSTFIX= !IF "\$(BuildSet)"=="Release" LIB\_POSTFIX=R !ENDIF !IF  
"\$(TARGET\_TYPE)" == "Executable" LinkDebug=\$(LinkDebug) /DEBUG  
LinkRelease=\$(LinkRelease) /OPT:NOREF !ELSEIF "\$(TARGET\_TYPE)" == "Library"  
LinkDebug=\$(LinkDebug) /DEBUGTYPE:CV !ENDIF !IF "\$(INSTRUMENTATION)" == "Animation"  
INST\_FLAGS=/D "OMANIMATOR" INST\_INCLUDES=/I \$(OMROOT)\LangCpp\aoom /I  
\$(OMROOT)\LangCpp\tom !IF "\$(RPFrameWorkDll)" == "True" INST\_LIBS=  
OXF\_LIBS=\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)oxfanimdll\$(LIB\_POSTFIX) \$(LIB\_EXT)  
!ELSE INST\_LIBS= \$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)aomanim\$(LIB\_POSTFIX) \$(LIB\_EXT)  
OXF\_LIBS=\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)oxfinst\$(LIB\_POSTFIX) \$(LIB\_EXT)  
\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)omComAppl\$(LIB\_POSTFIX) \$(LIB\_EXT) !ENDIF  
SOCK\_LIB=wsock32.lib !ELSEIF "\$(INSTRUMENTATION)" == "Tracing" INST\_FLAGS=/D  
"OMTRACER" INST\_INCLUDES=/I \$(OMROOT)\LangCpp\aoom /I \$(OMROOT)\LangCpp\tom !IF  
"\$(RPFrameWorkDll)" == "True" INST\_LIBS= OXF\_LIBS=  
\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)oxfracedll\$(LIB\_POSTFIX) \$(LIB\_EXT) !ELSE  
INST\_LIBS=\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)tomtrace\$(LIB\_POSTFIX) \$(LIB\_EXT)  
\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)aomtrace\$(LIB\_POSTFIX) \$(LIB\_EXT) OXF\_LIBS=  
\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)oxfinst\$(LIB\_POSTFIX) \$(LIB\_EXT)  
\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)omComAppl\$(LIB\_POSTFIX) \$(LIB\_EXT) !ENDIF  
SOCK\_LIB=wsock32.lib !ELSEIF "\$(INSTRUMENTATION)" == "None" INST\_FLAGS=  
INST\_INCLUDES= INST\_LIBS= !IF "\$(RPFrameWorkDll)" == "True"  
OXF\_LIBS=\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)oxfdll\$(LIB\_POSTFIX) \$(LIB\_EXT) !ELSE  
OXF\_LIBS=\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)oxf\$(LIB\_POSTFIX) \$(LIB\_EXT) !ENDIF  
SOCK\_LIB= !ELSE !ERROR An invalid Instrumentation \$(INSTRUMENTATION) is specified.  
!ENDIF

Generated Dependencies The generated dependencies section of the makefile contains a variable that expands to Rational Rhapsody-generated dependencies and compilation instructions. For example, the generated dependencies section of a C++ makefile for the Microsoft environment is: #####  
Generated dependencies #####  
##### SOMContextDependencies  
\$OMFileObjPath : \$OMMainImplementationFile \$(OBJS) \$(CPP) \$(ConfigurationCPPCompileSwitches)  
/Fo"\$OMFileObjPath" \$OMMainImplementationFile

Linking Instructions The linking instructions section of the makefile contains the predefined linking instructions. For example, the default linking instructions section of a C++ makefile for the Microsoft environment is: ##### Linking instructions #####  
#####  
\$(TARGET\_NAME)\$(EXE\_EXT): \$(OBJS) \$(ADDITIONAL\_OBJS) \$OMFileObjPath  
\$OMMakefileName \$OMModelLibs @echo Linking \$(TARGET\_NAME)\$(EXE\_EXT) \$(LINK\_CMD)  
\$OMFileObjPath \$(OBJS) \$(ADDITIONAL\_OBJS) \ \$(LIBS) \ \$(INST\_LIBS) \ \$(OXF\_LIBS) \  
\$(SOCK\_LIB) \ \$(LINK\_FLAGS) /out:\$(TARGET\_NAME)\$(EXE\_EXT)  
\$(TARGET\_NAME)\$(LIB\_EXT) : \$(OBJS) \$(ADDITIONAL\_OBJS) \$OMMakefileName @echo  
Building library \$@ \$(LIB\_CMD) \$(LIB\_FLAGS) /out:\$(TARGET\_NAME)\$(LIB\_EXT) \$(OBJS)  
\$(ADDITIONAL\_OBJS) clean: @echo Cleanup \$OMCleanOBJS if exist \$OMFileObjPath erase  
\$OMFileObjPath if exist \*\$(OBJ\_EXT) erase \*\$(OBJ\_EXT) if exist \$(TARGET\_NAME).pdb erase  
\$(TARGET\_NAME).pdb if exist \$(TARGET\_NAME)\$(LIB\_EXT) erase  
\$(TARGET\_NAME)\$(LIB\_EXT) if exist \$(TARGET\_NAME).ilk erase \$(TARGET\_NAME).ilk if exist  
\$(TARGET\_NAME)\$(EXE\_EXT) erase \$(TARGET\_NAME)\$(EXE\_EXT) \$(CLEAN\_OBJ\_DIR)

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the C\_CG::<Environment>::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## ObjCleanCommand

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*(Default = \$(RM) \$OMFileObjPath)*



## ObjectName

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = empty string*

## ObjectsDirectory

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = empty string*

## ObjExtension

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*(Default = .o)*

## OSFileSystemCaseSensitive

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Cleared*

## ParseErrorDescript

The ParseErrorDescript property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = ([^:]+):([0-9]+):(.\*)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning)

Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*(Default = ([^:]+):?([0-9]+):?)*

## ParseMakeError

The ParseMakeError property is used to define a regular expression that represents the format of make process or linker error messages. This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = (make)(.\*)(Error)*

## ParseSeverityError

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^:]+):?([0-9]+):?*

## ParseSeverityWarning

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^:]+):?([0-9]+):? (warning)*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation.

*(Default = /)*

## PathWhiteSpaceHandling

For different operating systems, there are different methods for handling spaces in file paths, for example, enclosing the entire path in quotation marks. The PathWhiteSpaceHandling property allows you to specify the method that should be used for a given environment. The possible values are:

- NoHandling - the path should be left as is, with no special handling for spaces
- SurroundingQuotes - the entire path should be enclosed in quotation marks
- BackslashBeforeSpace - spaces in paths should be preceded by backslashes, as is the practice in VxWorks platforms

*Default = BackslashBeforeSpace*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*(Default = Checked )*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running the product, whereas the target is the machine running the application. To run remotely, the UseRemoteHost property must be set to True.

If UseRemoteHost is True and RemoteHost is blank, the current host name is used for the remote host. You can use this as a workaround if you have problems running animated applications on Windows 95. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = empty string*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*(Default = .h)*

## UpdateBuildSettingsInIDE

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## UseNewBuildOutputWindow

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the CG::General::ShowLogViewAfterBuild property to True.

*Default = Checked*

## UseNonZeroStdInputHandle

The UseNonZeroStdInputHandle property is a Boolean value that specifies whether to use a non-zero standard input handle.

*Default = Checked*

## UseRemoteHost

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Checked*

## VxWorks6diab

The VxWorks6diab metaclass contains the Environment settings (Compiler, framework libraries, and so on) for VxWorks6diab compiler.

## AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*(Default = \$(OMROOT)/LangC/osconfig/VxWorks)*

## AdditionalReservedWords

The AdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use.

In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = empty string*

## **BSP**

The BSP property specifies the board support package (BSP) for the system. If you need to change the value of the CPU, you can simply reset the value in this property instead of changing the value in the MakeFileContent property.

*(Default = PENTIUM)*

## **BuildArgumentsInIDE**

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the C\_CG::[environment]::BuildCommandInIDE property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## **BuildCommandInIDE**

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## **BuildCommandSet**

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration. To change this property, use the Configuration window in the browser - do not change it by using the Properties window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link.

The possible values are:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*(Default = Debug)*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\$OMROOT/etc/Executer.exe" "\""\$OMROOT\etc\vx6make.bat" vxbuild.mak buildLibs 6.5 \*  
*\\"CPU=\$BSP\" \\"TOOL=diab\" \\"TOOL\_FAMILY=diab\" \\"BUILD=\$BuildCommandSet\" \\" "*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## CodeTestSettings

The CodeTestSettings property specifies the compiler command settings. This property supports integration with Applied Microsystems Corporation CodeTest.

*(Default = CC = \$(AMC\_HOME)\bin\ctcc)*

## CompileSwitches

The CompileSwitches property specifies the compiler switches.

The default value is:

```
$IgnoreSwitches -I$OMDefaultSpecificationDirectory -I$(OMROOT)/LangC -I$(OMROOT)/LangC/oxf
-DVxWorks $(INST_FLAGS) $(INCLUDE_PATH) $OMCPPCompileCommandSet -Wno-unused -c
```

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The default value is:

```
@echo Compiling $OMFileImpPath $(CREATE_OBJ_DIR) @$$(CC) $$(CFLAGS)
$OMFileCPPCompileSwitches -o $OMFileObjPath $OMFileImpPath
```

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*(Default = -O -g)*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*(Default = empty string )*

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: `$OMFileObjPath : $OMFileImpPath "*.bmp" "*.ico" "*.rc2"`

The default value is `$OMFileObjPath : $OMFileImpPath $OMFileSpecPath $OMFileDependencies`.

## DuplicateLibsListInMakeFile

The DuplicateLibsListInMakeFile property is a Boolean value that specifies whether Rational Rhapsody should duplicate the libraries list in the generated makefile link command. This property supports linkers that are sensitive to library order in the link command.

*Default = Checked*

## **EnableDebugIntegrationWithIDE**

When you use Rational Rhapsody in conjunction an IDE such as Eclipse, the EnableDebugIntegrationWithIDE property can be used to specify whether or not the IDE debugger should be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to True, the IDE debugger is used.

*Default = Cleared*

## **EntryPoint**

The EntryPoint property specifies the name of the main program for a given environment.

*(Default = vxmain)*

Use the "Filter" facility in this window to see the definition of the EntryPointDeclarationModifier property for more information.

## **ErrorMessageTokensFormat**

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens has three parameters, each with an integer value:

- TotalNumberOfTokens - The number of tokens in the regular expression
- FileTokenPosition - The position of the file name token in the expression
- LineTokenPosition - The position of the line number token in the expression

*(Default = TotalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2)*

## **ExeExtension**

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the C\_CG::<Environment>::ExeName property plus the value of the ExeExtension property.

*Default = .out*

## **ExeName**



By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the C\_CG::<Environment>::ExeExtension property.

*(Default = Blank)*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements.

The file inclusions are generated in the makefile.

The default value is \$OMSpecIncludeInElements \$OMImpIncludeInElements.

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile.

Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

This default value is \$(OMROOT)/LangC/lib/vxWebComponents\$(CPU)\$(TOOL)\$(LIB\_EXT),  
\$(OMROOT)/lib/vxWebServices\$(CPU)\$(TOOL)\$(LIB\_EXT).

## HasIDEInterface

The HasIDEInterface property is a Boolean value that specifies whether IDE support is enabled.

If IDE support is enabled (Checked), the IDEInterfaceDLL property points to an IDE adapter that provides connection to the IDE. If the property is set to Cleared, IDE support is disabled and IDE services are not attempted. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = Checked*

## IDEInterfaceDLL

The IDEInterfaceDLL property is a string that points to the IDE adapter DLL. You should not have any reason to modify this property. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*(Default = \$OMROOT/DLLs/WorkbenchDebuggerIDE.dll)*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment. The default values are:

*(Default = .c)*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*(Default = include)*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

The default value is:

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget"
```

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*(Default = .a)*

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

*(Default = \$OMLinkCommandSet)*

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the C\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*(Default = .mak)*

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros

- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration. For example, the default content of the target type section of a C++ makefile for the Microsoft environment is: ##### Target type (Debug/Release)

```
#####  
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease  
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease BuildSet=$OMBuildSet  
SUBSYSTEM=$OMSubSystem COM=$OMCOM RPFramewOrkDll=$OMRPFramewOrkDll  
ConfigurationCPPCompileSwitches= $OMReusableStatechartSwitches  
$OMConfigurationCPPCompileSwitches !IF "$ (RPFramewOrkDll)" == "True"  
ConfigurationCPPCompileSwitches= $(ConfigurationCPPCompileSwitches) /D "FRAMEWORK_DLL"  
!ENDIF !IF "$ (COM)" == "True" SUBSYSTEM=/SUBSYSTEM:windows !ENDIF
```

**Compilation Flags** The compilation flags section of the makefile contains the default compilation flags stored in the CompileSwitches property. For example, the default content of the compilation flags section of a C++ makefile for the Microsoft environment is: ##### Compilation flags

```
#####  
INCLUDE_QUALIFIER=/I LIB_PREFIX=MS
```

**Commands Definitions** The commands definition section of the makefile specifies programs to execute from the makefile. For example, the default commands definition section of a C++ makefile for the Microsoft environment is: ##### Commands definition #####

```
##### RMDIR = rmdir LIB_CMD=link.exe -lib  
LINK_CMD=link.exe LIB_FLAGS=$OMConfigurationLinkSwitches  
LINK_FLAGS=$OMConfigurationLinkSwitches $(SUBSYSTEM) / MACHINE:I386
```

**Generated Macros** The generated macros section of the makefile contains a variable that expands to the Rational Rhapsody-generated macros in the makefile. For example: ##### Generated macros

```
##### $OMContextMacros  
OBJ_DIR=$OMObjectsDir !IF "$ (OBJ_DIR)" != "" CREATE_OBJ_DIR=if not exist $(OBJ_DIR) mkdir  
$(OBJ_DIR) CLEAN_OBJ_DIR= if exist $(OBJ_DIR) $(RMDIR) $(OBJ_DIR) !ELSE  
CREATE_OBJ_DIR= CLEAN_OBJ_DIR= !ENDIF
```

The \$OMContextMacros keyword expands several macros in the makefile. Each makefile macro has its own keyword. You can use these keywords separately to customize the makefile. The

\$OMContextMacros variable modifies target-specific variables. Replace the \$OMContextMacros line in the MakeFileContent property with the following: FLAGSFILE=\$OMFlagsFile

```
RULESFILE=$OMRulesFile OMROOT=$OMROOT C_EXT=$OMImplExt H_EXT=$OMSpecExt  
OBJ_EXT=$OMObjExt EXE_EXT=$OMExeExt LIB_EXT=$OMLibExt  
INSTRUMENTATION=$OMInstrumentation TIME_MODEL=$OMTimeModel  
TARGET_TYPE=$OMTargetType TARGET_NAME=$OMTargetName $OMAllDependencyRule  
TARGET_MAIN=$OMTargetMain LIBS=$OMLibs INCLUDE_PATH=$OMIncludePath  
ADDITIONAL_OBJS=$OMAdditionalObjs OBJS= $OMObjs
```

**Predefined Macros** The predefined macros section of the makefile contains other macros than the Rational Rhapsody-generated macros specified in the generated macros section. For example, the default predefined macros section of a C++ makefile for the Microsoft environment is: #####

```
Predefined macros #####  
##### $(OBJS) : $(INST_LIBS)  
$(OXF_LIBS) LIB_POSTFIX= !IF "$ (BuildSet)" == "Release" LIB_POSTFIX=R !ENDIF !IF
```

```

"$(TARGET_TYPE)" == "Executable" LinkDebug=$(LinkDebug) /DEBUG
LinkRelease=$(LinkRelease) /OPT:NOREF !ELSEIF "$(TARGET_TYPE)" == "Library"
LinkDebug=$(LinkDebug) /DEBUGTYPE:CV !ENDIF !IF "$(INSTRUMENTATION)" == "Animation"
INST_FLAGS=/D "OMANIMATOR" INST_INCLUDES=/I $(OMROOT)\LangCpp\om /I
$(OMROOT)\LangCpp\tom !IF "$(RPFrameWorkDll)" == "True" INST_LIBS=
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfanimdll$(LIB_POSTFIX) $(LIB_EXT)
!ELSE INST_LIBS= $(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomanim$(LIB_POSTFIX) $(LIB_EXT)
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfinst$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComApp$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)" == "Tracing" INST_FLAGS=/D
"OMTRACER" INST_INCLUDES=/I $(OMROOT)\LangCpp\om /I $(OMROOT)\LangCpp\tom !IF
"$(RPFrameWorkDll)" == "True" INST_LIBS= OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfracedll$(LIB_POSTFIX) $(LIB_EXT) !ELSE
INST_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)tomtrace$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomtrace$(LIB_POSTFIX) $(LIB_EXT) OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfinst$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComApp$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)" == "None" INST_FLAGS=
INST_INCLUDES= INST_LIBS= !IF "$(RPFrameWorkDll)" == "True"
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfdll$(LIB_POSTFIX) $(LIB_EXT) !ELSE
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB= !ELSE !ERROR An invalid Instrumentation $(INSTRUMENTATION) is specified.
!ENDIF

```

Generated Dependencies The generated dependencies section of the makefile contains a variable that expands to Rational Rhapsody-generated dependencies and compilation instructions. For example, the generated dependencies section of a C++ makefile for the Microsoft environment is: #####  
Generated dependencies #####  
##### \$OMContextDependencies  
\$OMFileObjPath : \$OMMainImplementationFile \$(OBJS) \$(CPP) \$(ConfigurationCPPCompileSwitches)  
/Fo"\$OMFileObjPath" \$OMMainImplementationFile

Linking Instructions The linking instructions section of the makefile contains the predefined linking instructions. For example, the default linking instructions section of a C++ makefile for the Microsoft environment is: ##### Linking instructions #####  
#####  
\$(TARGET\_NAME)\$(EXE\_EXT): \$(OBJS) \$(ADDITIONAL\_OBJS) \$OMFileObjPath  
\$OMMakefileName \$OMModelLibs @echo Linking \$(TARGET\_NAME)\$(EXE\_EXT) \$(LINK\_CMD)  
\$OMFileObjPath \$(OBJS) \$(ADDITIONAL\_OBJS) \ \$(LIBS) \ \$(INST\_LIBS) \ \$(OXF\_LIBS) \  
\$(SOCK\_LIB) \ \$(LINK\_FLAGS) /out:\$(TARGET\_NAME)\$(EXE\_EXT)  
\$(TARGET\_NAME)\$(LIB\_EXT) : \$(OBJS) \$(ADDITIONAL\_OBJS) \$OMMakefileName @echo  
Building library \$@ \$(LIB\_CMD) \$(LIB\_FLAGS) /out:\$(TARGET\_NAME)\$(LIB\_EXT) \$(OBJS)  
\$(ADDITIONAL\_OBJS) clean: @echo Cleanup \$OMCleanOBJS if exist \$OMFileObjPath erase  
\$OMFileObjPath if exist \*\$(OBJ\_EXT) erase \*\$(OBJ\_EXT) if exist \$(TARGET\_NAME).pdb erase  
\$(TARGET\_NAME).pdb if exist \$(TARGET\_NAME)\$(LIB\_EXT) erase  
\$(TARGET\_NAME)\$(LIB\_EXT) if exist \$(TARGET\_NAME).ilk erase \$(TARGET\_NAME).ilk if exist  
\$(TARGET\_NAME)\$(EXE\_EXT) erase \$(TARGET\_NAME)\$(EXE\_EXT) \$(CLEAN\_OBJ\_DIR)

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the C\_CG::<Environment>::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## **NullValue**

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## **ObjCleanCommand**

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*(Default = \$(RM) \$OMFileObjPath)*

## **ObjectName**

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = empty string*

## **ObjectsDirectory**

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = empty string*

## **ObjExtension**

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*(Default = .o)*

## **OSFileSystemCaseSensitive**

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Cleared*

## OSVersion

Used for the various Wind River environments, the OSVersion property represents the version number of the product. The value of the property is used in the batch files that build applications and framework libraries for these environments. In these batch files, the value of the property is provided as a parameter for both the "wrenv" command and the "make" command.

*Default = 6.8*

## ParseErrorDescript

The ParseErrorDescript property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = [""]([^\:]+)[""][,][ ]line ([0-9]+)[\:] (.\*)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning)

Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*(Default = [""]([^\:]+)[""][,][ ]line ([0-9]+)[\:] )*

## ParseMakeError

The ParseMakeError property is used to define a regular expression that represents the format of make process or linker error messages. This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = (.\*)(make)(.)(Error)*

## ParseSeverityError

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = [""]([^\:]+)[""][,][ ]line ([0-9]+)[\:]*

## ParseSeverityWarning

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ["](^[^:]+)["],[/]line ([0-9]+)[:] (warning)*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation.

*(Default = /)*

## PathWhiteSpaceHandling

For different operating systems, there are different methods for handling spaces in file paths, for example, enclosing the entire path in quotation marks. The PathWhiteSpaceHandling property allows you to specify the method that should be used for a given environment. The possible values are:

- NoHandling - the path should be left as is, with no special handling for spaces
- SurroundingQuotes - the entire path should be enclosed in quotation marks
- BackslashBeforeSpace - spaces in paths should be preceded by backslashes, as is the practice in VxWorks platforms

*Default = BackslashBeforeSpace*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*(Default = Checked )*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running the product, whereas the target is the machine running the application.

To run remotely, the UseRemoteHost property must be set to True. If UseRemoteHost is True and RemoteHost is blank, the current host name is used for the remote host. You can use this as a workaround if you have problems running animated applications on Windows 95. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = empty string*



## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*(Default = .h)*

## UpdateBuildSettingsInIDE

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## UseNewBuildOutputWindow

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the CG::General::ShowLogViewAfterBuild property to True.

*Default = Checked*

## UseNonZeroStdInputHandle

The UseNonZeroStdInputHandle property is a Boolean value that specifies whether to use a non-zero standard input handle.

*Default = Checked*

## UseRemoteHost

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Checked*

## VxWorks6diab\_RTP

The VxWorks6diab\_RTP metaclass contains the Environment settings (Compiler, framework libraries, and so on) for VxWorks6diab compiler.

### AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*(Default = \$(OMROOT)/LangC/osconfig/VxWorks)*

### AdditionalReservedWords

The AdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use.

In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = empty string*

### BSP

The BSP property specifies the board support package (BSP) for the system. If you need to change the value of the CPU, you can simply reset the value in this property instead of changing the value in the MakeFileContent property.

*(Default = PENTIUM)*

## **BuildArgumentsInIDE**

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the C\_CG::[environment]::BuildCommandInIDE property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## **BuildCommandInIDE**

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## **BuildCommandSet**

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration. To change this property, use the Configuration window in the browser - do not change it by using the Properties window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link.

The possible values are:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*(Default = Debug)*

## **buildFrameworkCommand**

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

```
Default = "$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat" vxbuild.mak buildLibs 6.5  
\"CPU=$BSP\" \"TOOL=diab\" \"TOOL_FAMILY=diab\" \"BUILD=$BuildCommandSet\"  
\"DISTRIBUTED=TRUE\" \" \"
```

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## CodeTestSettings

The CodeTestSettings property specifies the compiler command settings. This property supports integration with Applied Microsystems Corporation CodeTest.

*(Default = CC = \$(AMC\_HOME)\bin\ctcc)*

## CompileSwitches

The CompileSwitches property specifies the compiler switches.

The default value is:

```
-I$OMDefaultSpecificationDirectory -I$(OMROOT)/LangC -I$(OMROOT)/LangC/oxf -DVxWorks  
$(INST_FLAGS) $(INCLUDE_PATH) $OMCPPCompileCommandSet -Wno-unused -c
```

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The default value is:

```
@echo Compiling $OMFileImpPath $(CREATE_OBJ_DIR) @$$(CC) $(CFLAGS)  
$OMFileCPPCompileSwitches -o $OMFileObjPath $OMFileImpPath
```

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*(Default = -g)*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*(Default = empty string )*

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: \$OMFileObjPath : \$OMFileImpPath "\*.bmp" "\*.ico" "\*.rc2"

The default value is \$OMFileObjPath : \$OMFileImpPath \$OMFileSpecPath \$OMFileDependencies.

## DuplicateLibsListInMakeFile

The DuplicateLibsListInMakeFile property is a Boolean value that specifies whether Rational Rhapsody should duplicate the libraries list in the generated makefile link command. This property supports linkers that are sensitive to library order in the link command.

*Default = Checked*

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction an IDE such as Eclipse, the EnableDebugIntegrationWithIDE property can be used to specify whether or not the IDE debugger should be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to True, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's `exit()` function is called automatically. The system function calls the destructors of framework instances in the order it determines. The `EndApplicationCode` property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an `endApplication()` function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to `endApplication()`, it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The `EntryPoint` property specifies the name of the main program for a given environment.

*(Default = main)*

Use the "Filter" facility in this window to see the definition of the `EntryPointDeclarationModifier` property for more information.

## ErrorMessageTokensFormat

The `ErrorMessageTokensFormat`, working with the `ParseErrorMessage` property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. `ErrorMessageTokensFormat` defines the number and location of tokens within the regular expression defined by the `ParseErrorMessage` property. `ErrorMessageTokens` has three parameters, each with an integer value:

- `TotalNumberOfTokens` - The number of tokens in the regular expression
- `FileTokenPosition` - The position of the file name token in the expression
- `LineTokenPosition` - The position of the line number token in the expression

*(Default = TotalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2)*

## ExeExtension

The `ExeExtension` property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the `C_CG::<Environment>::ExeName` property plus the value of the `ExeExtension` property.

*Default = .vxe*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the C\_CG::<Environment>::ExeExtension property.

*(Default = Blank)*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements.

The file inclusions are generated in the makefile.

The default value is \$OMSpecIncludeInElements \$OMImpIncludeInElements.

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile.

Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

This default value is

\$(OMROOT)/LangC/lib/vx\$(DIST\_PREFIX)WebComponents\$(CPU)\$(RTP\_SUFFIX)\$(TOOL)\$(RHP\_LIB\_EXT),  
\$(OMROOT)/lib/vxWebServices\$(CPU)\$(RTP\_SUFFIX)\$(TOOL)\$(RHP\_LIB\_EXT).

## HasIDEInterface

The HasIDEInterface property is a Boolean value that specifies whether IDE support is enabled.

If IDE support is enabled (Checked), the IDEInterfaceDLL property points to an IDE adapter that provides connection to the IDE. If the property is set to Cleared, IDE support is disabled and IDE services are not attempted. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = Cleared*

## IDEInterfaceDLL

The IDEInterfaceDLL property is a string that points to the IDE adapter DLL. You should not have any reason to modify this property. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = empty string*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment. The default values are:

*(Default = .c)*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*(Default = include)*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is:

```
"rhapsody_dir>\share\etc\Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

The default value is:

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget 6.2 $BSP diab"
```

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.



*Default = Cleared*

## **LibExtension**

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*(Default = .a)*

## **LinkDebug**

The LinkDebug property specifies the special link switches used to link in debug mode.

## **LinkRelease**

The LinkRelease property specifies the special link switches used to link in release mode.

## **LinkSwitches**

The LinkSwitches property specifies the standard link switches used to link in any mode.

*(Default = \$OMLinkCommandSet)*

## **MakeExtension**

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the C\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*(Default = .mak)*

## **MakeFileContent**

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros

- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration. For example, the default content of the target type section of a C++ makefile for the Microsoft environment is: ##### Target type (Debug/Release)

```
#####
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease BuildSet=$OMBuildSet
SUBSYSTEM=$OMSubSystem COM=$OMCOM RPFrameWorkDll=$OMRPFrameWorkDll
ConfigurationCPPCompileSwitches= $OMReusableStatechartSwitches
$OMConfigurationCPPCompileSwitches !IF "$ (RPFrameWorkDll)" == "True"
ConfigurationCPPCompileSwitches= $(ConfigurationCPPCompileSwitches) /D "FRAMEWORK_DLL"
!ENDIF !IF "$ (COM)" == "True" SUBSYSTEM=/SUBSYSTEM:windows !ENDIF
```

**Compilation Flags** The compilation flags section of the makefile contains the default compilation flags stored in the CompileSwitches property. For example, the default content of the compilation flags section of a C++ makefile for the Microsoft environment is: ##### Compilation flags

```
#####
INCLUDE_QUALIFIER=/I LIB_PREFIX=MS
```

**Commands Definitions** The commands definition section of the makefile specifies programs to execute from the makefile. For example, the default commands definition section of a C++ makefile for the Microsoft environment is: ##### Commands definition #####

```
##### RMDIR = rmdir LIB_CMD=link.exe -lib
LINK_CMD=link.exe LIB_FLAGS=$OMConfigurationLinkSwitches
LINK_FLAGS=$OMConfigurationLinkSwitches $(SUBSYSTEM) / MACHINE:I386
```

**Generated Macros** The generated macros section of the makefile contains a variable that expands to the Rational Rhapsody-generated macros in the makefile. For example: ##### Generated macros

```
##### $OMContextMacros
OBJ_DIR=$OMObjectsDir !IF "$ (OBJ_DIR)" != "" CREATE_OBJ_DIR=if not exist $(OBJ_DIR) mkdir
$(OBJ_DIR) CLEAN_OBJ_DIR= if exist $(OBJ_DIR) $(RMDIR) $(OBJ_DIR) !ELSE
CREATE_OBJ_DIR= CLEAN_OBJ_DIR= !ENDIF
```

The \$OMContextMacros keyword expands several macros in the makefile. Each makefile macro has its own keyword. You can use these keywords separately to customize the makefile. The \$OMContextMacros variable modifies target-specific variables. Replace the \$OMContextMacros line in the MakeFileContent property with the following: FLAGSFILE=\$OMFlagsFile

```
RULESFILE=$OMRulesFile OMROOT=$OMROOT C_EXT=$OMImplExt H_EXT=$OMSpecExt
OBJ_EXT=$OMObjExt EXE_EXT=$OMExeExt LIB_EXT=$OMLibExt
INSTRUMENTATION=$OMInstrumentation TIME_MODEL=$OMTimeModel
TARGET_TYPE=$OMTargetType TARGET_NAME=$OMTargetName $OMAllDependencyRule
TARGET_MAIN=$OMTargetMain LIBS=$OMLibs INCLUDE_PATH=$OMIncludePath
ADDITIONAL_OBJS=$OMAdditionalObjs OBJS= $OMObjs
```

**Predefined Macros** The predefined macros section of the makefile contains other macros than the Rational Rhapsody-generated macros specified in the generated macros section. For example, the default predefined macros section of a C++ makefile for the Microsoft environment is: #####

```
Predefined macros #####
```

```
##### $(OBJS) : $(INST_LIBS)
$(OXF_LIBS) LIB_POSTFIX= !IF "$$(BuildSet)"=="Release" LIB_POSTFIX=R !ENDIF !IF
"$(TARGET_TYPE)" == "Executable" LinkDebug=$(LinkDebug) /DEBUG
LinkRelease=$(LinkRelease) /OPT:NOREF !ELSEIF "$(TARGET_TYPE)" == "Library"
LinkDebug=$(LinkDebug) /DEBUGTYPE:CV !ENDIF !IF "$(INSTRUMENTATION)" == "Animation"
INST_FLAGS=/D "OMANIMATOR" INST_INCLUDES=/I $(OMROOT)\LangCpp\lom /I
$(OMROOT)\LangCpp\lom !IF "$(RPFrameWorkDll)" == "True" INST_LIBS=
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxanimdll$(LIB_POSTFIX) $(LIB_EXT)
!ELSE INST_LIBS= $(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomanim$(LIB_POSTFIX) $(LIB_EXT)
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfirst$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComApp$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)" == "Tracing" INST_FLAGS=/D
"OMTRACER" INST_INCLUDES=/I $(OMROOT)\LangCpp\lom /I $(OMROOT)\LangCpp\lom !IF
"$(RPFrameWorkDll)" == "True" INST_LIBS= OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfracedll$(LIB_POSTFIX) $(LIB_EXT) !ELSE
INST_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)tomtrace$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomtrace$(LIB_POSTFIX) $(LIB_EXT) OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfirst$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComApp$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)" == "None" INST_FLAGS=
INST_INCLUDES= INST_LIBS= !IF "$(RPFrameWorkDll)" == "True"
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfdll$(LIB_POSTFIX) $(LIB_EXT) !ELSE
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB= !ELSE !ERROR An invalid Instrumentation $(INSTRUMENTATION) is specified.
!ENDIF
```

Generated Dependencies The generated dependencies section of the makefile contains a variable that expands to Rational Rhapsody-generated dependencies and compilation instructions. For example, the generated dependencies section of a C++ makefile for the Microsoft environment is: #####  
Generated dependencies #####  
##### \$OMContextDependencies  
\$OMFileObjPath : \$OMMainImplementationFile \$(OBJS) \$(CPP) \$(ConfigurationCPPCompileSwitches)  
/Fo"\$OMFileObjPath" \$OMMainImplementationFile

Linking Instructions The linking instructions section of the makefile contains the predefined linking instructions. For example, the default linking instructions section of a C++ makefile for the Microsoft environment is: ##### Linking instructions #####  
#####  
\$(TARGET\_NAME)\$(EXE\_EXT): \$(OBJS) \$(ADDITIONAL\_OBJS) \$OMFileObjPath  
\$OMMakefileName \$OMModelLibs @echo Linking \$(TARGET\_NAME)\$(EXE\_EXT) \$(LINK\_CMD)  
\$OMFileObjPath \$(OBJS) \$(ADDITIONAL\_OBJS) \ \$(LIBS) \ \$(INST\_LIBS) \ \$(OXF\_LIBS) \  
\$(SOCK\_LIB) \ \$(LINK\_FLAGS) /out:\$(TARGET\_NAME)\$(EXE\_EXT)  
\$(TARGET\_NAME)\$(LIB\_EXT) : \$(OBJS) \$(ADDITIONAL\_OBJS) \$OMMakefileName @echo  
Building library \$@ \$(LIB\_CMD) \$(LIB\_FLAGS) /out:\$(TARGET\_NAME)\$(LIB\_EXT) \$(OBJS)  
\$(ADDITIONAL\_OBJS) clean: @echo Cleanup \$OMCleanOBJS if exist \$OMFileObjPath erase  
\$OMFileObjPath if exist \*\$(OBJ\_EXT) erase \*\$(OBJ\_EXT) if exist \$(TARGET\_NAME).pdb erase  
\$(TARGET\_NAME).pdb if exist \$(TARGET\_NAME)\$(LIB\_EXT) erase  
\$(TARGET\_NAME)\$(LIB\_EXT) if exist \$(TARGET\_NAME).ilk erase \$(TARGET\_NAME).ilk if exist  
\$(TARGET\_NAME)\$(EXE\_EXT) erase \$(TARGET\_NAME)\$(EXE\_EXT) \$(CLEAN\_OBJ\_DIR)

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile

generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the C\_CG::<Environment>::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## **NullValue**

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## **ObjCleanCommand**

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*(Default = \$(RM) \$OMFileObjPath)*

## **ObjectName**

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = empty string*

## **ObjectsDirectory**

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = empty string*

## **ObjExtension**

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*(Default = .o)*

## **OSFileSystemCaseSensitive**

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Cleared*

## OSVersion

Used for the various Wind River environments, the OSVersion property represents the version number of the product. The value of the property is used in the batch files that build applications and framework libraries for these environments. In these batch files, the value of the property is provided as a parameter for both the "wrenv" command and the "make" command.

*Default = 6.8*

## ParseErrorDescript

The ParseErrorDescript property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = [""]([^\:]+)[""][,][ ]line ([0-9]+)[:] (.\*)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning)

Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*(Default = ([""]([^\:]+)[""][,][ ]line ([0-9]+)[:] )*

## ParseMakeError

The ParseMakeError property is used to define a regular expression that represents the format of make process or linker error messages. This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = (.\*)(make)(.\*)(Error)*

## ParseSeverityError

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should

be displayed alongside the message on the Build tab of the Output window.

*Default = [""]([^:]+)""[,][ ]line ([0-9]+)[:]*

## ParseSeverityWarning

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = [""]([^:]+)""[,][ ]line ([0-9]+)[:] (warning)*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation.

*(Default = / )*

## PathWhiteSpaceHandling

For different operating systems, there are different methods for handling spaces in file paths, for example, enclosing the entire path in quotation marks. The PathWhiteSpaceHandling property allows you to specify the method that should be used for a given environment. The possible values are:

- NoHandling - the path should be left as is, with no special handling for spaces
- SurroundingQuotes - the entire path should be enclosed in quotation marks
- BackslashBeforeSpace - spaces in paths should be preceded by backslashes, as is the practice in VxWorks platforms

*Default = BackslashBeforeSpace*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*(Default = Checked )*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running the product, whereas the target is the machine running the application. To run remotely, the UseRemoteHost property must be set to True.

If UseRemoteHost is True and RemoteHost is blank, the current host name is used for the remote host. You can use this as a workaround if you have problems running animated applications on Windows 95.

The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = empty string*

## **SpecExtension**

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*(Default = .h)*

## **UpdateBuildSettingsInIDE**

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## **UseNewBuildOutputWindow**

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the CG::General::ShowLogViewAfterBuild property to True.

*Default = Checked*

## **UseNonZeroStdInputHandle**

The UseNonZeroStdInputHandle property is a Boolean value that specifies whether to use a non-zero standard input handle.

*Default = Checked*

## **UseRemoteHost**

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Checked*

## VxWorks6gnu

The VxWorks6gnu metaclass contains the Environment settings (Compiler, framework libraries, and so on) for VxWorks6gnu compiler.

### AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*(Default = \$(OMROOT)/LangC/osconfig/VxWorks)*

### AdditionalReservedWords

The AdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use.

In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = empty string*

### BSP

The BSP property specifies the board support package (BSP) for the system. If you need to change the value of the CPU, you can simply reset the value in this property instead of changing the value in the



MakeFileContent property.

*(Default = PENTIUM)*

## BuildArgumentsInIDE

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the C\_CG::[environment]::BuildCommandInIDE property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandInIDE

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration. To change this property, use the Configuration window in the browser - do not change it by using the Properties window or by modifying the site.prp file.

Note that this property also affects the names of the framework libraries used in the link. The possible values are:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*(Default = Debug)*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\$OMROOT/etc/Executer.exe" "\""\$OMROOT\etc\vx6make.bat" vxbuild.mak buildLibs 6.5  
\"CPU=\$BSP\" \"TOOL=gnu\" \"TOOL\_FAMILY=gnu\" \"BUILD=\$BuildCommandSet\" \" \"*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## CodeTestSettings

The CodeTestSettings property specifies the compiler command settings. This property supports integration with Applied Microsystems Corporation CodeTest.

*(Default = CC = \$(AMC\_HOME)\bin\ctcc)*

## CompileSwitches

The CompileSwitches property specifies the compiler switches.

The default value is:

*-\$OMDefaultSpecificationDirectory -I\$(OMROOT)/LangC -I\$(OMROOT)/LangC/oxf -DVxWorks  
\$(INST\_FLAGS) \$(INCLUDE\_PATH) \$OMCPCPPCompileCommandSet -Wno-unused -c*

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The default value is:

*@echo Compiling \$OMFileImpPath \$(CREATE\_OBJ\_DIR) @\$ (CC) \$(CFLAGS)*

\$OMFileCPPCompileSwitches -o \$OMFileObjPath \$OMFileImpPath

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*(Default = -O0 -g)*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*(Default = empty string )*

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: \$OMFileObjPath : \$OMFileImpPath "\*.bmp" "\*.ico" "\*.rc2"

The default value is \$OMFileObjPath : \$OMFileImpPath \$OMFileSpecPath \$OMFileDependencies.

## DuplicateLibsListInMakeFile

The DuplicateLibsListInMakeFile property is a Boolean value that specifies whether Rational Rhapsody should duplicate the libraries list in the generated makefile link command. This property supports linkers that are sensitive to library order in the link command.

*Default = Checked*

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction an IDE such as Eclipse, the EnableDebugIntegrationWithIDE property can be used to specify whether or not the IDE debugger should be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to True, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's `exit()` function is called automatically. The system function calls the destructors of framework instances in the order it determines. The `EndApplicationCode` property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an `endApplication()` function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to `endApplication()`, it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The `EntryPoint` property specifies the name of the main program for a given environment.

*(Default = vxmain)*

Use the "Filter" facility in this window to see the definition of the `EntryPointDeclarationModifier` property for more information.

## ErrorMessageTokensFormat

The `ErrorMessageTokensFormat`, working with the `ParseErrorMessage` property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. `ErrorMessageTokensFormat` defines the number and location of tokens within the regular expression defined by the `ParseErrorMessage` property. `ErrorMessageTokens` has three parameters, each with an integer value:

- `TotalNumberOfTokens` - The number of tokens in the regular expression
- `FileTokenPosition` - The position of the file name token in the expression
- `LineTokenPosition` - The position of the line number token in the expression

*(Default = TotalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2)*

## ExeExtension

The `ExeExtension` property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the `C_CG::<Environment>::ExeName` property plus the value of the `ExeExtension` property.

*Default = .out*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the C\_CG::<Environment>::ExeExtension property.

*(Default = Blank)*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements.

The file inclusions are generated in the makefile.

The default value is \$OMSpecIncludeInElements \$OMImpIncludeInElements.

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile.

Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

This default value is \$(OMROOT)/LangC/lib/vxWebComponents\$(CPU)\$(LIB\_EXT),  
\$(OMROOT)/lib/vxWebServices\$(CPU)\$(LIB\_EXT).

## HasIDEInterface

The HasIDEInterface property is a Boolean value that specifies whether IDE support is enabled.

If IDE support is enabled (Checked), the IDEInterfaceDLL property points to an IDE adapter that provides connection to the IDE. If the property is set to Cleared, IDE support is disabled and IDE services are not attempted. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = Checked*

## IDEInterfaceDLL

The IDEInterfaceDLL property is a string that points to the IDE adapter DLL. You should not have any reason to modify this property. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*(Default = \$OMROOT/DLLs/WorkbenchDebuggerIDE.dll)*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment. The default values are:

*(Default = .c)*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*(Default = include)*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

The default value is:

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget"
```

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## **LibExtension**

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*(Default = .a)*

## **LinkDebug**

The LinkDebug property specifies the special link switches used to link in debug mode.

*(Default = -g)*

## **LinkRelease**

The LinkRelease property specifies the special link switches used to link in release mode.

## **LinkSwitches**

The LinkSwitches property specifies the standard link switches used to link in any mode.

*(Default = \$OMLinkCommandSet)*

## **MakeExtension**

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the C\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*(Default = .mak)*

## **MakeFileContent**

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags

- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration. For example, the default content of the target type section of a C++ makefile for the Microsoft environment is: ##### Target type (Debug/Release)

```
#####  
C++CompileDebug=$OMC++CompileDebug C++CompileRelease=$OMC++CompileRelease  
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease BuildSet=$OMBuildSet  
SUBSYSTEM=$OMSubSystem COM=$OMCOM RPFrameWorkDll=$OMRPFrameWorkDll  
ConfigurationC++CompileSwitches= $OMReusableStatechartSwitches  
$OMConfigurationC++CompileSwitches !IF "$ (RPFrameWorkDll)" == "True"  
ConfigurationC++CompileSwitches= $(ConfigurationC++CompileSwitches) /D "FRAMEWORK_DLL"  
!ENDIF !IF "$ (COM)" == "True" SUBSYSTEM=/SUBSYSTEM:windows !ENDIF
```

**Compilation Flags** The compilation flags section of the makefile contains the default compilation flags stored in the CompileSwitches property. For example, the default content of the compilation flags section of a C++ makefile for the Microsoft environment is: ##### Compilation flags

```
#####  
INCLUDE_QUALIFIER=/I LIB_PREFIX=MS
```

**Commands Definitions** The commands definition section of the makefile specifies programs to execute from the makefile. For example, the default commands definition section of a C++ makefile for the Microsoft environment is: ##### Commands definition #####

```
##### RMDIR = rmdir LIB_CMD=link.exe -lib  
LINK_CMD=link.exe LIB_FLAGS=$OMConfigurationLinkSwitches  
LINK_FLAGS=$OMConfigurationLinkSwitches $(SUBSYSTEM) / MACHINE:I386
```

**Generated Macros** The generated macros section of the makefile contains a variable that expands to the Rational Rhapsody-generated macros in the makefile. For example: ##### Generated macros

```
##### $OMContextMacros  
OBJ_DIR=$OMObjectsDir !IF "$ (OBJ_DIR)" != "" CREATE_OBJ_DIR=if not exist $(OBJ_DIR) mkdir  
$(OBJ_DIR) CLEAN_OBJ_DIR= if exist $(OBJ_DIR) $(RMDIR) $(OBJ_DIR) !ELSE  
CREATE_OBJ_DIR= CLEAN_OBJ_DIR= !ENDIF
```

The \$OMContextMacros keyword expands several macros in the makefile. Each makefile macro has its own keyword. You can use these keywords separately to customize the makefile. The \$OMContextMacros variable modifies target-specific variables. Replace the \$OMContextMacros line in the MakeFileContent property with the following: FLAGSFILE=\$OMFlagsFile

```
RULESFILE=$OMRulesFile OMROOT=$OMROOT C_EXT=$OMImplExt H_EXT=$OMSpecExt  
OBJ_EXT=$OMObjExt EXE_EXT=$OMExeExt LIB_EXT=$OMLibExt  
INSTRUMENTATION=$OMInstrumentation TIME_MODEL=$OMTimeModel  
TARGET_TYPE=$OMTargetType TARGET_NAME=$OMTargetName $OMAllDependencyRule  
TARGET_MAIN=$OMTargetMain LIBS=$OMLibs INCLUDE_PATH=$OMIncludePath  
ADDITIONAL_OBJS=$OMAdditionalObjs OBJS= $OMObjs
```

**Predefined Macros** The predefined macros section of the makefile contains other macros than the Rational



Rhapsody-generated macros specified in the generated macros section. For example, the default predefined macros section of a C++ makefile for the Microsoft environment is: #####

```

Predefined macros #####
##### $(OBJS) : $(INST_LIBS)
$(OXF_LIBS) LIB_POSTFIX= !IF "$ (BuildSet)"=="Release" LIB_POSTFIX=R !ENDIF !IF
"$(TARGET_TYPE)" == "Executable" LinkDebug=$(LinkDebug) /DEBUG
LinkRelease=$(LinkRelease) /OPT:NOREF !ELSEIF "$(TARGET_TYPE)" == "Library"
LinkDebug=$(LinkDebug) /DEBUGTYPE:CV !ENDIF !IF "$(INSTRUMENTATION)" == "Animation"
INST_FLAGS=/D "OMANIMATOR" INST_INCLUDES=/I $(OMROOT)\LangCpp\om /I
$(OMROOT)\LangCpp\om !IF "$(RPFrameWorkDll)" == "True" INST_LIBS=
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfanimdll$(LIB_POSTFIX) $(LIB_EXT)
!ELSE INST_LIBS= $(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomanim$(LIB_POSTFIX) $(LIB_EXT)
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfirst$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)" == "Tracing" INST_FLAGS=/D
"OMTRACER" INST_INCLUDES=/I $(OMROOT)\LangCpp\om /I $(OMROOT)\LangCpp\om !IF
"$(RPFrameWorkDll)" == "True" INST_LIBS= OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxftracedll$(LIB_POSTFIX) $(LIB_EXT) !ELSE
INST_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)tomtrace$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomtrace$(LIB_POSTFIX) $(LIB_EXT) OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfirst$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)" == "None" INST_FLAGS=
INST_INCLUDES= INST_LIBS= !IF "$(RPFrameWorkDll)" == "True"
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfdll$(LIB_POSTFIX) $(LIB_EXT) !ELSE
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB= !ELSE !ERROR An invalid Instrumentation $(INSTRUMENTATION) is specified.
!ENDIF

```

Generated Dependencies The generated dependencies section of the makefile contains a variable that expands to Rational Rhapsody-generated dependencies and compilation instructions. For example, the generated dependencies section of a C++ makefile for the Microsoft environment is: #####

```

Generated dependencies #####
##### $OMContextDependencies
$OMFileObjPath : $OMMainImplementationFile $(OBJS) $(CPP) $(ConfigurationCPPCompileSwitches)
/Fo"$OMFileObjPath" $OMMainImplementationFile

```

Linking Instructions The linking instructions section of the makefile contains the predefined linking instructions. For example, the default linking instructions section of a C++ makefile for the Microsoft environment is: #####

```

#####
$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS) $OMFileObjPath
$OMMakefileName $OMModelLibs @echo Linking $(TARGET_NAME)$(EXE_EXT) $(LINK_CMD)
$OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \ $(OXF_LIBS) \
$(SOCK_LIB) \ $(LINK_FLAGS) /out:$(TARGET_NAME)$(EXE_EXT)
$(TARGET_NAME)$(LIB_EXT) : $(OBJS) $(ADDITIONAL_OBJS) $OMMakefileName @echo
Building library $@ $(LIB_CMD) $(LIB_FLAGS) /out:$(TARGET_NAME)$(LIB_EXT) $(OBJS)
$(ADDITIONAL_OBJS) clean: @echo Cleanup $OMCleanOBJS if exist $OMFileObjPath erase
$OMFileObjPath if exist *$(OBJ_EXT) erase *$(OBJ_EXT) if exist $(TARGET_NAME).pdb erase
$(TARGET_NAME).pdb if exist $(TARGET_NAME)$(LIB_EXT) erase
$(TARGET_NAME)$(LIB_EXT) if exist $(TARGET_NAME).ilk erase $(TARGET_NAME).ilk if exist
$(TARGET_NAME)$(EXE_EXT) erase $(TARGET_NAME)$(EXE_EXT) $(CLEAN_OBJ_DIR)

```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the C\_CG::<Environment>::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## ObjCleanCommand

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*(Default = \$(RM) \$OMFileObjPath)*

## ObjectName

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = empty string*

## ObjectsDirectory

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = empty string*

## ObjExtension

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*(Default = .o)*

## OSFileSystemCaseSensitive

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Cleared*

## OSVersion

Used for the various Wind River environments, the OSVersion property represents the version number of the product. The value of the property is used in the batch files that build applications and framework libraries for these environments. In these batch files, the value of the property is provided as a parameter for both the "wrenv" command and the "make" command.

*Default = 6.8*

## ParseErrorDescript

The ParseErrorDescript property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = ([^:]+)[:]( [0-9]+)[:] (error|warning)[:] (.\*)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning)

Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*(Default = ([^:]+)[:]( [0-9]+)[:] )*

## ParseMakeError

The ParseMakeError property is used to define a regular expression that represents the format of make process or linker error messages. This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = (.\*)(make)(.\*)(Error)*

## ParseSeverityError

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^:]+):([0-9]+): (error)*

## ParseSeverityWarning

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^:]+):([0-9]+): (warning)*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation.

*(Default = /)*

## PathWhiteSpaceHandling

For different operating systems, there are different methods for handling spaces in file paths, for example, enclosing the entire path in quotation marks. The PathWhiteSpaceHandling property allows you to specify the method that should be used for a given environment. The possible values are:

- NoHandling - the path should be left as is, with no special handling for spaces
- SurroundingQuotes - the entire path should be enclosed in quotation marks
- BackslashBeforeSpace - spaces in paths should be preceded by backslashes, as is the practice in VxWorks platforms

*Default = BackslashBeforeSpace*

## ProcessorArchitecture

The ProcessorArchitecture property is used to specify whether the application is being developed for 32-bit or 64-bit target systems. The value of the property is used in a number of places, such as the command used for building the Rhapsody framework, the content of generated makefiles, and the names of generated libraries.

Verify that ProcessorArchitecture is set to the appropriate value for the target systems. The possible values are x86 (for 32-bit) and x64 (for 64-bit).

Once you have set the property to the appropriate value, rebuild the Rhapsody framework, and generate

and build your application.

*Default = x86*

## **QuoteOMROOT**

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*(Default = Checked )*

## **RemoteHost**

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running the product, whereas the target is the machine running the application.

To run remotely, the UseRemoteHost property must be set to True. If UseRemoteHost is True and RemoteHost is blank, the current host name is used for the remote host. You can use this as a workaround if you have problems running animated applications on Windows 95. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = empty string*

## **SpecExtension**

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*(Default = .h)*

## **UpdateBuildSettingsInIDE**

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## **UseNewBuildOutputWindow**

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the `CG::General::ShowLogViewAfterBuild` property to `True`.

*Default = Checked*

## UseNonZeroStdInputHandle

The `UseNonZeroStdInputHandle` property is a Boolean value that specifies whether to use a non-zero standard input handle.

*Default = Checked*

## UseRemoteHost

The `UseRemoteHost` property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Checked*

## VxWorks6gnu\_RTP

The `VxWorks6gnu_RTP` metaclass contains the Environment settings (Compiler, framework libraries, and so on) for the `VxWorks6gnu_RTP` compiler.

## AdaptorSearchPath

The `AdaptorSearchPath` property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the `AdaptorSearchPath` property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*(Default = \$(OMROOT)/LangC/osconfig/VxWorks)*

## AdditionalReservedWords

The AdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use.

In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = empty string*

## BSP

The BSP property specifies the board support package (BSP) for the system. If you need to change the value of the CPU, you can simply reset the value in this property instead of changing the value in the MakeFileContent property.

*(Default = PENTIUM)*

## BuildArgumentsInIDE

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the C\_CG::[environment]::BuildCommandInIDE property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandInIDE

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration. To change this property, use the Configuration window in the browser - do not change it by using the Properties window or by modifying the site.prp file.

Note that this property also affects the names of the framework libraries used in the link. The possible values are:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*(Default = Debug)*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

```
Default = "$OMROOT/etc/Executer.exe" "\""$OMROOT\etc\vx6make.bat" vxbuild.mak buildLibs 6.5  
\"CPU=$BSP\" \"TOOL=gnu\" \"TOOL_FAMILY=gnu\" \"BUILD=$BuildCommandSet\"  
\"DISTRIBUTED=TRUE\" \" \"
```

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Cleared*

## CodeTestSettings

The CodeTestSettings property specifies the compiler command settings. This property supports integration with Applied Microsystems Corporation CodeTest.



*(Default = CC = \$(AMC\_HOME)\bin\ctcc)*

## CompileSwitches

The CompileSwitches property specifies the compiler switches.

The default value is:

`-$OMDefaultSpecificationDirectory -I$(OMROOT)/LangC -I$(OMROOT)/LangC/oxf -DVxWorks  
$(INST_FLAGS) $(INCLUDE_PATH) $OMCPPCompileCommandSet -Wno-unused -c`

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The default value is:

`@echo Compiling $OMFileImpPath $(CREATE_OBJ_DIR) @$(CC) $(CFLAGS)  
$OMFileCPPCompileSwitches -o $OMFileObjPath $OMFileImpPath`

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*(Default = -O0 -g)*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*(Default = empty string )*

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: `$OMFileObjPath : $OMFileImpPath "*.bmp" "*.ico" "*.rc2"`

The default value is `$OMFileObjPath : $OMFileImpPath $OMFileSpecPath $OMFileDependencies`.

## DuplicateLibsListInMakeFile

The DuplicateLibsListInMakeFile property is a Boolean value that specifies whether Rational Rhapsody should duplicate the libraries list in the generated makefile link command. This property supports linkers that are sensitive to library order in the link command.

*Default = Checked*

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction an IDE such as Eclipse, the EnableDebugIntegrationWithIDE property can be used to specify whether or not the IDE debugger should be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to True, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's exit() function is called automatically. The system function calls the destructors of framework instances in the order it determines. The EndApplicationCode property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an endApplication() function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to endApplication(), it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The EntryPoint property specifies the name of the main program for a given environment.

*(Default = main)*

Use the "Filter" facility in this window to see the definition of the EntryPointDeclarationModifier property for more information.

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location

of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens has three parameters, each with an integer value:

- TotalNumberOfTokens - The number of tokens in the regular expression
- FileTokenPosition - The position of the file name token in the expression
- LineTokenPosition - The position of the line number token in the expression

*(Default = TotalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2)*

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the C\_CG::<Environment>::ExeName property plus the value of the ExeExtension property.

*Default = .vxe*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the C\_CG::<Environment>::ExeExtension property.

*(Default = Blank)*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements.

The file inclusions are generated in the makefile.

The default value is \$OMSpecIncludeInElements \$OMImpIncludeInElements.

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation,

these libraries are added to the generated makefile.

Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

This default value is

`$(OMROOT)/LangC/lib/vx$(DIST_PREFIX)WebComponents$(CPU)$(RTP_SUFFIX)$(TOOL)$(RHP_LIB_EXT),  
$(OMROOT)/lib/vxWebServices$(CPU)$(RTP_SUFFIX)$(TOOL)$(RHP_LIB_EXT).`

## HasIDEInterface

The HasIDEInterface property is a Boolean value that specifies whether IDE support is enabled.

If IDE support is enabled (Checked), the IDEInterfaceDLL property points to an IDE adapter that provides connection to the IDE. If the property is set to Cleared, IDE support is disabled and IDE services are not attempted. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = Cleared*

## IDEInterfaceDLL

The IDEInterfaceDLL property is a string that points to the IDE adapter DLL. You should not have any reason to modify this property. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = empty string*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment. The default values are:

*(Default = .c)*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*(Default = include)*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch

file are target-dependent. The command syntax is:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

The default value is:

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget"
```

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*(Default = .a)*

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

*(Default = -g)*

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

(Default = \$OMLinkCommandSet -MD -MP)

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the C\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

(Default = .mak)

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration. For example, the default content of the target type section of a C++ makefile for the Microsoft environment is: ##### Target type (Debug/Release)

```
#####  
C++CompileDebug=$OMC++CompileDebug C++CompileRelease=$OMC++CompileRelease  
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease BuildSet=$OMBldSet  
SUBSYSTEM=$OMSubSystem COM=$OMCOM RPFramewOrkDll=$OMRPFramewOrkDll  
ConfigurationC++CompileSwitches= $OMReusableStatechartSwitches  
$OMConfigurationC++CompileSwitches !IF "$RPFramewOrkDll" == "True"  
ConfigurationC++CompileSwitches= $(ConfigurationC++CompileSwitches) /D "FRAMEWORK_DLL"  
!ENDIF !IF "$(COM)" == "True" SUBSYSTEM=/SUBSYSTEM:windows !ENDIF
```

**Compilation Flags** The compilation flags section of the makefile contains the default compilation flags stored in the CompileSwitches property. For example, the default content of the compilation flags section of a C++ makefile for the Microsoft environment is: ##### Compilation flags

```
#####  
INCLUDE_QUALIFIER=/I LIB_PREFIX=MS
```

**Commands Definitions** The commands definition section of the makefile specifies programs to execute

from the makefile. For example, the default commands definition section of a C++ makefile for the Microsoft environment is: ##### Commands definition #####  
##### RMDIR = rmdir LIB\_CMD=link.exe -lib  
LINK\_CMD=link.exe LIB\_FLAGS=\$OMConfigurationLinkSwitches  
LINK\_FLAGS=\$OMConfigurationLinkSwitches \$(SUBSYSTEM) / MACHINE:I386

Generated Macros The generated macros section of the makefile contains a variable that expands to the Rational Rhapsody-generated macros in the makefile. For example: ##### Generated macros #####  
##### \$OMContextMacros  
OBJ\_DIR=\$OMObjectsDir !IF "\$ (OBJ\_DIR)" != "" CREATE\_OBJ\_DIR=if not exist \$(OBJ\_DIR) mkdir  
\$(OBJ\_DIR) CLEAN\_OBJ\_DIR= if exist \$(OBJ\_DIR) \$(RMDIR) \$(OBJ\_DIR) !ELSE  
CREATE\_OBJ\_DIR= CLEAN\_OBJ\_DIR= !ENDIF

The \$OMContextMacros keyword expands several macros in the makefile. Each makefile macro has its own keyword. You can use these keywords separately to customize the makefile. The \$OMContextMacros variable modifies target-specific variables.

Replace the \$OMContextMacros line in the MakeFileContent property with the following:  
FLAGSFILE=\$OMFlagsFile RULESFILE=\$OMRulesFile OMROOT=\$OMROOT  
C\_EXT=\$OMImplExt H\_EXT=\$OMSpecExt OBJ\_EXT=\$OMObjExt EXE\_EXT=\$OMExeExt  
LIB\_EXT=\$OMLibExt INSTRUMENTATION=\$OMInstrumentation TIME\_MODEL=\$OMTimeModel  
TARGET\_TYPE=\$OMTargetType TARGET\_NAME=\$OMTargetName \$OMAllDependencyRule  
TARGET\_MAIN=\$OMTargetMain LIBS=\$OMLibs INCLUDE\_PATH=\$OMIncludePath  
ADDITIONAL\_OBJS=\$OMAdditionalObjs OBJS= \$OMObjs

Predefined Macros The predefined macros section of the makefile contains other macros than the Rational Rhapsody-generated macros specified in the generated macros section. For example, the default predefined macros section of a C++ makefile for the Microsoft environment is: #####  
Predefined macros #####

##### \$(OBJS) : \$(INST\_LIBS)  
\$(OXF\_LIBS) LIB\_POSTFIX= !IF "\$ (BuildSet)" == "Release" LIB\_POSTFIX=R !ENDIF !IF  
"\$ (TARGET\_TYPE)" == "Executable" LinkDebug=\$(LinkDebug) /DEBUG  
LinkRelease=\$(LinkRelease) /OPT:NOREF !ELSEIF "\$ (TARGET\_TYPE)" == "Library"  
LinkDebug=\$(LinkDebug) /DEBUGTYPE:CV !ENDIF !IF "\$ (INSTRUMENTATION)" == "Animation"  
INST\_FLAGS=/D "OMANIMATOR" INST\_INCLUDES=/I \$(OMROOT)\LangCpp\aoom /I  
\$(OMROOT)\LangCpp\tom !IF "\$ (RPFrameWorkDll)" == "True" INST\_LIBS=  
OXF\_LIBS=\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)oxanimdll\$(LIB\_POSTFIX) \$(LIB\_EXT)  
!ELSE INST\_LIBS= \$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)aomanim\$(LIB\_POSTFIX) \$(LIB\_EXT)  
OXF\_LIBS=\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)oxfirst\$(LIB\_POSTFIX) \$(LIB\_EXT)  
\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)omComAppl\$(LIB\_POSTFIX) \$(LIB\_EXT) !ENDIF  
SOCK\_LIB=wsock32.lib !ELSEIF "\$ (INSTRUMENTATION)" == "Tracing" INST\_FLAGS=/D  
"OMTRACER" INST\_INCLUDES=/I \$(OMROOT)\LangCpp\aoom /I \$(OMROOT)\LangCpp\tom !IF  
"\$ (RPFrameWorkDll)" == "True" INST\_LIBS= OXF\_LIBS=  
\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)oxfracedll\$(LIB\_POSTFIX) \$(LIB\_EXT) !ELSE  
INST\_LIBS=\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)tomtrace\$(LIB\_POSTFIX) \$(LIB\_EXT)  
\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)aomtrace\$(LIB\_POSTFIX) \$(LIB\_EXT) OXF\_LIBS=  
\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)oxfirst\$(LIB\_POSTFIX) \$(LIB\_EXT)  
\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)omComAppl\$(LIB\_POSTFIX) \$(LIB\_EXT) !ENDIF  
SOCK\_LIB=wsock32.lib !ELSEIF "\$ (INSTRUMENTATION)" == "None" INST\_FLAGS=  
INST\_INCLUDES= INST\_LIBS= !IF "\$ (RPFrameWorkDll)" == "True"  
OXF\_LIBS=\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)oxfdll\$(LIB\_POSTFIX) \$(LIB\_EXT) !ELSE  
OXF\_LIBS=\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)oxf\$(LIB\_POSTFIX) \$(LIB\_EXT) !ENDIF  
SOCK\_LIB= !ELSE !ERROR An invalid Instrumentation \$(INSTRUMENTATION) is specified.  
!ENDIF

Generated Dependencies The generated dependencies section of the makefile contains a variable that expands to Rational Rhapsody-generated dependencies and compilation instructions. For example, the generated dependencies section of a C++ makefile for the Microsoft environment is: #####  
Generated dependencies #####  
##### SOMContextDependencies  
\$OMFileObjPath : \$OMMainImplementationFile \$(OBJS) \$(CPP) \$(ConfigurationCPPCompileSwitches)  
/Fo"\$OMFileObjPath" \$OMMainImplementationFile

Linking Instructions The linking instructions section of the makefile contains the predefined linking instructions. For example, the default linking instructions section of a C++ makefile for the Microsoft environment is: ##### Linking instructions #####  
#####  
\$(TARGET\_NAME)\$(EXE\_EXT): \$(OBJS) \$(ADDITIONAL\_OBJS) \$OMFileObjPath  
\$OMMakefileName \$OMModelLibs @echo Linking \$(TARGET\_NAME)\$(EXE\_EXT) \$(LINK\_CMD)  
\$OMFileObjPath \$(OBJS) \$(ADDITIONAL\_OBJS) \ \$(LIBS) \ \$(INST\_LIBS) \ \$(OXF\_LIBS) \ \$(SOCK\_LIB) \ \$(LINK\_FLAGS) /out:\$(TARGET\_NAME)\$(EXE\_EXT)  
\$(TARGET\_NAME)\$(LIB\_EXT) : \$(OBJS) \$(ADDITIONAL\_OBJS) \$OMMakefileName @echo Building library \$@ \$(LIB\_CMD) \$(LIB\_FLAGS) /out:\$(TARGET\_NAME)\$(LIB\_EXT) \$(OBJS) \$(ADDITIONAL\_OBJS) clean: @echo Cleanup \$OMCleanOBJS if exist \$OMFileObjPath erase \$OMFileObjPath if exist \*\$(OBJ\_EXT) erase \*\$(OBJ\_EXT) if exist \$(TARGET\_NAME).pdb erase \$(TARGET\_NAME).pdb if exist \$(TARGET\_NAME)\$(LIB\_EXT) erase \$(TARGET\_NAME)\$(LIB\_EXT) if exist \$(TARGET\_NAME).ilk erase \$(TARGET\_NAME).ilk if exist \$(TARGET\_NAME)\$(EXE\_EXT) erase \$(TARGET\_NAME)\$(EXE\_EXT) \$(CLEAN\_OBJ\_DIR)

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the C\_CG::<Environment>::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## ObjCleanCommand

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*(Default = \$(RM) \$OMFileObjPath)*



## ObjectName

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = empty string*

## ObjectsDirectory

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = empty string*

## ObjExtension

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*(Default = .o)*

## OSFileSystemCaseSensitive

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Cleared*

## OSVersion

Used for the various Wind River environments, the OSVersion property represents the version number of the product. The value of the property is used in the batch files that build applications and framework libraries for these environments. In these batch files, the value of the property is provided as a parameter for both the "wrenv" command and the "make" command.

*Default = 6.8*

## ParseErrorDescript

The ParseErrorDescript property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = ([^:]+)[:]( [0-9]+)[:] (error|warning)[:] (.\*)*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning)

Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*(Default = ([^:]+):?([0-9]+):?)*

## ParseMakeError

The ParseMakeError property is used to define a regular expression that represents the format of make process or linker error messages. This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = (.\*)(make)(.)(Error)*

## ParseSeverityError

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^:]+):?([0-9]+):? (error)*

## ParseSeverityWarning

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^:]+):?([0-9]+):? (warning)*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation.

*(Default = /)*

## PathWhiteSpaceHandling

For different operating systems, there are different methods for handling spaces in file paths, for example,

enclosing the entire path in quotation marks. The PathWhiteSpaceHandling property allows you to specify the method that should be used for a given environment. The possible values are:

- NoHandling - the path should be left as is, with no special handling for spaces
- SurroundingQuotes - the entire path should be enclosed in quotation marks
- BackslashBeforeSpace - spaces in paths should be preceded by backslashes, as is the practice in VxWorks platforms

*Default = BackslashBeforeSpace*

## ProcessorArchitecture

The ProcessorArchitecture property is used to specify whether the application is being developed for 32-bit or 64-bit target systems. The value of the property is used in a number of places, such as the command used for building the Rhapsody framework, the content of generated makefiles, and the names of generated libraries.

Verify that ProcessorArchitecture is set to the appropriate value for the target systems. The possible values are x86 (for 32-bit) and x64 (for 64-bit).

Once you have set the property to the appropriate value, rebuild the Rhapsody framework, and generate and build your application.

*Default = x86*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*(Default = Checked )*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running the product, whereas the target is the machine running the application. To run remotely, the UseRemoteHost property must be set to True.

If UseRemoteHost is True and RemoteHost is blank, the current host name is used for the remote host. You can use this as a workaround if you have problems running animated applications on Windows 95. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = empty string*

## ReusableStatechartSwitches

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts. See the upgrade history on the support site for information on this change.

*(Default = -DOM\_REUSABLE\_STATECHART\_IMPLEMENTATION)*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*(Default = .h)*

## UpdateBuildSettingsInIDE

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## UseNewBuildOutputWindow

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the CG::General::ShowLogViewAfterBuild property to True.

*Default = Checked*

## UseNonZeroStdInputHandle

The UseNonZeroStdInputHandle property is a Boolean value that specifies whether to use a non-zero standard input handle.

*Default = Checked*

## UseRemoteHost

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Checked*

## WorkbenchManaged

The WorkbenchManaged metaclass contains the Environment settings (Compiler, framework libraries, and so on) for WorkbenchManaged compiler.

### AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*(Default = \$(OMROOT)/LangC/osconfig/VxWorks)*

### AdditionalReservedWords

The AdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use.

In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = empty string*

### AutoAttachToIDEDebugger

The AutoAttachToIDEDebugger property is used to specify that you would like the Workbench debugger to be automatically synchronized with the Rational Rhapsody animation. If for some reason you do not

want automatic synchronization, for example, if you prefer to manually connect to the relevant target server, then you can set the value of this property to False.

*Default = Checked*

## **BSP**

The BSP property specifies the board support package (BSP) for the system. If you need to change the value of the CPU, you can simply reset the value in this property instead of changing the value in the MakeFileContent property.

*(Default = PENTIUM)*

## **BuildArgumentsInIDE**

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the C\_CG::[environment]::BuildCommandInIDE property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## **BuildCommandInIDE**

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## **BuildCommandSet**

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration. To change this property, use the Configuration window in the browser - do not change it by using the Properties window or by modifying the site.prp file.

Note that this property also affects the names of the framework libraries used in the link. The possible values are:

- **Debug** - Generate the debug command that is set in the makefile.
- **DebugNoExp** - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- **Release** - Generate the release command that is set in the makefile.
- **ReleaseNoExp** - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*(Default = Debug)*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

*Default = "\$OMROOT/etc/Executer.exe" "\""\$OMROOT\etc\vx6make.bat" vxbuild.mak buildLibs 6.5 \"CPU=\$BSP\" \"TOOL=\$Tool\" \"TOOL\_FAMILY=\$Tool\" \"BUILD=\$BuildCommandSet\" \" \"*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Checked*

## CodeTestSettings

The CodeTestSettings property specifies the compiler command settings. This property supports integration with Applied Microsystems Corporation CodeTest.

*(Default = CC = \$(AMC\_HOME)\bin\ctcc)*

## CompileSwitches

The CompileSwitches property specifies the compiler switches.

The default value is:

`$IgnoreSwitches -I$OMDefaultSpecificationDirectory -I$(OMROOT)/LangC -I$(OMROOT)/LangC/oxf -DVxWorks $(INST_FLAGS) $(INCLUDE_PATH) $OMCPCPPCompileCommandSet -Wno-unused -c`

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The default value is:

```
@echo Compiling $OMFileImpPath $(CREATE_OBJ_DIR) @$ (CC) $(CFLAGS)
$OMFileCPPCompileSwitches -o $OMFileObjPath $OMFileImpPath
```

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*(Default = -O -g)*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*(Default = empty string )*

## DependencyRule

The DependencyRule property specifies how file dependencies for a configuration are generated in the makefile.

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: \$OMFileObjPath : \$OMFileImpPath "\*.bmp" "\*.ico" "\*.rc2"

The default value is \$OMFileObjPath : \$OMFileImpPath \$OMFileSpecPath \$OMFileDependencies.

## DuplicateLibsListInMakeFile

The DuplicateLibsListInMakeFile property is a Boolean value that specifies whether Rational Rhapsody should duplicate the libraries list in the generated makefile link command. This property supports linkers that are sensitive to library order in the link command.

*Default = Checked*

## EnableDebugIntegrationWithIDE



When you use Rational Rhapsody in conjunction an IDE such as Eclipse, the `EnableDebugIntegrationWithIDE` property can be used to specify whether or not the IDE debugger should be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to `True`, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's `exit()` function is called automatically. The system function calls the destructors of framework instances in the order it determines. The `EndApplicationCode` property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an `endApplication()` function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to `endApplication()`, it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The `EntryPoint` property specifies the name of the main program for a given environment.

*(Default = vxmain)*

Use the "Filter" facility in this window to see the definition of the `EntryPointDeclarationModifier` property for more information.

## ErrorMessageTokensFormat

The `ErrorMessageTokensFormat`, working with the `ParseErrorMessage` property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. `ErrorMessageTokensFormat` defines the number and location of tokens within the regular expression defined by the `ParseErrorMessage` property. `ErrorMessageTokens` has three parameters, each with an integer value:

- `TotalNumberOfTokens` - The number of tokens in the regular expression
- `FileTokenPosition` - The position of the file name token in the expression
- `LineTokenPosition` - The position of the line number token in the expression

*(Default = TotalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2)*

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the C\_CG::<Environment>::ExeName property plus the value of the ExeExtension property.

*Default = .out*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the C\_CG::<Environment>::ExeExtension property.

*(Default = Blank)*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements.

The file inclusions are generated in the makefile.

The default value is \$OMSpecIncludeInElements \$OMImpIncludeInElements.

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile.

Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

This default value is \$(OMROOT)/LangC/lib/vxWebComponents\$(CPU)\$(TOOL)\$(LIB\_EXT), \$(OMROOT)/lib/vxWebServices\$(CPU)\$(TOOL)\$(LIB\_EXT)).

## HasIDEInterface

The HasIDEInterface property is a Boolean value that specifies whether IDE support is enabled.

If IDE support is enabled (Checked), the IDEInterfaceDLL property points to an IDE adapter that provides connection to the IDE. If the property is set to Cleared, IDE support is disabled and IDE services are not attempted. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = Checked*

## IDEInterfaceDLL

The IDEInterfaceDLL property is a string that points to the IDE adapter DLL. You should not have any reason to modify this property. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*(Default = \$OMROOT/DLLs/WorkbenchDebuggerIDE.dll)*

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment. The default values are:

*(Default = .c)*

## Include

The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*(Default = include)*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

The default value is:

```
"$OMROOT/etc/Executer.exe" "\"$OMROOT\etc\vx6make.bat\" $makefile $maketarget"
```

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*(Default = .a)*

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

*(Default = -g)*

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

*(Default = \$OMLinkCommandSet -MD -MP)*

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the C\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*(Default = .makefile)*

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration. For example, the default content of the target type section of a C++ makefile for the Microsoft environment is: ##### Target type (Debug/Release)

```
#####  
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease  
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease BuildSet=$OMBuildSet  
SUBSYSTEM=$OMSubSystem COM=$OMCOM RPFrameWorkDll=$OMRPFrameWorkDll  
ConfigurationCPPCompileSwitches= $OMReusableStatechartSwitches  
$OMConfigurationCPPCompileSwitches !IF "$ (RPFrameWorkDll)" == "True"  
ConfigurationCPPCompileSwitches= $(ConfigurationCPPCompileSwitches) /D "FRAMEWORK_DLL"  
!ENDIF !IF "$ (COM)" == "True" SUBSYSTEM=/SUBSYSTEM:windows !ENDIF
```

**Compilation Flags** The compilation flags section of the makefile contains the default compilation flags stored in the CompileSwitches property. For example, the default content of the compilation flags section of a C++ makefile for the Microsoft environment is: ##### Compilation flags

```
#####  
INCLUDE_QUALIFIER=/I LIB_PREFIX=MS
```

**Commands Definitions** The commands definition section of the makefile specifies programs to execute from the makefile. For example, the default commands definition section of a C++ makefile for the Microsoft environment is: ##### Commands definition #####

```
##### RMDIR = rmdir LIB_CMD=link.exe -lib  
LINK_CMD=link.exe LIB_FLAGS=$OMConfigurationLinkSwitches  
LINK_FLAGS=$OMConfigurationLinkSwitches $(SUBSYSTEM) / MACHINE:I386
```

**Generated Macros** The generated macros section of the makefile contains a variable that expands to the Rational Rhapsody-generated macros in the makefile. For example: ##### Generated macros ##### \$OMContextMacros  
OBJ\_DIR=\$OMObjectsDir !IF "\$ (OBJ\_DIR)" != "" CREATE\_OBJ\_DIR=if not exist \$(OBJ\_DIR) mkdir  
\$(OBJ\_DIR) CLEAN\_OBJ\_DIR= if exist \$(OBJ\_DIR) \$(RMDIR) \$(OBJ\_DIR) !ELSE  
CREATE\_OBJ\_DIR= CLEAN\_OBJ\_DIR= !ENDIF

The \$OMContextMacros keyword expands several macros in the makefile. Each makefile macro has its own keyword. You can use these keywords separately to customize the makefile. The

\$OMContextMacros variable modifies target-specific variables.

Replace the \$OMContextMacros line in the MakeFileContent property with the following:

```
FLAGSFILE=$OMFlagsFile RULESFILE=$OMRulesFile OMROOT=$OMROOT
C_EXT=$OMImplExt H_EXT=$OMSpecExt OBJ_EXT=$OMObjExt EXE_EXT=$OMExeExt
LIB_EXT=$OMLibExt INSTRUMENTATION=$OMInstrumentation TIME_MODEL=$OMTimeModel
TARGET_TYPE=$OMTargetType TARGET_NAME=$OMTargetName $OMAllDependencyRule
TARGET_MAIN=$OMTargetMain LIBS=$OMLibs INCLUDE_PATH=$OMIncludePath
ADDITIONAL_OBJS=$OMAdditionalObjs OBJS= $OMObjs
```

**Predefined Macros** The predefined macros section of the makefile contains other macros than the Rational Rhapsody-generated macros specified in the generated macros section. For example, the default predefined macros section of a C++ makefile for the Microsoft environment is: #####

Predefined macros #####

```
##### $(OBJS) : $(INST_LIBS)
$(OXF_LIBS) LIB_POSTFIX= !IF "$$(BuildSet)"=="Release" LIB_POSTFIX=R !ENDIF !IF
"$(TARGET_TYPE)"=="Executable" LinkDebug=$(LinkDebug) /DEBUG
LinkRelease=$(LinkRelease) /OPT:NOREF !ELSEIF "$(TARGET_TYPE)"=="Library"
LinkDebug=$(LinkDebug) /DEBUGTYPE:CV !ENDIF !IF "$(INSTRUMENTATION)"=="Animation"
INST_FLAGS=/D "OMANIMATOR" INST_INCLUDES=/I $(OMROOT)\LangCpp\om /I
$(OMROOT)\LangCpp\tom !IF "$(RPFrameWorkDll)"=="True" INST_LIBS=
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxanimdll$(LIB_POSTFIX) $(LIB_EXT)
!ELSE INST_LIBS= $(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomanim$(LIB_POSTFIX) $(LIB_EXT)
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfinst$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)"=="Tracing" INST_FLAGS=/D
"OMTRACER" INST_INCLUDES=/I $(OMROOT)\LangCpp\om /I $(OMROOT)\LangCpp\tom !IF
"$(RPFrameWorkDll)"=="True" INST_LIBS= OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfracedll$(LIB_POSTFIX) $(LIB_EXT) !ELSE
INST_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)tomtrace$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)aomtrace$(LIB_POSTFIX) $(LIB_EXT) OXF_LIBS=
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfinst$(LIB_POSTFIX) $(LIB_EXT)
$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)omComAppl$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB=wsock32.lib !ELSEIF "$(INSTRUMENTATION)"=="None" INST_FLAGS=
INST_INCLUDES= INST_LIBS= !IF "$(RPFrameWorkDll)"=="True"
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfdll$(LIB_POSTFIX) $(LIB_EXT) !ELSE
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$(LIB_POSTFIX) $(LIB_EXT) !ENDIF
SOCK_LIB= !ELSE !ERROR An invalid Instrumentation $(INSTRUMENTATION) is specified.
!ENDIF
```

**Generated Dependencies** The generated dependencies section of the makefile contains a variable that expands to Rational Rhapsody-generated dependencies and compilation instructions. For example, the generated dependencies section of a C++ makefile for the Microsoft environment is: #####

Generated dependencies #####

```
##### $OMContextDependencies
$OMFileObjPath : $OMMainImplementationFile $(OBJS) $(CPP) $(ConfigurationCPPCompileSwitches)
/Fo"$OMFileObjPath" $OMMainImplementationFile
```

**Linking Instructions** The linking instructions section of the makefile contains the predefined linking instructions. For example, the default linking instructions section of a C++ makefile for the Microsoft environment is: ##### Linking instructions #####

```
#####
$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS) $OMFileObjPath
$OMMakefileName $OMModelLibs @echo Linking $(TARGET_NAME)$(EXE_EXT) $(LINK_CMD)
```

```

$OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \ $(OXF_LIBS) \
$(SOCK_LIB) \ $(LINK_FLAGS) /out:$(TARGET_NAME)$(EXE_EXT)
$(TARGET_NAME)$(LIB_EXT) : $(OBJS) $(ADDITIONAL_OBJS) $OMMakefileName @echo
Building library $@ $(LIB_CMD) $(LIB_FLAGS) /out:$(TARGET_NAME)$(LIB_EXT) $(OBJS)
$(ADDITIONAL_OBJS) clean: @echo Cleanup $OMCleanOBJS if exist $OMFileObjPath erase
$OMFileObjPath if exist *$(OBJ_EXT) erase *$(OBJ_EXT) if exist $(TARGET_NAME).pdb erase
$(TARGET_NAME).pdb if exist $(TARGET_NAME)$(LIB_EXT) erase
$(TARGET_NAME)$(LIB_EXT) if exist $(TARGET_NAME).ilk erase $(TARGET_NAME).ilk if exist
$(TARGET_NAME)$(EXE_EXT) erase $(TARGET_NAME)$(EXE_EXT) $(CLEAN_OBJ_DIR)

```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the C\_CG::<Environment>::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*(Default = Blank)*

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## ObjCleanCommand

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default = \$(RM) \$OMFileObjPath*

## ObjectName

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = empty string*

## ObjectsDirectory

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the

generated makefile.

*Default = empty string*

## **ObjExtension**

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*Default = .o*

## **OSFileSystemCaseSensitive**

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Cleared*

## **OSVersion**

Used for the various Wind River environments, the OSVersion property represents the version number of the product. The value of the property is used in the batch files that build applications and framework libraries for these environments. In these batch files, the value of the property is provided as a parameter for both the "wrenv" command and the "make" command.

*Default = 6.8*

## **ParseErrorMessage**

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning)

Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = ([^:]+):([0-9]+):*

## **PathDelimiter**

The PathDelimiter property specifies an alternative path separator for code generation.

*Default = /*



## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Checked*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running the product, whereas the target is the machine running the application.

To run remotely, the UseRemoteHost property must be set to True. If UseRemoteHost is True and RemoteHost is blank, the current host name is used for the remote host. You can use this as a workaround if you have problems running animated applications on Windows 95. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = empty string*

## ReusableStatechartSwitches

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts. See the upgrade history on the support site for information on this change.

*(Default = -DOM\_REUSABLE\_STATECHART\_IMPLEMENTATION)*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*(Default = .h)*

## UnixPathNameForOMROOT

The UnixPathNameForOMROOT property specifies whether the makefile must include UNIXstyle path names.

The pRISM compilers do not tolerate DOS-style path conventions. If you do not set this property correctly, there might be many compilation problems.

*Default = Checked*

## UpdateBuildSettingsInIDE

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## UseNonZeroStdInputHandle

The UseNonZeroStdInputHandle property is a Boolean value that specifies whether to use a non-zero standard input handle.

*Default = Checked*

## UseRemoteHost

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Checked*

## WorkbenchManaged\_RTP

The WorkbenchManaged\_RTP metaclass contains the Environment settings (Compiler, framework libraries, and so on) for WorkbenchManaged\_RTP compiler.

## AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path.

Previously, the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files.

To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*(Default = \$(OMROOT)/LangC/osconfig/VxWorks)*

## **AdditionalReservedWords**

The AdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use.

In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

The property value is checked at run time when you name or rename an element, based on the active configuration environment setting.

Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = empty string*

## **AutoAttachToIDEDebugger**

The AutoAttachToIDEDebugger property is used to specify that you would like the Workbench debugger to be automatically synchronized with the Rational Rhapsody animation. If for some reason you do not want automatic synchronization, for example, if you prefer to manually connect to the relevant target server, then you can set the value of this property to False.

*Default = Checked*

## **BSP**

The BSP property specifies the board support package (BSP) for the system. If you need to change the value of the CPU, you can simply reset the value in this property instead of changing the value in the MakeFileContent property.

*Default = PENTIUM*

## **BuildArgumentsInIDE**

The BuildArgumentsInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. The property allows you to enter a string of arguments that is used as build arguments if you are not using the IDE default build command.

The arguments provided here are only used if the value of the C\_CG::[environment]::BuildCommandInIDE property is not an empty string.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandInIDE

The BuildCommandInIDE property is used when building an application in an IDE that has been integrated with Rational Rhapsody, such as Eclipse. If this property is left blank, the IDE default build command is used. If you enter a different string, then the command you entered is used when building the application rather than the IDE default build command.

Keep in mind that the build is performed by the IDE only if the value of the BuildInIDE property is set to True.

*Default = Blank*

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration. To change this property, use the Configuration window in the browser - do not change it by using the Properties window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link. The possible values are:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*Default = Debug*

## buildFrameworkCommand

The Code menu in Rational Rhapsody includes an option called Build Framework. When you select this option, Rational Rhapsody rebuilds its framework libraries for the environment specified on the Settings tab of the Features window for the active configuration. The buildFrameworkCommand property is used to specify the command that should be carried out when the Build Framework option is selected.

```
Default = "$OMROOT/etc/Executer.exe" "\""$OMROOT/etc/vx6make.bat" vxbuild.mak buildLibs 6.5  
\"CPU=$BSP\" \"TOOL=$Tool\" \"TOOL_FAMILY=$Tool\" \"BUILD=$BuildCommandSet\"  
\"DISTRIBUTED=TRUE\" \" \"
```

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = Checked*

## CodeTestSettings

The CodeTestSettings property specifies the compiler command settings. This property supports integration with Applied Microsystems Corporation CodeTest.

*Default = CC = \$(AMC\_HOME)\bin\ctcc*

## CompileSwitches

The CompileSwitches property specifies the compiler switches.

The default value is:

`$IgnoreSwitches -I$OMDefaultSpecificationDirectory -I$(OMROOT)/LangC -I$(OMROOT)/LangC/oxf  
-DVxWorks $(INST_FLAGS) $(INCLUDE_PATH) $OMCPPCompileCommandSet -Wno-unused -c`

## CPPCompileCommand

The CPPCompileCommand property is a string that specifies a different compile command. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default =*

`@echo Compiling $OMFileImpPath $(CREATE_OBJ_DIR) @$ (CC) $(CFLAGS)  
$OMFileCPPCompileSwitches -o $OMFileObjPath $OMFileImpPath`

## CPPCompileDebug

The CPPCompileDebug property modifies the makefile compile command with switches for building a debug version of the component.

*Default = -O -g*

## CPPCompileRelease

The CPPCompileRelease property specifies additional compilation flags for a configuration set to Release mode.

*Default = empty string*

## DependencyRule

The `DependencyRule` property specifies how file dependencies for a configuration are generated in the makefile.

For example, the following dependency rule lists the file dependencies for a Windows application with a GUI, including bitmaps, icons, and resource files: `$OMFileObjPath : $OMFileImpPath "*.bmp" "*.ico" "*.rc2"`

The default value is `$OMFileObjPath : $OMFileImpPath $OMFileSpecPath $OMFileDependencies`.

## DuplicateLibsListInMakeFile

The `DuplicateLibsListInMakeFile` property is a Boolean value that specifies whether Rational Rhapsody should duplicate the libraries list in the generated makefile link command. This property supports linkers that are sensitive to library order in the link command.

*Default = Checked*

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction an IDE such as Eclipse, the `EnableDebugIntegrationWithIDE` property can be used to specify whether or not the IDE debugger should be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to `True`, the IDE debugger is used.

*Default = Cleared*

## EndApplicationCode

When the main function ends, the system's `exit()` function is called automatically. The system function calls the destructors of framework instances in the order it determines. The `EndApplicationCode` property provides a mechanism for a more orderly destruction of these instances.

The value of this property should be the code that you would like to run when the application ends.

Rational Rhapsody provides an `endApplication()` function that you can call instead of providing your own code or in addition to any such code that you would like to run. If you include a call to `endApplication()`, it should come after any additional code that you have specified.

*Default = Blank*

## EntryPoint

The `EntryPoint` property specifies the name of the main program for a given environment.

*(Default = main)*

Use the "Filter" facility in this window to see the definition of the EntryPointDeclarationModifier property for more information.

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens has three parameters, each with an integer value:

- TotalNumberOfTokens - The number of tokens in the regular expression
- FileTokenPosition - The position of the file name token in the expression
- LineTokenPosition - The position of the line number token in the expression

*(Default = TotalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2)*

## ExeExtension

The ExeExtension property is used to specify the file extension you would like to use for the executable program created by Rational Rhapsody.

Note that the full name of the executable program is composed of the value of the C\_CG::<Environment>::ExeName property plus the value of the ExeExtension property.

*Default = .vxe*

## ExeName

By default, the name of the executable program created by Rational Rhapsody is the name of the active component. If you would like to use a different name for the executable program, enter the name as the value of the ExeName property.

If you leave the value of the property blank, the name of the active component is used.

The name provided for this property is used both for executable programs and for libraries.

Note that the full name of the executable program is composed of the value of this property plus the value of the C\_CG::<Environment>::ExeExtension property.

*Default = Blank*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements.

The file inclusions are generated in the makefile.

The default value is `$OMSpecIncludeInElements $OMImpIncludeInElements`.

## GetConnectedRuntimeLibraries

The `GetConnectedRuntimeLibraries` property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile.

Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

This default value is

`$(OMROOT)/LangC/lib/vx$(DIST_PREFIX)WebComponents$(CPU)$(RTP_SUFFIX)$(TOOL)$(RHP_LIB_EXT),  
$(OMROOT)/lib/vxWebServices$(CPU)$(RTP_SUFFIX)$(TOOL)$(RHP_LIB_EXT)`.

## HasIDEInterface

The `HasIDEInterface` property is a Boolean value that specifies whether IDE support is enabled.

If IDE support is enabled (Checked), the `IDEInterfaceDLL` property points to an IDE adapter that provides connection to the IDE. If the property is set to Cleared, IDE support is disabled and IDE services are not attempted. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = Checked*

## IDEInterfaceDLL

The `IDEInterfaceDLL` property is a string that points to the IDE adapter DLL. You should not have any reason to modify this property. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody.

*Default = \$OMROOT/DLLs/WorkbenchDebuggerIDE.dll*

## ImpExtension

The `ImpExtension` property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment. The default values are:

*(Default = .c)*

## Include



The Include property specifies the environment-specific command that is generated in the makefile to include other makefiles.

*Default = include*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is:

```
"rhapsody_dir>\share/etc/Executer.exe""dir; dir | more"
```

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\vx6make.bat\" \$makefile \$maketarget"*

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = .a*

## LinkDebug

The LinkDebug property specifies the special link switches used to link in debug mode.

*Default = empty string*

## LinkRelease

The LinkRelease property specifies the special link switches used to link in release mode.

## LinkSwitches

The LinkSwitches property specifies the standard link switches used to link in any mode.

*Default = \$OMLinkCommandSet*

## MakeExtension

The MakeExtension property can be used to specify the file extension you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the extension that you would like to use. Note that the first part of the filename can be customized by modifying the value of the C\_CG::<Environment>::MakeFileName property.

If you do not want Rational Rhapsody to add a file extension, leave the value of this property blank.

*Default = .makefile*

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration. For example, the default content of the target type section of a C++ makefile for the Microsoft environment is as follows: ##### Target type (Debug/Release)

```
#####  
CPPCompileDebug=$OMCPPCompileDebug CPPCompileRelease=$OMCPPCompileRelease  
LinkDebug=$OMLinkDebug LinkRelease=$OMLinkRelease BuildSet=$OMBuildSet  
SUBSYSTEM=$OMSubSystem COM=$OMCOM RPFrameWorkDll=$OMRPFrameWorkDll  
ConfigurationCPPCompileSwitches= $OMReusableStatechartSwitches  
$OMConfigurationCPPCompileSwitches !IF "$ (RPFrameWorkDll)" == "True"  
ConfigurationCPPCompileSwitches= $(ConfigurationCPPCompileSwitches) /D "FRAMEWORK_DLL"  
!ENDIF !IF "$ (COM)" == "True" SUBSYSTEM=/SUBSYSTEM:windows !ENDIF
```

**Compilation Flags** The compilation flags section of the makefile contains the default compilation flags stored in the CompileSwitches property. For example, the default content of the compilation flags section

of a C++ makefile for the Microsoft environment is as follows: ##### Compilation flags  
#####  
INCLUDE\_QUALIFIER=/I LIB\_PREFIX=MS

**Commands Definitions** The commands definition section of the makefile specifies programs to execute from the makefile. For example, the default commands definition section of a C++ makefile for the Microsoft environment is as follows: ##### Commands definition #####  
##### RMDIR = rmdir LIB\_CMD=link.exe -lib  
LINK\_CMD=link.exe LIB\_FLAGS=\$OMConfigurationLinkSwitches  
LINK\_FLAGS=\$OMConfigurationLinkSwitches \$(SUBSYSTEM) / MACHINE:I386

**Generated Macros** The generated macros section of the makefile contains a variable that expands to the Rational Rhapsody-generated macros in the makefile. For example: ##### Generated macros  
##### \$OMContextMacros  
OBJ\_DIR=\$OMObjectsDir !IF "\$(OBJ\_DIR)"!=" CREATE\_OBJ\_DIR=if not exist \$(OBJ\_DIR) mkdir  
\$(OBJ\_DIR) CLEAN\_OBJ\_DIR= if exist \$(OBJ\_DIR) \$(RMDIR) \$(OBJ\_DIR) !ELSE  
CREATE\_OBJ\_DIR= CLEAN\_OBJ\_DIR= !ENDIF

The \$OMContextMacros keyword expands several macros in the makefile. Each makefile macro has its own keyword. You can use these keywords separately to customize the makefile. The \$OMContextMacros variable modifies target-specific variables. Replace the \$OMContextMacros line in the MakeFileContent property with the following: FLAGSFILE=\$OMFlagsFile  
RULESFILE=\$OMRulesFile OMROOT=\$OMROOT C\_EXT=\$OMImplExt H\_EXT=\$OMSpecExt  
OBJ\_EXT=\$OMObjExt EXE\_EXT=\$OMExeExt LIB\_EXT=\$OMLibExt  
INSTRUMENTATION=\$OMInstrumentation TIME\_MODEL=\$OMTimeModel  
TARGET\_TYPE=\$OMTargetType TARGET\_NAME=\$OMTargetName \$OMAllDependencyRule  
TARGET\_MAIN=\$OMTargetMain LIBS=\$OMLibs INCLUDE\_PATH=\$OMIncludePath  
ADDITIONAL\_OBJS=\$OMAdditionalObjs OBJS= \$OMObjs

**Predefined Macros** The predefined macros section of the makefile contains other macros than the Rational Rhapsody-generated macros specified in the generated macros section. For example, the default predefined macros section of a C++ makefile for the Microsoft environment is as follows:

##### Predefined macros #####  
##### \$(OBJS) : \$(INST\_LIBS)  
\$(OXF\_LIBS) LIB\_POSTFIX= !IF "\$(BuildSet)"=="Release" LIB\_POSTFIX=R !ENDIF !IF  
"\$(TARGET\_TYPE)" == "Executable" LinkDebug=\$(LinkDebug) /DEBUG  
LinkRelease=\$(LinkRelease) /OPT:NOREF !ELSEIF "\$(TARGET\_TYPE)" == "Library"  
LinkDebug=\$(LinkDebug) /DEBUGTYPE:CV !ENDIF !IF "\$(INSTRUMENTATION)" == "Animation"  
INST\_FLAGS=/D "OMANIMATOR" INST\_INCLUDES=/I \$(OMROOT)\LangCpp\aoom /I  
\$(OMROOT)\LangCpp\tom !IF "\$(RPFrameWorkDll)" == "True" INST\_LIBS=  
OXF\_LIBS=\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)oxanimdll\$(LIB\_POSTFIX) \$(LIB\_EXT)  
!ELSE INST\_LIBS= \$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)aomanip\$(LIB\_POSTFIX) \$(LIB\_EXT)  
OXF\_LIBS=\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)oxfirst\$(LIB\_POSTFIX) \$(LIB\_EXT)  
\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)omComApp\$(LIB\_POSTFIX) \$(LIB\_EXT) !ENDIF  
SOCK\_LIB=wsock32.lib !ELSEIF "\$(INSTRUMENTATION)" == "Tracing" INST\_FLAGS=/D  
"OMTRACER" INST\_INCLUDES=/I \$(OMROOT)\LangCpp\aoom /I \$(OMROOT)\LangCpp\tom !IF  
"\$(RPFrameWorkDll)" == "True" INST\_LIBS= OXF\_LIBS=  
\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)oxframedll\$(LIB\_POSTFIX) \$(LIB\_EXT) !ELSE  
INST\_LIBS=\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)tomtrace\$(LIB\_POSTFIX) \$(LIB\_EXT)  
\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)aomanip\$(LIB\_POSTFIX) \$(LIB\_EXT) OXF\_LIBS=  
\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)oxfirst\$(LIB\_POSTFIX) \$(LIB\_EXT)  
\$(OMROOT)\LangCpp\lib\\$(LIB\_PREFIX)omComApp\$(LIB\_POSTFIX) \$(LIB\_EXT) !ENDIF  
SOCK\_LIB=wsock32.lib !ELSEIF "\$(INSTRUMENTATION)" == "None" INST\_FLAGS=  
INST\_INCLUDES= INST\_LIBS= !IF "\$(RPFrameWorkDll)" == "True"

```

OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxfdll$(LIB_POSTFIX)$(LIB_EXT) !ELSE
OXF_LIBS=$(OMROOT)\LangCpp\lib\$(LIB_PREFIX)oxf$(LIB_POSTFIX)$(LIB_EXT) !ENDIF
SOCK_LIB= !ELSE !ERROR An invalid Instrumentation $(INSTRUMENTATION) is specified.
!ENDIF

```

**Generated Dependencies** The generated dependencies section of the makefile contains a variable that expands to Rational Rhapsody-generated dependencies and compilation instructions. For example, the generated dependencies section of a C++ makefile for the Microsoft environment is as follows:

```

##### Generated dependencies #####
##### SOMContextDependencies
$OMFileObjPath : $OMMainImplementationFile $(OBJS) $(CPP) $(ConfigurationCPPCompileSwitches)
/Fo"$OMFileObjPath" $OMMainImplementationFile

```

**Linking Instructions** The linking instructions section of the makefile contains the predefined linking instructions. For example, the default linking instructions section of a C++ makefile for the Microsoft environment is as follows:

```

##### Linking instructions #####
#####
$(TARGET_NAME)$(EXE_EXT): $(OBJS) $(ADDITIONAL_OBJS) $OMFileObjPath
$OMMakefileName $OMModelLibs @echo Linking $(TARGET_NAME)$(EXE_EXT) $(LINK_CMD)
$OMFileObjPath $(OBJS) $(ADDITIONAL_OBJS) \ $(LIBS) \ $(INST_LIBS) \ $(OXF_LIBS) \
$(SOCK_LIB) \ $(LINK_FLAGS) /out:$(TARGET_NAME)$(EXE_EXT)
$(TARGET_NAME)$(LIB_EXT) : $(OBJS) $(ADDITIONAL_OBJS) $OMMakefileName @echo
Building library $@ $(LIB_CMD) $(LIB_FLAGS) /out:$(TARGET_NAME)$(LIB_EXT) $(OBJS)
$(ADDITIONAL_OBJS) clean: @echo Cleanup $OMCleanOBJS if exist $OMFileObjPath erase
$OMFileObjPath if exist *$(OBJ_EXT) erase *$(OBJ_EXT) if exist $(TARGET_NAME).pdb erase
$(TARGET_NAME).pdb if exist $(TARGET_NAME)$(LIB_EXT) erase
$(TARGET_NAME)$(LIB_EXT) if exist $(TARGET_NAME).ilk erase $(TARGET_NAME).ilk if exist
$(TARGET_NAME)$(EXE_EXT) erase $(TARGET_NAME)$(EXE_EXT) $(CLEAN_OBJ_DIR)

```

## MakeFileName

The MakeFileName property can be used to specify the filename you would like to use for the makefile generated by Rational Rhapsody. For the value of this property, enter the name that you would like to use for the file.

Note that this property only specifies the first part of the filename. The extension is specified by using the C\_CG::<Environment>::MakeExtension property.

If the property value is left blank, Rational Rhapsody uses the name of the component.

*Default = Blank*

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## ObjCleanCommand

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default = \$(RM) \$OMFileObjPath*

## ObjectName

The ObjectName property specifies an alternative name for the compiled object file in the generated makefile.

*Default = empty string*

## ObjectsDirectory

The ObjectsDirectory property specifies an alternate name for the directory for compiled object files in the generated makefile.

*Default = empty string*

## ObjExtension

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*Default = .o*

## OSFileSystemCaseSensitive

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

*Default = Cleared*

## OSVersion

Used for the various Wind River environments, the OSVersion property represents the version number of the product. The value of the property is used in the batch files that build applications and framework libraries for these environments. In these batch files, the value of the property is provided as a parameter for both the "wrenv" command and the "make" command.

*Default = 6.8*

## ParseErrorMessage

The ParseErrorMessage property defines a regular expression that matches the format of error messages.

The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning)

Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = [""]([^:]+)[""][,][ ]line ([0-9]+)[:]*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation.

*Default = /*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Checked*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running the product, whereas the target is the machine running the application. To run remotely, the UseRemoteHost property must be set to True.

If UseRemoteHost is True and RemoteHost is blank, the current host name is used for the remote host. You can use this as a workaround if you have problems running animated applications on Windows 95. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = empty string*

## ReusableStatechartSwitches

The ReusableStatechartSwitches property defines the compilation switch that was added to the makefile to support reusable statecharts. See the upgrade history on the support site for information on this change.

*Default = -DOM\_REUSABLE\_STATECHART\_IMPLEMENTATION*

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*Default = .h*

## **UnixPathNameForOMROOT**

The UnixPathNameForOMROOT property specifies whether the makefile must include UNIXstyle path names.

The pRISM compilers do not tolerate DOS-style path conventions. If you do not set this property correctly, there might be many compilation problems.

*Default = Checked*

## **UpdateBuildSettingsInIDE**

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = Checked*

## **UseNonZeroStdInputHandle**

The UseNonZeroStdInputHandle property is a Boolean value that specifies whether to use a non-zero standard input handle.

*Default = Checked*

## **UseRemoteHost**

The UseRemoteHost property specifies whether you intend to run an application remotely over a network by default in a given environment.

*Default = Checked*

# C\_ReverseEngineering

---

In addition to the ReverseEngineering subject, Rational Rhapsody provides language-specific subjects to control how Rational Rhapsody imports legacy code. Most of the properties are identical for each language. Any language-specific properties are clearly labeled. In general, most of the reverse engineering properties have graphical representation in the Reverse Engineering Advanced Options window. You should change the options by using the Reverse Engineering Advanced Options window instead of the corresponding properties.

## Filtering

The Filtering metaclass contains properties that control which items are analyzed during the reverse engineering operation.

### AnalyzeGlobalFunctions

The AnalyzeGlobalFunctions property specifies whether to analyze global functions.

*Default = Checked*

### AnalyzeGlobalTypes

The AnalyzeGlobalTypes property specifies whether to analyze global types.

*Default = Checked*

### AnalyzeGlobalVariables

The AnalyzeGlobalVariables property specifies whether to analyze global variables.

*Default = Checked*

### CreateReferenceClasses

The CreateReferenceClasses property specifies whether to create external classes for undefined classes that result from forward declarations and inheritance. By default, reference classes are created (as in earlier versions of Rational Rhapsody). If the incomplete class cannot be resolved, the tool deletes the incomplete class if this property is set to Cleared. In some cases, the class cannot be deleted (for example, a class referenced by a typedef type).

*Default = Checked*



## IncludeInheritanceInReference

The IncludeInheritanceInReference property specifies whether to include inheritance information in reference classes.

*Default = Cleared*

## ReferenceClasses

The ReferenceClasses property specifies which classes to model as reference classes. Reference classes are classes that can be mentioned in the final design as placeholders without having to specify their internal details. For example, you can include the MFC classes as reference classes, without having to specify any of their members or relations. They would simply be modeled as terminals for context, to show that they are acting as superclasses or peers to other classes.

*Default = empty string*

## ReferenceDirectories

The ReferenceDirectories property specifies which directories (and subdirectories) contain reference classes.

*Default = empty string*

## ImplementationTrait

The ImplementationTrait metaclass contains properties that determine the implementation traits used during the reverse engineering operation.

## AnalyzeIncludeFiles

The AnalyzeIncludeFiles property specifies which, if any, include files should be analyzed during reverse engineering. The possible values are as follows:

- AllIncludes - Analyze all include files.
- IgnoreIncludes - Ignore all include files.
- OnlyFromSelected - Analyze the specified include files only.
- OnlyLogicalHeader - Analyze the logical header files only.

*(C Default = AllIncludes)*

## AutomaticIncludePath

When Rational Rhapsody reverse engineers a file, there might be cases where the file references a header file but the path in the include directive is not clear enough for the product to find the file. If you set the value of the AutomaticIncludePath property to Checked, then in such cases, Rational Rhapsody searches the list of files to be reverse engineered to see if the list contains a header file with that name. If there is such a file, Rational Rhapsody uses the full path that was provided for that header file, assuming that this is the header file that was being referenced in the original file.

Rational Rhapsody performs this search for ambiguous header files when it does macro collection. This means that if the value of the C\_ReverseEngineering::ImplementationTrait::CollectMode property is set to None, then Rational Rhapsody does not search for ambiguous header files even if the value of the AutomaticIncludePath property is set to Checked.

*Default = Checked*

## CreateBlackDiamondAssociations

The CreateBlackDiamondAssociations property specifies how the reverse engineering feature should handle composition relationships. If the value of the property is set to False, then Rational Rhapsody creates parts. If the value of the property is set to Checked, Rational Rhapsody creates composition associations (black diamond).

*Default = Cleared*

## CreateDependencies

The CreateDependencies property is used during reverse engineering (RE) for creating dependencies from include statements found in the imported code. This property determines whether the RE utility creates dependencies. Reverse engineering imports include statements as dependencies if the option Create Dependencies from Includes is set in the Rational Rhapsody GUI. This operation is successful if the reverse engineering utility analyzes both the included file and the source - and the source and included files contain class declarations for creating the dependencies between them. If there is not enough information, the includes are not converted dependencies. This can happen in the following cases:

- The include file was not found, or is not in the scope Input tab settings.
- A class is not defined in the include file or source file, so the dependency could not be created.

If the dependency is not created successfully, the include files that were not converted to dependencies are imported to the C.CG::Class::SpecIncludes or ImpIncludes properties so you do not have to re-create them manually. If the include file is in the specification file, the information is imported to the SpecIncludes property; if it is in the implementation file, the information is imported to the ImpIncludes property. If a file contains several classes, include information is imported for all the classes in the file. The possible values for this property are as follows:

- None - Nothing is imported from include statements.
- DependenciesOnly - Model dependencies are created from include statements when it is possible to do so. This is the RE behavior of previous versions of Rational Rhapsody.
- All - The reverse engineering utility attempts to map the include file as a dependency. If it fails, the information is written to a property.

In previous versions of Rational Rhapsody, this property was a Boolean value. For compatibility with earlier versions, the old values are mapped as follows:

Old Value New Value Checked DependenciesOnly Cleared None

1. In addition to influencing reverse engineering, the CreateDependencies property also impacts the reverse engineering of user code added to model elements. The rules for interpreting #include and friend declarations for reverse engineering are as follows:

- Any #include OTHER in FILE is represented as a Uses dependency between each (outer) packages or classes in FILE to any (outer) packages or class in OTHER.
- If OTHER is not a specification file, the information is lost.
- If FILE is a specification file, the RefereeEffect is Specification. If FILE is an implementation file, the RefereeEffect is Implementation. Otherwise, the information is lost.

2. Any forward of a class or a package ( by way of a namespace) E in FILE is represented as a Uses dependency between each (outer) packages/classes in FILE to E. The RefereeEffect is Existence.

3. This dependency is not added, if a Uses dependency can be matched.

4. Redundant Uses dependencies are removed. For example, when a relation is synthesized from a pointer to B, it is not necessary to add a Uses dependency.

5. A friend F (only when F is a class) of class C is represented as a dependency with DependencyType to be Friendship from F to C.

*Default = All)*

## CreateFilesIn

The CreateFilesIn property is a placeholder for the reverse engineering option Create File-s In option. See the Rational Rhapsody help for more information. You should not set this value directly. The default value for C is Package.

## CollectMode

The CollectMode property allows Rational Rhapsody to collect macros. The possible values are as follows:

- None - Macros are not collected from include files that are not on the reverse engineering list.
- Once - Macros are collected only if the model does not yet include a controlled file of collected macros.
- Always - Macros are collected each time reverse engineering is carried out. The controlled file that stores the macros are replaced each time.

*(C Default = None)*

## DataTypesLibrary

The Mapping tab of the Reverse Engineering Options window allows you to specify a list of types that should be modeled as "Language" types. You can add individual types to the list or groups of types that

you have previously defined as data types for a specific library.

If you select the option of adding a library, you are presented with a drop-down list of libraries to choose from. The libraries on this list are taken from the value of the `DataTypesLibrary` property. You can add a number of libraries to the drop-down list by using a comma-separated list of names as the value for this property.

When you select a library from the drop-down list, all of the types that were defined for that library are added to the list of types.

You define types for a library by carrying out the following steps:

- In the relevant .prp file, under the `C_ReverseEngineering` subject, add a metaclass with the name of the library (use the same name you used in the value of the `DataTypesLibrary` property).
- Under the new metaclass, add a property called `DataTypes`.
- For the value of the `DataTypes` property that you added, enter a comma-separated list of the types that you want to include for that library.
- Now, if you select the library from the drop-down list displayed on the Mapping tab, the types you defined with the `DataTypes` property is automatically added to the list of types that should be modeled as "Language" types.

*Default = Blank*

## ImportAsExternal

The `ImportAsExternal` property specifies whether the elements contained in the files you are reverse engineering should be brought into the model as "external" elements. This means that code is not generated for these elements during code generation.

This property corresponds to the Import as External check box on the Mapping tab of the Reverse Engineering Options window.

*Default = Cleared*

## ImportDefineAsType

The `ImportDefineAsType` property is a Boolean value that specifies how to import a `#define`. Note that models created before Version 5.2 automatically have this property overridden (set to True) when the model is loaded. The possible values are as follows:

- True - Import a `#define` as a user type.
- False - Import a `#define` as a constant variable, constant function, or type according to the following policy:
- If the `#define` has parameters, Rational Rhapsody creates a constant function. This applies to Rational Rhapsody Developer for C only.
- If the `#define` does not have parameters and its value includes only one line, Rational Rhapsody creates a constant variable. In Rational Rhapsody Developer for C++, the `CG::Attribute::ConstantVariableAsDefine` property is set to True.

- If the #define was not imported as a variable or function, Rational Rhapsody creates a type (the behavior of Rational Rhapsody 5.0.1).

*Default = False*

## ImportGlobalAsPrivate

The ImportGlobalAsPrivate property allows you to import C functions as public or private. The possible values are as follows:

- Never - Import globals (functions) as public. The declaration remains in the specification file.
- InImplementation - Global functions are imported as private. Both the declaration and the implementation of the function are imported into the implementation (.c) file.
- StaticInImplementation - Globals are imported as private in the implementation (.c) file and the functions are marked as static. (same as "InImplementation" but the keyword "static" is added to the declaration and implementation of the function).

## ImportPreprocessorDirectives

Before running reverse engineering or roundtrip operations, set the ImportPreprocessorDirectives property to preserve the order of preprocessor directives during code generation. This property is only used if the C\_Roundtrip:General:RoundtripScheme property is set to the "Respect" scheme.

However, the order of #include and #define is always preserved regardless of the setting of the ImportPreprocessorDirectives property.

*Default = Checked*

## ImportStructAsClass

The ImportStructAsClass property is a Boolean value specifies how structs in external code are imported during reverse engineering. The possible values are as follows:

- Checked - structs are imported as classes (as in Rational Rhapsody 5.0 and earlier).
- Cleared - structs are imported as types of kind Structure.

*Default = Cleared*

## LocalizeRespectInformation

When reverse engineering code in Respect mode, Rational Rhapsody stores information such as the order of code elements so that when code is regenerated from the model, the code resembles as much as possible the original code.

When the LocalizeRespectInformation property is set to Checked, Rational Rhapsody stores this information as SourceArtifact elements below the relevant class. (These elements are not visible by default, but you can see them in the model if you set the value of the ShowSourceArtifacts property to True.)

If the value of the LocalizeRespectInformation property is set to Cleared, then Rational Rhapsody stores this "respect" information as File elements under the relevant Component.

*Default = Checked*

## MacroExpansion

Early versions of Rational Rhapsody were not capable of importing macros in code such that they would be regenerated as macros. Rather, the code represented by the macro was stored in the model, and when the code was regenerated, the macro calls would be replaced with the relevant code.

Now, by default, Rational Rhapsody imports macros such that when the code is regenerated, the macro definition and macro calls are generated as they appeared in the original code that was reverse engineered.

If you would like the previous Rational Rhapsody behavior, that is, replacement of macro calls with the actual macro code, you can set the MacroExpansion property to Checked.

Note that the C\_ReverseEngineering::Parser::ForceExpansionMacros property allows you to specify that individual macros should be expanded during reverse engineering even if the value of the MacroExpansion property is set to False.

*Default = Cleared*

## MapGlobalsToComponentFiles

The MapGlobalsToComponentFiles property allows you to specify whether Rational Rhapsody should map global variables, functions, and types to component files, reflecting the original file location of these elements in the files that were reverse engineered. The property can take any of the following values:

- OnExternal - Global variables, functions, and types should be mapped to component files only if the user selected the reverse engineering option "Import as External"
- TypesOnly - Global types should be mapped to component files, but not global variables and functions
- TypesOnExternal - Only global types should be mapped to component files, and this should only be done if the user selected the reverse engineering option "Import as External"
- False - Global variables, functions, and types should not be mapped to component files

*Default = TypesOnExternal*

## MapToPackage

The MapToPackage property allows you to specify how the code elements you are reverse engineering should be divided into packages.

The property represents the options that appear in the Map to Package section of the Mapping tab in the Reverse Engineering Options window.

When the value of the property is set to Directory, a separate package is created for each subdirectory in the directory you have chosen to reverse engineer. The elements found in the files in each subdirectory is

added to the package that corresponds to that subdirectory.

If you set the value of this property to User, then Rational Rhapsody puts all reverse engineered elements into a single package in the model. The name of the package is taken from the property `C_ReverseEngineering::ImplementationTrait::UserPackage`.

*Default = Directory*

## ModelStyle

The ModelStyle property determines how model elements are opened in the browser after reverse engineering - by using a file-based functional approach or by using an object-oriented approach based on classes (the corresponding property values are Functional and ObjectBased).

This property corresponds to the Modeling Policy radio buttons on the Mapping tab of the Reverse Engineering Options window.

Note that for C++ and Java, the file-based approach can only be used for visualization purposes. Rational Rhapsody does not generate code from the model for elements imported that uses the Functional option. (Notice that in the Reverse Engineering Options window, you can only select the File radio button if you first select the Visualization Only option.)

*Default = Functional in RiC, ObjectBased in RiC++ and RiJ*

## PackageForExternals

If the value of the UsePackageForExternals property is set to Checked, the Rational Rhapsody reverse engineering feature puts all external elements in a separate package. You can control the name of this package by changing the value of the PackageForExternals property.

*Default = Externals*

## PreCommentSensibility

During reverse engineering, a comment that comes immediately before the code for an element is considered a comment for that element, and the comment text is brought into Rational Rhapsody as the description for that element.

The PreCommentSensibility property is used to specify the maximum number of lines by which a comment can precede the code for an element and still be considered a comment for that element. Any comment that precedes an element by more than the number of lines specified is considered a global comment.

A value of 1 means that a comment must appear on the line prior to the code for an element to be considered a comment for that element.

*Default = 2*

## ReflectDataMembers

The ReflectDataMembers property determines how the visibility of attributes is brought into the model when code is reverse engineered. The property affects both the visibility of the attribute in the regenerated code and the generation of get and set operations for the attribute. The property can take any of the following values:

- None - The visibility used for attributes is the same as that specified in the code that was reverse engineered. However, Rational Rhapsody generates public get/set operations for the attributes regardless of the visibility specified.
- VisibilityOnly - The visibility used for attributes is the same as that specified in the code that was reverse engineered. In addition, Rational Rhapsody generates get/set operations for the attribute with the same visibility. For example, if the visibility for an attribute in the original code was private, the visibility is private in the regenerated code and the code also includes private get/set operations for the attribute.
- VisibilityAndHelpers - The visibility used for attributes is the same as that specified in the code that was reverse engineered. Rational Rhapsody does not generate get/set operations for the attribute if the original code did not contain such operations.

Note that when the property is set to VisibilityAndHelpers, get/set operations are not generated for attributes, and Rational Rhapsody does not generate any of its automatically-generated operations such as default constructors.

*Default = VisibilityAndHelpers*

## RespectCodeLayout

The RespectCodeLayout property determines to what degree Rational Rhapsody attempts to save information about the code that is reverse engineered so that it is possible to match the original code when code is later regenerated from the model. The saved information includes:

- order of #includes and other code elements
- handling of preprocessor directives such as #ifdefs
- keeping macro calls as they were rather than expanding the macro in the regenerated code
- handling of global comments

The property can take any of the following values:

- None - Rational Rhapsody does not save information about the order of elements in the code that is imported, nor does it save the information necessary to regenerate all elements back to the files from which they were originally imported.
- Mapping - Rational Rhapsody saves partial information so that it can regenerate all elements back to the files from which they were originally imported.
- Ordering - Rational Rhapsody saves all the information it can so that the regenerated code matches the original code as much as possible. See the examples listed above.

Note that even if the value of this property is set to Ordering, Rational Rhapsody only attempts to match the regenerated code to the original code if the C\_CG::Configuration::CodeGeneratorTool property is set to Advanced, which is the default value for that property.



*Default = Ordering*

## RootDirectory

This property specifies the root directory for reverse engineering. This root directory might contain all the folders that should become package during the reverse engineering process. Rhapsody builds the package hierarchy according to the folder tree from the specified path.

*Default = empty string*

## UseCalculatedRootDirectory

This property controls the use of the `<lang>_ReverseEngineering::Implementation::RootDirectory` property.

The possible values are:

- Never - Do not calculate the root directory.
- Always - Calculate the root directory and override the RootDirectory property.
- Auto - Ask the user if they want to override the value in the RootDirectory property if it is different from the calculated root directory. If the RootDirectory property is empty, Rational Rhapsody uses the calculated value without asking. This is the default value.

*Default = Auto*

## UsePackageForExternals

When Rational Rhapsody generates code, it does not regenerate code for elements that have been brought in as "external" elements. By default, the reverse engineering feature puts all external elements in a separate package in the model. You can change this behavior by changing the value of the UsePackageForExternals property. When a separate package is used, the name of the package is taken from the value of the PackageForExternals property.

*Default = Checked*

## UserDataTypes

The UserDataTypes specifies classes to be modeled as data types. This property corresponds to types entered in the Add Type window.

*Default = empty string*

## UserPackage

When reverse engineering files, Rational Rhapsody allows you the option of having packages created for each subdirectory or having all of the reverse-engineered elements placed in a single package. This option

is controlled by the property `C_ReverseEngineering::ImplementationTrait::MapToPackage`.

When `MapToPackage` is set to "User", you can use the `UserPackage` property to provide the name that you would like Rational Rhapsody to use for the single package that contains all of the reverse-engineered elements.

You can specify a nested package by using the following syntax: `package1::package2`

If the model already contains a package with the specified name, the reverse-engineered elements are put in that package. If not, Rational Rhapsody creates the package.

This property corresponds to the text field provided for the package name in the Map to Package section of the Mapping tab in the Reverse Engineering Options window.

*Default = ReverseEngineering*

## VisualizationUpdate

The `VisualizationUpdate` property instructs Rational Rhapsody to use the code-centric approach during reverse engineering and roundtripping. This approach assumes that the code serves as the blueprint for the software, and that the visual modeling capabilities of Rational Rhapsody are being used primarily to visualize the code.

In general, in code-centric mode, Rhapsody allows more drastic code changes to be brought into the model, relative to the changes that are imported when using the model-centric mode.

*Default = Checked (in code-centric settings)*

## Main

The metaclass `Main` contains properties that define the file extensions used for filtering files in the reverse engineering file selection window, as well as properties that enable jumping to problematic lines of code by double-clicking messages in the Output window.

## ErrorMessageTokensFormat

When errors are encountered during reverse engineering, they are displayed in the Rational Rhapsody Output window. If you double-click the error message, you are taken to the problematic line in the relevant source file.

This ability is made possible by the values provided for the `ParseErrorMessage` and `ErrorMessageTokensFormat` properties.

The value of the `ParseErrorMessage` property is a regular expression that extracts the relevant filename and line number information from the Rational Rhapsody-generated error message. The value of the `ErrorMessageTokensFormat` property is then used to interpret the information that was extracted from the

error message.

The value of the ErrorMessageTokensFormat property consists of a comma-separated list of keyword-value pairs representing the number of tokens contained in the extracted information, which token represents the filename, and which token represents the line number.

Users should not change the value of this property.

*Default = ToTalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2*

## ImplementationExtension

The ImplementationExtension property specifies the file extensions used to filter the list of files displayed in the Add Files window of the reverse engineering tool.

*(C Default = .c)*

## MakefileExtension

This property specifies the list of file extensions that are displayed and analyzed in the Reverse Engineering window when "Makefiles" is selected.

In order to import another kind of makefile, use the property  
C\_ReverseEngineering:MakefileImport:MakefileType and the set of properties under  
C\_ReverseEngineering:MakefileUserDefined.

*Default = mak,mk,makefile,gpj*

## ParseErrorMessage

When errors are encountered during reverse engineering, they are displayed in the Rational Rhapsody Output window. If you double-click the error message, you are taken to the problematic line in the relevant source file.

This ability is made possible by the values provided for the ParseErrorMessage and ErrorMessageTokensFormat properties.

The value of the ParseErrorMessage property is a regular expression that extracts the relevant filename and line number information from the Rational Rhapsody-generated error message. The value of the ErrorMessageTokensFormat property is then used to interpret the information that was extracted from the error message.

Users should not change the value of this property.

*Default = "([a-zA-Z\_]+[:0-9a-zA-Z\_\\.\\])\*"[:][ ]\*LINE[ ]\*([0-9]+)*

## SpecificationExtension

The SpecificationExtension property is used to specify the filename extensions that should be used to filter files in the reverse engineering file selection window. This property is used in conjunction with the ImplementationExtension property.

You can specify a number of extensions. They should be entered as a comma-separated list.

*Default = h,inl*

## **useCodeCentricAbsolutePath**

For models that use code-centric mode, you can specify whether paths to the code files should be saved as absolute or relative paths. This setting is controlled by the property useCodeCentricAbsolutePath. Note that prior to release 8.1.2, all paths to the code files were saved as absolute paths.

*Default = False*

## **UseCodeCentricSettings**

The UseCodeCentricSettings property specifies whether the visualization result of running reverse engineering is in code-centric mode in Rational Rhapsody.

If this property is checked, the code-centric settings are added to the model during reverse engineering (if they do not already exist) and a dependency with the applied profile stereotype is added between the active component to the code-centric settings.

*Default = Cleared*

## **MFC**

The MFC metaclass contains a property that affects the MFC type library.

## **DataTypes**

The DataTypes property specifies classes to be modeled as MFC data types. There is only one predefined library (MFC) that contains only one class (CString). You can, however, expand this short list of classes by the addition of classes in this property or the creation of new libraries in the property files factory.prpfactory and site.prpsite.

*Default = CString*

## **MSVC60**

The MSVC60 metaclass contains properties used to control the Microsoft Visual C++ environment.

## Defined

The Defined property specifies symbols that are defined for the Microsoft Visual C++ version 6.0 (MSVC60) preprocessor. These symbols are automatically filled into the Name list of the Preprocessing tab of the Reverse Engineering Options window when you select Add > Dialect: MSVC60.

The default value is as follows:

```
__STDC__, __STDC_VERSION__, __cplusplus, __DATE__,  
__TIME__, __WIN32__, __cdecl, __cdecl, __int64=int, __stdcall,  
__export, __export, __AFX_PORTABLE, __M_IX86=500, __declspec,  
__MSC_VER=1200, __inline=inline, __far, __near, __far, __near,  
__pascal, __pascal, __asm, __finally=catch, __based,  
__inline=inline, __single_inheritance, __cdecl, __int8=int,  
__stdcall, __declspec, __int16=int, __int32=int, __try=try,  
__int64=int, __virtual_inheritance, __except=catch, __leave=catch, __fastcall, __multiple_inheritance)
```

## IncludePath

The IncludePath property specifies necessary include paths for the Microsoft Visual C++ preprocessor. It is possible to specify the path to the site installation of the compiler as part of the site.prp, thus doing it only once and not for every project.

*Default = empty string*

## Undefined

The Undefined property specifies symbols that must be undefined for the Microsoft Visual C++ preprocessor.

*Default = empty string*

## Parser

The metaclass Parser contains properties that can be used to modify the way the parser handles code during reverse engineering.

## AdditionalKeywords

The AdditionalKeywords property can be used to list non-standard keywords that might appear in the code that you reverse engineer. This allows Rational Rhapsody to parse this code correctly during reverse engineering.

The value of this property should be a comma-separated list of the additional keywords you want to include.

Note that keywords with parameters are not supported, nor are keywords that consist of more than one word.

This property corresponds to the keywords listed on the Preprocessing tab of the Reverse Engineering Options window. Note that when you add additional keywords by using the controls on the Preprocessing tab, these keywords are included in the value of the AdditionalKeywords property at the level of the active configuration.

*Default = far,near*

## Defined

The Defined property specifies symbols and macros to be defined by using #define. For example, you can enter the following to define name> as text with the appropriate intermediate character: /D name{=|#}text

*Default = empty string*

## Dialects

The Dialects property specifies which symbols are added to the Preprocessing tab of the Reverse Engineering window box when that dialect is selected. The default value is MSVC60, which is itself defined by a metaclass of the same name under subject C\_ReverseEngineering. This dialect specifies the symbols that are defined for the Microsoft Visual C++ environment. You can define your own dialect (in the site.prp file) and select it in the Dialects property. The default value for C is an empty string.

## ForceExpansionMacros

By default, Rational Rhapsody reverse engineers macros such that when the code is regenerated, the macro definition and macro calls are generated as they appeared in the original code that was reverse engineered. (This behavior can be controlled with the C\_ReverseEngineering::ImplementationTrait::MacroExpansion property.)

In some cases, you might find that you are not satisfied with the way that Rational Rhapsody imports the macro. For such situations, you can use the ForceExpansionMacros property to list specific macros that should be expanded during reverse engineering even if the value of the MacroExpansion property is set to False.

The value of this property should be a comma-separated list of the macros that you would like Rational Rhapsody to expand during reverse engineering.

*Default = Blank*

## IncludePath

The Preprocessing tab of the Reverse Engineering Options window allows you to specify an include path

(classpath for Java) for the parser to use. The `In property` `includePath` represents this path.

For the value of this property, you can enter a comma-separated list of directories. Note that you have to specify subdirectories individually.

The directories you list here is combined with the directories specified in `#include` statements in order to find the necessary files. For example, if you have `c:\d1\d2\d3\file.h`, you can enter `c:\d1\d2` as the value of this property and then use `d3\file.h` in the `#include` statement.

You should take into account that the value of this property also determines the structure of the source file directory when code is generated from the model. So, in the above example, the top-level directory created is `d3`.

*Default = Blank*

## Undefined

The Undefined property specifies symbols and macros to be undefined by using `#undef`.

*Default = empty string*

## Promotions

The metaclass Promotion contains a number of properties used to specify whether Rational Rhapsody should add various advanced modeling constructs to your model based on relationships/patterns uncovered during reverse engineering.

### EnableAttributeToRelation

The `EnableAttributeToRelation` property is used to specify whether Rational Rhapsody should add Associations to the model for attributes whose type is another class in the model.

For example, if you have two classes, A and B, and B contains an attribute of type A, Rational Rhapsody adds an Association to the model reflecting this relationship.

*Default = Checked*

### EnableFunctionToObjectBasedOperation

The `EnableFunctionToObjectBasedOperation` property specifies whether object-based promotion is enabled during reverse engineering. Object-based promotion “promotes” a global function to comply with the pattern specified in the `C_CG::Operation::PublicName` and `C_CG::Operation::ProtectedName` properties to be an operation of the class (`object_type`) defined in the `me` parameter for the function.

*Default = Cleared*

## EnableResolveIncompleteClasses

Sometimes, during reverse engineering, Rational Rhapsody is not able to find the base class for a given class. The EnableResolveIncompleteClasses property is used to specify that if Rational Rhapsody finds a class with the same name as the base class in a different location, it should assume that this class is the missing base class.

*Default = Checked*

## Update

The Update metaclass contains properties used to control various aspects of the Rational Rhapsody behavior during and after reverse engineering.

### CreateFlowcharts

Use this property to specify whether or not Rational Rhapsody should automatically create flowcharts for operations during reverse engineering of code.

Set this property before you start the reverse engineering process.

Use this property in conjunction with the FlowchartCreationCriterion, FlowchartMinLOC, FlowchartMaxLOC, FlowchartMinControlStructures and FlowchartMaxControlStructures properties so that flowcharts are created only for operations that are within a given range in terms of lines of code or in terms of the number of control structures in the operation.

*Default = Cleared*

### FlowchartActionFixedWidth

The FlowchartActionFixedWidth property can be used to control the width of action elements in flowcharts that are created automatically during reverse engineering or when you manually select the Populate Flowchart option.

When set to True, a constant width is used, regardless of the length of the code lines that are to be displayed inside.

When set to False, the width of the graphic element representing the action is expanded so that the contained lines of code are not split.

Note that when the property is set to False, the degree to which action elements will be expanded is limited by the value of the property FlowchartActionMaxWidthSize. You can modify the value of that property if necessary.

*Default = True*



## FlowchartActionMaxWidthSize

When the value of the property FlowchartActionFixedWidth is set to False, the width of the graphic element representing an action is expanded so that the contained lines of code are not split. However, the degree to which action elements will be expanded is limited by the value of the property FlowchartActionMaxWidthSize.

The value of FlowchartActionMaxWidthSize should be the maximum number of pixels you want to allow for the width of the text portion of the box.

*Default = 500*

## FlowchartCreationCriterion

If you have set the CreateFlowcharts property to have Rational Rhapsody create flowcharts during reverse engineering, you can use the FlowchartCreationCriterion property to select the criterion that should be used to decide what operations Rational Rhapsody should create flowcharts for.

The property can take the following values:

- **Control Structures** - the decision whether or not to generate a flowchart for an operation is based on the number of control structures in the operation. When this option is selected, the minimum and maximum number of control structures used to define the inclusion criterion are taken from the FlowchartMinControlStructures and FlowchartMaxControlStructures properties.
- **LOC** - the decision whether or not to generate a flowchart for an operation is based on the number of lines of code in the operation. When this option is selected, the minimum and maximum lines of code used to define the inclusion criterion are taken from the FlowchartMinLOC and FlowchartMaxLOC properties.

*Default = LOC*

## FlowchartMaxControlStructures

If you have set the CreateFlowcharts property to have Rational Rhapsody create flowcharts during reverse engineering, and you have set the value of the FlowchartCreationCriterion property to Control Structures, then you can use the FlowchartMaxControlStructures property to specify the maximum number of control structures that an operation can have, above which Rational Rhapsody does not create a flowchart for it.

*Default = 10*

## FlowchartMaxLOC

If you have set the CreateFlowcharts property to have Rational Rhapsody create flowcharts during reverse engineering, and you have set the value of the FlowchartCreationCriterion property to LOC, then you can use the FlowchartMaxLOC property to specify the maximum number of lines of code that an operation can have, above which Rational Rhapsody does not create a flowchart for it.

*Default = 100*

## FlowchartMinControlStructures

If you have set the CreateFlowcharts property to have Rational Rhapsody create flowcharts during reverse engineering, and you have set the value of the FlowchartCreationCriterion property to Control Structures, then you can use the FlowchartMinControlStructures property specify the minimum number of control structures that an operation must have in order to have Rational Rhapsody create a flowchart for it.

*Default = 2*

## FlowchartMinLOC

If you have set the CreateFlowcharts property to have Rational Rhapsody create flowcharts during reverse engineering, and you have set the value of the FlowchartCreationCriterion property to LOC, then you can use the FlowchartMinLOC property to specify the minimum number of lines of code that an operation must have in order to have Rational Rhapsody create a flowchart for it.

*Default = 10*

# C\_Roundtrip

---

The C\_Roundtrip subject contains metaclasses that contain properties that affect roundtripping.

## General

The General metaclass contains properties that control how changes to code are roundtripped in Rational Rhapsody.

### AnalyzePeripheralIncludes

When the AnalyzePeripheralIncludes property is set to True, Rhapsody uses information from files specified as #includes in the current file in order to correctly interpret #ifdef and other preprocessor directives.

*Default = True*

### CancelIfReadOnly

Set this property to Checked in order to cancel Roundtripping if there are any Read Only (Checked-In) units that are related to one of the files to be roundtripped.

By default, Roundtrip changes only Read/Write units and provides warnings in the Output window regarding any attempted changes to Read Only units.

*Default = Cleared*

### NotifyOnInvalidatedModel

The NotifyOnInvalidatedModel property is a Boolean value that determines whether a warning window is displayed during roundtrip. This warning is displayed when information might get lost because the model was changed between the last code generation and the roundtrip operation.

*(Default = Checked)*

## ParserErrors

The ParserErrors property specifies the behavior of roundtrip when a parser error is encountered. The possible values are as follows:

- Abort - Stop roundtrip whenever there is a parser error in the code. No changes is applied to the model.
- AbortOnCritical - Stop roundtrip if any critical parser errors are encountered in the code.
- AskUser - When Rational Rhapsody encounters an error, the program displays a message asking what

you want to do.

- Ignore - Continue roundtrip, ignoring any parser errors that are encountered.

*(C Default = AskUser)*

## PredefineIncludes

The PredefineIncludes property specifies the predefined include path for roundtripping.

*(Default = empty string)*

## PredefineMacros

The PredefineMacros property specifies the predefined macros for roundtripping. The C default value is as follows:

```
OMADD_SER(p\, cvrtFunc), OMADD_OMSER(theEvent\, p), OMADD_ASER(p\,size\,sizeOfP\,cvrtF),
OMADD_ARCSER(p\size), OMADD_OMUNSER(t\,p\,destrFunc), OMADD_UNSER(t\,p\,destrFunc),
RICBAD_PARAM(p), BAD_MISSING_PARAM(p), OMDefaultThread=0, NULL=0,
OMDECLARE_GUARDED, RIC_EMPTY_STRUCT, OM_INSTRUMENT_OBJECT(theClass\,
thePackage\, thePackage\, isSingleton\, serVtbl),
OM_INSTRUMENT_PACKAGE(thePackage\,thePackage\,serVtbl),
OM_INSTRUMENT_CLASS(theClass\,thePackage\,theFullPackage\,isSingleton\,serVtbl),
OM_INSTRUMENT_OBJECT_TYPE(theClass\,thePackage\,theFullPackage\,isSingleton\,serVtbl),
OM_INSTRUMENT_FILE_CLASS(theClass\,theFullClassName\,thePackage\,theFullPackage\,isSingleton\,serVtbl),
OM_INSTRUMENT_FILE_OBJECT(theClass\,theFullClassName\,thePackage\,theFullPackage\,isSingleton\,serVtbl),
OM_INSTRUMENT_INSTANCE(me\,meAsReactive\,theClass),
OM_INSTRUMENT_EVENT_NO_UNSERIALIZE(theEvent\,thePackage\,theFullPackage\,signature),
OM_INSTRUMENT_EVENT_INSTANCE(rawMe\,theEventClass),
OM_INSTRUMENT_EVENT(theEventClass\, thePackage\, thePackage\, theEventClass),
RIC_DECLARE_MEMORY_ALLOCATOR_MEMBER(CLASSNAME),
RIC_DECLARE_MEMORY_ALLOCATOR(CLASSNAME),
RIC_MEMORY_ALLOCATOR_GET(CLASSNAME),
RIC_MEMORY_ALLOCATOR_RETURN(ME\, CLASSNAME),
RIC_IMPLEMENT_MEMORY_ALLOCATOR(CLASSNAME\,INITNUM\,INCREMENTSIZE\,ISPROTECTED)
```

## ReportChanges

The ReportChanges property defines which changes are reported (and displayed) by the roundtrip operation. The possible values are as follows:

- None - No changes are displayed in the output window.
- AddRemove - Only the elements added to, or removed from, the model are displayed in the output window.
- UpdateFailures - Only unsuccessful changes to the model are displayed in the output window.
- All - All changes to the model are displayed in the output window.

*(Default = AddRemove)*

## RestrictedMode

The RestrictedMode property is a Boolean value that specifies whether restricted-mode roundtripping is enabled. This property can be modified on the configuration level. Restricted mode of full roundtrip roundtrips unusual usage of Rational Rhapsody elements, such as a class declaration in a user-defined type.

Restricted mode has more limitations, but preserves the model from unexpected changes. The additional limitations for restricted mode are as follows:

- User-defined types cannot be removed or changed on roundtrip because Rational Rhapsody code generation adds the Ignore annotation for a user-defined type declaration.
- Relations cannot be removed or changed on roundtrip.
- New classes are not added to the model.

*(Default = Cleared)*

## RoundtripNewFiles

When working in Eclipse platform integration, use this property to set Roundtrip policy for the files that the user added to the Eclipse project and are connected to the active Rational Rhapsody Eclipse Configuration.

Optional values:

- AskUser - A message is displayed and asks the user whether or not to Roundtrip the new file.
- Always - New file is Roundtripped into Rational Rhapsody model with no message.
- Never - New file is never Roundtripped into Rational Rhapsody model with no message.

*(Default = AskUser)*

## RoundtripPreprocessorDirectives

By default, the Rational Rhapsody roundtripping feature takes into account changes made to preprocessor directives. The RoundtripPreprocessorDirectives property can be used to turn off roundtripping for the following types of preprocessor directives:

- elif
- else
- endif
- error
- if
- ifdef
- ifndef
- import

- line
- pragma
- undef
- using

*Default = Checked*

## RoundtripScheme

Determines what type of changes can be roundtripped back into the model. The possible values are Basic and Advanced.

When set to Basic, only changes to the bodies of operations and actions are roundtripped into the model.

When set to Advanced, roundtripping also takes into account elements that have been added, such as attributes and operations, and can optionally take into account elements that have been modified or removed.

When set to Respect, roundtripping also takes into account the changes that are covered by the Rational Rhapsody code preserving feature, for example, the order of class members.

*Default = Respect*

## Update

The Update metaclass contains a property that controls the update process used during roundtripping.

## AcceptChanges

The AcceptChanges property is an enumerated type that specifies which changes are applied to each CG element (attribute, operation, type, class, or package).

You can apply separate properties to each type of CG element. The possible values are as follows:

- All - All the changes can be applied to the model element.
- Default—1) Rhapsody does not roundtrip deletions if the updated code results in parser errors. 2) Rhapsody does not roundtrip the deletion of classes.
- NoDelete - All the changes except deletion can be applied to the model element. This setting prevents accidental removal of operations, constructors, attributes, relations, variables, instances, and functions.
- AddOnly - Apply only the addition of an aggregate to the model element. You cannot delete or change elements.
- NoChanges - Do not apply any changes to the model element.

Note that the value of the property is propagated to all the aggregates of an element. Therefore, if a package has the NoChanges value, no elements in that package is changed.

*Default = "Default" (in code-centric settings, default value is All)*

## **MergePolicy**

The MergePolicy property determines the merge policy Rational Rhapsody uses during roundtripping when comparing the updated code to the saved model. When the value of this property is CodeDriven, Rhapsody imports certain types of code elements that it does not import when working in model-centric mode.

This property differs from the AcceptChanges property in that AcceptChanges deals with changes to model elements (adding, deleting, modifying) while MergePolicy is used as a general indication for Rational Rhapsody that the code, rather than the model, should be given precedence when it comes to merging changes.

*Default = CodeDriven (in code-centric settings)*

## **ReplaceRTFDescription**

The property ReplaceRTFDescription is used to determine whether element descriptions containing RTF are overwritten with plain text when changes to comments are roundtripped into the model.

When set to False, changes to comments are not brought into the model if the existing element description contains RTF.

When set to True, the updated comment is taken from the code and the RTF in the existing description is replaced with plain text.

Note that in code-centric mode, the updated comment is always brought into the model, regardless of the value of this property.

*Default = False*

# DDS

---

Contains properties relating to DDS models.

## Type

Contains properties relating to types in DDS models.

### Discriminator

The Discriminator property is used to specify the type of the discriminator you will be using when defining a discriminated union in a DDS model.

This property is used in conjunction with the property `DDS::Attribute::UnionCase`.

For more information, see "Defining discriminated unions in a DDS model" in the Rational Rhapsody help.

*Default = Blank*

### StringMaximumSize

If you select String as the type of a DDS element, you can use the `DDS::Type::StringMaximumSize` property to specify the maximum string length that you want to allow. The value of the property can be any positive integer. If you leave the value of the property blank, the string is an unlimited string.

For example, if you enter 16 for the value of the `StringMaximumSize` property, the code that is generated in the IDL file resembles the following code:

```
struct Person {  
  
    /// Attributes ///  
  
    string<16> name; /// attribute name  
  
};
```

*Default = Blank*

### StringType

If you select String as the type of a DDS element, it is by default a string that uses 8-bit characters.

If you want the String to be generated as a wstring in the IDL code, change the value of the `StringType`



property from Char8 to Char32.

## Attribute

Contains properties relating to attributes in DDS models.

### UnionCase

The UnionCase property is used to specify the discriminator value that should be used for an attribute when defining a discriminated union in a DDS model. The values should reflect the type you specified for the property DDS::Type::Discriminator.

Note that you can also set the value of the property UnionCase to the string "default" for the data type that you want to use as the default data type for the union.

For example, if you are using "short" as the discriminator type, you can set the value of the UnionCase property to 1 for an attribute named length\_short, 2 for an attribute named length\_long, and "default" for an attribute named length\_double.

Using these values, the following code will be generated:

```
union length switch (short) {  
  
case 1 : short length_short; /// attribute length_short  
  
case 2 : long length_long; /// attribute length_long  
  
default : double length_double; /// attribute length_double  
  
};
```

For more information, see "Defining discriminated unions in a DDS model" in the Rational Rhapsody help.

*Default = Blank*

# DeploymentDiagram

---

The DeploymentDiagram subject contains metaclasses that contain properties for controlling the deployment diagram editor.

## AutoPopulate

The AutoPopulate metaclass contains properties that can be used to control the appearance of diagrams that are drawn automatically by Rational Rhapsody.

### ArrowDirection

The ArrowDirection property is used when Rational Rhapsody automatically generates a diagram, and it determines whether the flow of connectors in the diagram runs from top to bottom or bottom to top.

There are two situations where Rational Rhapsody automatically generates diagrams:

- If you have selected the Populate Diagrams option for Reverse Engineering (for those diagrams where this feature is supported).
- If you double-click a diagram in the browser that was generated by using the Rational Rhapsody API.

*Default = Bottom-Top*

## Comment

The Comment metaclass contains properties that control the appearance of comments in deployment diagrams.

### color

*The color property specifies the default color of the border of a graphical item, such as an object box. (Default = 128,128,0)*

### FillColor

*The Fillcolor property specifies the default fill color for the object. (Default = 0,255,255)*

### line\_width

*The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action*

*state lines). (Default = 1)*

## **name\_color**

*The name\_color property specifies the default color of names of graphical items. (Default = 0,0,0)*

# **Communication\_Path**

The Communication\_Path metaclass contains properties that control the attributes of the communication path of the deployment diagram.

## **color**

*The color property specifies the default color of the border of a graphical item, such as an object box. (Default = 255,0,0)*

## **line\_style**

The line\_style property specifies the default line style for a graphical item. The possible values are as follows:

- straight\_arrows - Draw a straight line.
- rectilinear\_arrows - Draw a rectilinear lines with right-angled corners placed at appropriate locations, depending on the starting and ending points of the line.
- spline\_arrows - Draw a curved line without corners.

*Default = rectilinear\_arrows*

## **line\_width**

*The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines). (Default = 1)*

## **name\_color**

*The name\_color property specifies the default color of names of graphical items. (Default = 255,0,0)*

# **Complete**

The metaclass Complete contains properties that determine whether or not Rational Rhapsody automatically draws the relations that exist between an element added to a diagram and elements already on the diagram.

## Complete\_Relation

The Complete\_Relation property is used to specify that when an element is added to a diagram, Rational Rhapsody should automatically draw the relations that exist between the element and elements already on the diagram.

*Default = Cleared*

## ComponentInstance

The ComponentInstance metaclass contains properties that control the attributes of the component instance.

### color

*The color property specifies the default color of the border of a graphical item, such as an object box. (Default = 0,0,255)*

### line\_width

*The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines). (Default = 1)*

### name\_color

*The name\_color property specifies the default color of names of graphical items. (Default = 0,0,255)*

## ShowName

The ShowName property specifies how the name of an object should be displayed. The possible values are as follows:

- Full\_path - Show the object name with the full path.. For example, "Default::A.B."
- Relative - Show the object name with a relative path. For example, "A.B."
- Name\_only - Show only the object name without any path information. For example, "B."

*Default = Name\_only*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## Constraint

The Constraint metaclass contains properties that control the appearance of constraints in deployment diagrams.

### color

*The color property specifies the default color of the border of a graphical item, such as an object box. (Default = 128,128,0)*

### FillColor

*The Fillcolor property specifies the default fill color for the object. (Default = 0,255,255)*

### line\_width

*The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines). (Default = 1)*

### name\_color

*The name\_color property specifies the default color of names of graphical items. (Default = 0,0,0)*

## Depends

The Depends metaclass contains a property that controls the appearance of dependency relation lines in deployment diagrams.

## color

*The color property specifies the default color of the border of a graphical item, such as an object box. (Default = 0,0,255)*

## line\_style

The line\_style property specifies the default line style for a graphical item.

The possible values are:

- straight\_arrows - straight line.
- rectilinear\_arrows - rectilinear line with right-angled corners placed at appropriate locations, depending on the starting and ending points of the line
- spline\_arrows - curved line without corners
- rounded\_rectilinear\_arrows - similar to ordinary rectilinear lines but use rounded bends rather than straight ninety-degree angles

*Default = spline\_arrows*

*Default = straight\_arrows*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = None*

## line\_width

*The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines). (Default = 1)*

## name\_color

*The name\_color property specifies the default color of names of graphical items. (Default = 0,0,255)*

# DeploymentDiagramGE

The DeploymentDiagramGE metaclass contains a property that controls the fill color used in deployment diagrams.

## FillColor

The Fillcolor property specifies the default fill color for the object.

*Default = 218,218,218*

# DiagramFrame

The DiagramFrame metaclass contains properties that control the appearance of a diagram frame.

## DefaultSize

The DefaultSize property specifies default values for diagram frame dimensions.

*Default = 20, 20, 590, 500*

## ShowName

The ShowName property determines the text to display next to a graphic element in a diagram. For most elements, you can provide a name and a label. Your descriptive label can be useful in cases where the name itself might not be sufficient due to various constraints, such as the inability to use spaces if the name of the element is to appear in the code.

The possible values for this property vary for the different elements, as does the default value used. The possible values are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the

package containing a class is displayed with the class name only if the class is not positioned inside the package in the diagram.

- Specification - the content of the specification field; relevant for elements such as constraints

Note that for some elements, such as destruction elements in sequence diagrams and timing diagrams, the property is a boolean property that hides/shows the name of the element.

*Default = Name*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## Flow

The Flow metaclass contains properties that control how information flows are displayed in deployment diagrams.

### color

*The color property specifies the default color of the border of a graphical item, such as an object box. (Default = 0,147,0)*

### flowKeyword

The flowKeyword property is a Boolean value that specifies whether the flow keyword for the information flow is displayed in the diagram.

*Default = Checked*

### infoItemsColor

*The infoItemsColor property specifies the color used to draw information items in diagrams. (Default = 0,0,255)*

### line\_style



The `line_style` property specifies the default line style for a graphical item.

The possible values are:

- `straight_arrows` - straight line.
- `rectilinear_arrows` - rectilinear line with right-angled corners placed at appropriate locations, depending on the starting and ending points of the line
- `spline_arrows` - curved line without corners
- `rounded_rectilinear_arrows` - similar to ordinary rectilinear lines but use rounded bends rather than straight ninety-degree angles

*Default = `spline_arrows`*

*Default = `rectilinear_arrows`*

## **line\_width**

*The `line_width` property specifies the default line width, in pixels, for drawing lines (for example, action state lines). (Default = 1)*

## **General**

Contains properties relating to the general appearance of deployment diagrams.

### **SysMLDiagramKindTitle**

When a diagram is created in SysML projects, the diagram frame contains a header for the diagram that includes information such as the name of the package that contains the diagram. The property `SysMLDiagramKindTitle` is used to specify a prefix for the diagram frame header that reflects the type of diagram. For example, when you create an internal block diagram, the header in the diagram frame starts with the prefix "ibd".

Note that the structure of the entire header is controlled by the property `General::Graphics::DiagramHeader`.

## **NodeProcessor**

The `NodeProcessor` metaclass contains properties that control the attributes of the component instance.

### **color**

*The color property specifies the default color of the border of a graphical item, such as an object box. (Default = 0,128,0)*

## **line\_width**

*The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines). (Default = 1)*

## **name\_color**

*The name\_color property specifies the default color of names of graphical items. (Default = 0,128,0)*

## **ShowName**

The ShowName property specifies how the name of an object should be displayed. The possible values are as follows:

- Full\_path - Show the object name with the full path.. For example, "Default::A.B."
- Relative - Show the object name with a relative path. For example, "A.B."
- Name\_only - Show only the object name without any path information. For example, "B."

*Default = Name\_only*

## **ShowStereotype**

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## **Note**

The Note metaclass contains properties that control the appearance of notes in deployment diagrams.

## **color**

*The color property specifies the default color of the border of a graphical item, such as an object box. (Default = 0,128,64)*

## Fillcolor

*The Fillcolor property specifies the default fill color for the object. (Default = 0,255,255)*

## line\_width

*The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines). (Default = 1)*

## name\_color

*The name\_color property specifies the default color of names of graphical items. (Default = 0,128,255)*

# Requirement

The Requirement metaclass contains properties that control the appearance of requirements in deployment diagrams.

## color

*The color property specifies the default color of the border of a graphical item, such as an object box. (Default = 128,128,0)*

## Fillcolor

*The Fillcolor property specifies the default fill color for the object. (Default = 0,255,255)*

## line\_width

*The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines). (Default = 1)*

## name\_color

*The name\_color property specifies the default color of names of graphical items. (Default = 0,0,0)*

# DiagramPrintSettings

---

The DiagramPrintSettings subject contains properties that affect how diagrams are printed.

## General

The General metaclass contains properties that control the default print settings for diagrams.

### Footer

The Footer property specifies a string footer that is added to the bottom of the page for printed diagrams.

*(Default = Page \$PgNum of \$Pages)*

### Header

The Header property specifies a string header that is added to the top of the page for printed diagrams.

*(Default = \$Name)*

### Orientation

The Orientation property specifies whether to print the diagram in portrait or landscape mode.

*(Default = Portrait)*

### PrintBackground

The PrintBackground property specifies whether to print the background color for the diagram.

*(Default = Cleared)*

### Scale

The Scale property specifies the default scaling factor to use when printing diagrams.

*(Default = 100)*

### ShrinkToFitOnPage

The `ShrinkToFitOnPage` property specifies whether to scale the diagram as necessary so the entire diagram fits on a single page.

*(Default = Checked)*

# Dialog

---

The Dialog subject contains metaclasses that contain properties that affect which properties are displayed on the Properties tab.

## All

Contains properties that affect the behavior of the Rational Rhapsody GUI when you select the View All property filter.

## PropertiesPerspectives

The PropertiesPerspectives property is used to define "property perspectives" - sets of related properties that you group into subcategories in order to make them more accessible in an element's Features window.

The XML structure described below makes it possible to define multiple perspectives - each with its own set of "pages" that contain related properties. See, for example, the property perspective defined for the MicroC profile.

```
<Perspectives>
  <Perspective>Perspective1
    <Pages>
      <Page>Page1
        <Properties>
          <Property>SubjectName::MetaclassName::Property1</Property>
          <Property>SubjectName::MetaclassName::Property2</Property>
          <Property>SubjectName::MetaclassName::Property3</Property>
          <Property>SubjectName::MetaclassName::Property4</Property>
        </Properties>
      </Page>
    </Pages>
  </Perspective>
</Perspectives>
```

By default, the property includes the text `<AdditionalProfilesPerspectives>`. This instructs Rhapsody to check which profiles are loaded in the model and to check to see if profile-specific property perspectives were defined.

Profile-specific property perspectives are defined by adding a property called `Dialog::All::[profile_name]PropertiesPerspectives`, for example, `Dialog::All::AUTOSAR_42PropertiesPerspectives` for the property perspectives to use with models that contain the profile `AUTOSAR_42`. When adding such a property to define property perspectives for a profile, the XML in the value of the profile-specific property should start with the `<Perspective>` tag. It should not contain the `<Perspectives>` tag.

*Default = `?<IsConditionalProperty><Perspectives>$<AdditionalProfilesPerspectives></Perspectives>`*

## UserDefinedSubjects

The `UserDefinedSubjects` property allows you to enter a comma-separated list of subjects that you always want to be visible when the View All property filter is selected, regardless of the context.

*Default = Blank*

## MicroCPropertiesPerspectives

Defines the perspectives, the pages for each perspective, and the properties.

Use the property: `Dialog::All::PropertiesPerspectives` to define new perspectives, the pages for each perspective, and the properties to appear on each page.

The following represents the generic XML tagging for adding a new perspective, pages and properties:  
`<Perspectives> <Perspective>Perspective1 <Pages> <Page> Page1 <Properties>  
<Property>Subject::Metaclass::Property1</Property>  
<Property>Subject::Metaclass::Property2</Property> ... </Properties> </Page> <Page>Page2  
<Properties> <Property>Subject::Metaclass::Property3</Property> ... .... </Perspective> </Perspectives>`

Note: The properties appear in the perspective only if they exist.

You can refer to a perspective as if it is a predefined filter.

## Attribute

The `Attribute` metaclass contains properties that control which subjects and metaclasses are displayed for attributes when you use the Common filter for the properties.

## CommonProperties

*The `CommonProperties` property specifies which properties are displayed when you select the Common*

*filter for the properties. (Default = CG::Attribute::Animate, WebComponents::Attribute::WebManaged)*

## **C\_CommonProperties**

*The C\_CommonProperties property specifies which Rational Rhapsody Developer for C-specific properties are displayed when you select the Common filter for the properties. (Default = C\_CG::Attribute::AccessorGenerate, C\_CG::Attribute::MutatorGenerate, C\_CG::Attribute::VariableInitializationFile)*

## **CPP\_CommonProperties**

*The CPP\_CommonProperties property specifies which Rational Rhapsody Developer for C++-specific properties are displayed when you select the Common filter for the properties. (Default = CPP\_CG::Attribute::AccessorGenerate, CPP\_CG::Attribute::MutatorGenerate, CPP\_CG::Attribute::ReferenceImplementationPattern)*

## **JAVA\_CommonProperties**

*The JAVA\_CommonProperties property specifies which Rational Rhapsody Developer for Java-specific properties are displayed when you select the Common filter for the properties. (Default = JAVA\_CG::Attribute::AccessorGenerate, JAVA\_CG::Attribute::MutatorGenerate)*

# **Class**

The Class metaclass contains properties that control which subjects and metaclasses are displayed for classes when you use the Common filter for the properties.

## **CommonProperties**

*The CommonProperties property specifies which properties are displayed when you select the Common filter for the properties. (Default = CG::Class::UseAsExternal, CG::Class::Animate, CG::Class::ActiveThreadName, CG::Class::ActiveThreadPriority, CG::Class::ActiveStackSize, CG::Class::ActiveMessageQueueSize, WebComponents::Class::WebManaged)*

## **Ada\_CommonProperties**

*The Ada\_CommonProperties property specifies which Rational Rhapsody Developer for Ada-specific properties are displayed when you select the Common filter for the properties. (Default = Ada\_CG::Class::Visibility, Ada\_CG::Class::TaskBody)*

## **C\_CommonProperties**



*The C\_CommonProperties property specifies which Rational Rhapsody Developer for C-specific properties are displayed when you select the Common filter for the properties. (Default = C\_CG::Class::EnableDynamicAllocation, C\_CG::Class::EnableUseFromCPP, C\_CG::Class::GenerateDestructor, C\_CG::Class::ImpIncludes, C\_CG::Class::SpecIncludes, C\_CG::Class::ObjectTypeAsSingleton)*

## **CPP\_CommonProperties**

*The CPP\_CommonProperties property specifies which Rational Rhapsody Developer for C++-specific properties are displayed when you select the Common filter for the properties. (Default = CPP\_CG::Class::DeclarationModifier, CPP\_CG::Class::Embeddable, CPP\_CG::Class::ImpIncludes, CPP\_CG::Class::IsReactiveInterface, CPP\_CG::Class::SpecIncludes)*

## **ShowInheritedAttributes**

Select this property to display the Show Inherited check box on the Attributes tab of the Features window.

*(Default = Cleared)*

## **ShowInheritedFlowPorts**

Select this property to display the Show Inherited check box on the Flow Ports tab of the Features window.

*(Default = Cleared)*

## **ShowInheritedFlowProperties**

This property filters the list of inherited Flow properties on the Flow Properties tab of the Features window. When this property is cleared, the list of inheritance is shown for the selected class only. When this property is checked, the list includes inheritance from the base class.

*(Default = Cleared)*

## **ShowInheritedOperations**

Select this property to display the Show Inherited check box on the Operations tab of the Features window.

*(Default = Cleared)*

## **ShowInheritedPorts**

Select this property to display the Show Inherited check box on the Ports tab of the Features window.

(Default = Cleared)

## ClassifierRole

The ClassifierRole metaclass contains properties that control which subjects and metaclasses are displayed for classifier roles when you use the Common filter for the properties.

### CommonProperties

*The CommonProperties property specifies which properties are displayed when you select the Common filter for the properties. (Default = Animation::ClassifierRole::MappingPolicy, Animation::ClassifierRole::DisplayMessagesToSelf)*

## Component

The Component metaclass contains a property that controls which subjects and metaclasses are displayed for components when you use the Common filter for the properties.

### Ada\_CommonProperties

*The Ada\_CommonProperties property specifies which Rational Rhapsody Developer for Ada-specific properties are displayed when you select the Common filter for the properties. (Default = Ada\_CG::Component::AdaVersion)*

## Configuration

The Configuration metaclass contains properties that control which subjects and metaclasses are displayed for configurations when you use the Common filter for the properties.

### CommonProperties

*The CommonProperties property specifies which properties are displayed when you select the Common filter for the properties. (Default = CG::CGGeneral::GeneratedCodeInBrowser)*

### C\_CommonProperties

*The C\_CommonProperties property specifies which Rational Rhapsody Developer for C-specific*

*properties are displayed when you select the Common filter for the properties. (Default = C\_CG::Configuration::ClassStateDeclaration, C\_CG::Configuration::DefaultImplementationDirectory, C\_CG::Configuration::DefaultSpecificationDirectory, C\_CG::Configuration::InitializeEmbeddableObjectsByValue)*

## CPP\_CommonProperties

*The CPP\_CommonProperties property specifies which Rational Rhapsody Developer for C++-specific properties are displayed when you select the Common filter for the properties. (Default = CPP\_CG::Configuration::ContainerSet, CPP\_CG::Configuration::DefaultImplementationDirectory, CPP\_CG::Configuration::DefaultSpecificationDirectory, CPP\_CG::Configuration::InitializeEmbeddableObjectsByValue)*

## Dependency

The Dependency metaclass contains properties that control which subjects and metaclasses are displayed for dependencies when you use the Common filter for the properties.

## CommonProperties

*The CommonProperties property specifies which properties are displayed when you select the Common filter for the properties. (Default = CG::Dependency::UsageType)*

## Ada\_CommonProperties

*The Ada\_CommonProperties property specifies which Rational Rhapsody Developer for Ada-specific properties are displayed when you select the Common filter for the properties. (Default = Ada\_CG::Dependency::CreateUseStatement)*

## CPP\_CommonProperties

*The CPP\_CommonProperties property specifies which Rational Rhapsody Developer for C++-specific properties are displayed when you select the Common filter for the properties. (Default = CPP\_CG::Dependency::UseNamespace)*

## Diagrams

The Diagrams metaclass contains a property that controls which subjects and metaclasses are displayed for diagrams when you use the Common filter for the properties.

## CommonProperties

*The CommonProperties property specifies which properties are displayed when you select the Common filter for the properties. (Default = General::Graphics::ShowLabels, General::Graphics::MarkMisplacedElements, ComponentDiagram::ComponentDiagramGE::FillColor, DeploymentDiagram*

## Event

The Event metaclass contains properties that control which subjects and metaclasses are displayed for events when you use the Common filter for the properties.

## CommonProperties

*The CommonProperties property specifies which properties are displayed when you select the Common filter for the properties. (Default = CG::Event::Animate, CG::Event::DeleteAfterConsumption, WebComponents::Event::WebManaged)*

## C\_CommonProperties

*The C\_CommonProperties property specifies which Rational Rhapsody Developer for C-specific properties are displayed when you select the Common filter for the properties. (Default = C\_CG::Event::EnableDynamicAllocation)*

## File

The File metaclass contains properties that control which subjects and metaclasses are displayed for files when you use the Common filter for the properties.

## CommonProperties

*The CommonProperties property specifies which properties are displayed when you select the Common filter for the properties. (Default = WebComponents::File::WebManaged)*

## C\_CommonProperties

*The C\_CommonProperties property specifies which Rational Rhapsody Developer for C-specific properties are displayed when you select the Common filter for the properties. (Default = C\_CG::File::ImplementationHeader, C\_CG::File::SpecificationHeader)*

## ConvertSelectedPathToRelative

The property controls whether the path of a component file, or a component folder, is converted to a relative path or remains as it was entered by the user, either by using the "..." dialog, and/or by editing the text box.

A relative path is entered when all of the following hold: \* The property value is True. \* You enter an absolute path for a specific component file, or a specific component folder. \* The entered path is a sub-path of the parent container path.

As a result, the path is converted to be relative to the parent container of the specific component file or folder. For example, "L:/myProject/myConfig/X/Y" becomes "X/Y".

## CPP\_CommonProperties

*The CPP\_CommonProperties property specifies which Rational Rhapsody Developer for C++-specific properties are displayed when you select the Common filter for the properties. (Default = CPP\_CG::File::ImplementationHeader, CPP\_CG::File::SpecificationHeader)*

## JAVA\_CommonProperties

*The JAVA\_CommonProperties property specifies which Rational Rhapsody Developer for Java-specific properties are displayed when you select the Common filter for the properties. (Default = JAVA\_CG::File::Header)*

## General

The General metaclass contains a property that controls which filter is applied to the list of properties.

## ModelElement

The ModelElement metaclass contains properties related to the display of model elements in the Features window.

## ExportPropertiesToCSVContentLine

From the Properties tab of the Features window, you can export the property information for the selected model element to a csv file. You can use the property ExportPropertiesToCSVContentLine to control what

information is exported for each property. You can use the following keywords in this property:

- `$<Subject>` - the Subject that the property is under
- `$<MetaClass>` - the Metaclass that the property is under
- `$<PropertyName>` - the name of the property
- `$<PropertyValue>` - the value of the property
- `$<PropertyOrigin>` - the origin of the property, for example, factory.prp or locally overridden
- `$<OriginFullPathName>` - for overridden properties, the full path to the element where the value is overridden
- `$<TabName>` - for properties organized in a "property perspective", this keyword will give the name of the tab that contains the property
- `$<View>` - for properties organized in a "property perspective", this keyword will give the selected view for the tab that contains the property, for example "Overridden"

*Default = ?<IsConditionalProperty>\$<Subject>,  
\$<MetaClass>,\$<PropertyName>,\$<PropertyValue>,\$<PropertyOrigin>,\$<OriginFullPathName>?<begin>\$<IsPerspective>?<end>*

## ExportPropertiesToCSVHeaderLine

From the Properties tab of the Features window, you can export the property information for the selected model element to a csv file. You can use the property `ExportPropertiesToCSVHeaderLine` to customize the headings used for the columns of property data.

*Default = ?<IsConditionalProperty>Subject,Meta Class,Property Name,Property Value,Property Origin,Origin Full Path Name?<begin>\$<IsPerspective>?<==>True?<?>,Tab Name,View?<:>?<end>*

## ExportPropertiesToCSVSummaryLine

From the Properties tab of the Features window, you can export the property information for the selected model element to a csv file. In addition to the property data and the headings used for the data, the generated csv file includes a summary, which by default shows the name of the element selected in the browser and the the selected view for the Properties tab (for example, "Overridden"). You can use the property `ExportPropertiesToCSVSummaryLine` to customize this summary. You can use the following keywords in this property:

- `$<ElementName>` - the name of the element selected in the browser
- `$<View>` - the view selected for the Properties tab

*Default = ?<IsConditionalProperty>Element Name: \$<ElementName>, View: \$<View>*

## TagsAutoArrange

The `TagsAutoArrange` property corresponds to the "Use default order" check box on the Tags tab of the Properties window.

When the value of the property is set to True, tags are sorted alphabetically.

If you set the value of the property to False, the tags are displayed in the order that they were created, or according to the user-specified order if the original order was changed by using the ordering feature provided in the browser ("Enable Ordering" under View > Browser Display Options).

*Default = True*

## ObjectModelDiagram

The ObjectModelDiagram metaclass contains a property that controls which subjects and metaclasses are displayed for OMDs when you use the Common filter for the properties.

### CommonProperties

*The CommonProperties property specifies which properties are displayed when you select the Common filter for the properties. (Default = General::Graphics::ShowLabels, ObjectModelGe::Class::ShowAttributes, ObjectModelGe::Class::ShowOperations, ObjectModelGe::Class::ShowName, ObjectModelGe::Class::ShowStereotype, ObjectModelGe::Complete::Complete\_Relation, ObjectModelGe::Package::ShowName)*

## Operation

The Operation metaclass contains properties that control which subjects and metaclasses are displayed for operations when you use the Common filter for the properties.

### CommonProperties

*The CommonProperties property specifies which properties are displayed when you select the Common filter for the properties. (Default = CG::Operation::Animate, CG::Operation::Concurrency, CG::Operation::VariableLengthArgumentList, WebComponents::Operation::WebManaged)*

### Ada\_CommonProperties

*The Ada\_CommonProperties property specifies which Rational Rhapsody Developer for Ada-specific properties are displayed when you select the Common filter for the properties. (Default = Ada\_CG::Operation::EntryCondition, Ada\_CG::Operation::LocalVariablesDeclaration, Ada\_CG::Operation::Renames)*

### CPP\_CommonProperties

*The CPP\_CommonProperties property specifies which Rational Rhapsody Developer for C++-specific properties are displayed when you select the Common filter for the properties. (Default =*

*CPP\_CG::Operation::ImplementationEpilog, CPP\_CG::Operation::ImplementationProlog,  
CPP\_CG::Operation::SpecificationEpilog, CPP\_CG::Operation::SpecificationProlog,  
CPP\_CG::Operation::ThrowExceptions)*

## **JAVA\_CommonProperties**

*The JAVA\_CommonProperties property specifies which Rational Rhapsody Developer for Java-specific properties are displayed when you select the Common filter for the properties. (Default = JAVA\_CG::Operation::IsNative, JAVA\_CG::Operation::ThrowExceptions)*

## **UseReturnTypeFromCG**

The UseReturnTypeFromCG property specifies whether the signature field on the General tab of the features window for operations should display the actual return type that is generated during code generation.

*Default = Cleared*

## **Package**

The Package metaclass contains properties that control which subjects and metaclasses are displayed for packages when you use the Common filter for the properties.

## **CommonProperties**

*The CommonProperties property specifies which properties are displayed when you select the Common filter for the properties. (Default = General::Graphics::ShowLabels, SequenceDiagram::General::RealizeMessages, SequenceDiagram::General::ShowSequenceNumbers, SequenceDiagram::General::ClassCentricMode, SequenceDiagram::General::ShowArguments, ObjectModelGe::Class::ShowAttributes, ObjectModelGe::Class::ShowOperations, ObjectModelGe::Class::ShowName)*

## **C\_CommonProperties**

*The C\_CommonProperties property specifies which Rational Rhapsody Developer for C-specific properties are displayed when you select the Common filter for the properties. (Default = C\_CG::Package::ImpIncludes, C\_CG::Package::SpecIncludes)*

## **CPP\_CommonProperties**

*The CPP\_CommonProperties property specifies which Rational Rhapsody Developer for C++-specific properties are displayed when you select the Common filter for the properties. (Default = CPP\_CG::Package::Animate, CPP\_CG::Package::SpecIncludes, CPP\_CG::Package::ImpIncludes,*



*CPP\_CG::Package::DefineNameSpace, CPP\_CG::Package::ImpIncludes)*

## Project

The Project metaclass contains properties that control which subjects and metaclasses are displayed for projects when you use the Common filter for the properties.

### CommonProperties

*The CommonProperties property specifies which properties are displayed when you select the Common filter for the properties. (Default = General::Graphics::MaintainWindowContent, General::Graphics::grid\_display, General::Graphics::grid\_snap, SequenceDiagram::General::RealizeMessages, SequenceDiagram::General::ShowSequenceNumbers, SequenceDiagram::General::CleanupRealized, SequenceDiagram::General::ClassCentricMode, SequenceDiagram::General::ShowArguments, ObjectModelGe::Class::ShowAttributes, ObjectModelGe::Class::ShowOperations, ObjectModelGe::Class::ShowName, ConfigurationManagement::General::CMTool, ConfigurationManagement::General::UseSCCtool, RTInterface::DOORS::InstallationDir, RTInterface::DOORS::LmLicenseFile, CG::CGGeneral::GeneratedCodeInBrowser)*

### C\_CommonProperties

*The C\_CommonProperties property specifies which Rational Rhapsody Developer for C-specific properties are displayed when you select the Common filter for the properties. (Default = C\_CG::Configuration::InitializeEmbeddableObjectsByValue, C\_CG::Attribute::AccessorGenerate, C\_CG::Attribute::MutatorGenerate)*

### CPP\_CommonProperties

*The CPP\_CommonProperties property specifies which Rational Rhapsody Developer for C++-specific properties are displayed when you select the Common filter for the properties. (Default = CPP\_CG::Attribute::AccessorGenerate, CPP\_CG::Attribute::MutatorGenerate, CPP\_CG::Configuration::InitializeEmbeddableObjectsByValue)*

### JAVA\_CommonProperties

*The JAVA\_CommonProperties property specifies which Rational Rhapsody Developer for Java-specific properties are displayed when you select the Common filter for the properties. (Default = JAVA\_CG::Attribute::AccessorGenerate, JAVA\_CG::Attribute::MutatorGenerate)*

## Relation

The Relation metaclass contains properties that control which subjects and metaclasses are displayed for relations when you use the Common filter for the properties.

## CommonProperties

*The CommonProperties property specifies which properties are displayed when you select the Common filter for the properties. (Default = CG::Relation::Ordered)*

## C\_CommonProperties

*The C\_CommonProperties property specifies which Rational Rhapsody Developer for C-specific properties are displayed when you select the Common filter for the properties. (Default = C\_CG::Relation::ImplementWithStaticArray)*

## CPP\_CommonProperties

*The CPP\_CommonProperties property specifies which Rational Rhapsody Developer for C++-specific properties are displayed when you select the Common filter for the properties. (Default = CPP\_CG::Relation::ImplementWithStaticArray, CPP\_CG::Relation::Static)*

## JAVA\_CommonProperties

*The JAVA\_CommonProperties property specifies which Rational Rhapsody Developer for Java-specific properties are displayed when you select the Common filter for the properties. (Default = JAVA\_CG::Relation::Static)*

# SequenceDiagram

The SequenceDiagram metaclass contains a property that controls which subjects and metaclasses are displayed for sequence diagrams when you use the Common filter for the properties.

## CommonProperties

*The CommonProperties property specifies which properties are displayed when you select the Common filter for the properties. (Default = General::Graphics::ShowLabels, SequenceDiagram::General::RealizeMessages, SequenceDiagram::General::ShowSequenceNumbers, SequenceDiagram::General::ShowArguments, Animation::ClassifierRole::MappingPolicy, Animation::ClassifierRole::DisplaysMessagesToSelf)*

# Stereotype

The Stereotype metaclass contains a property that controls which subjects and metaclasses are displayed for stereotypes when you use the Common filter for the properties.

## CommonProperties

*The CommonProperties property specifies which properties are displayed when you select the Common filter for the properties. (Default = Model::Stereotype::BrowserIcon, Model::Stereotype::BrowserGroupIcon, Model::Stereotype::DrawingShape, Model::Stereotype::DrawingText)*

## TimingDiagram

Contains properties relating to the information that is displayed in the Features dialog for timing diagrams.

## CommonProperties

One of the filters provided on the Properties tab of the Features dialog is "Common". Use the property CommonProperties to specify the properties that should be displayed when this filter is selected in the Features dialog for timing diagrams.

*Default =*

*TimingDiagram::TimeAxis::StartValue, TimingDiagram::TimeAxis::IncrementValue, TimingDiagram::TimeAxis::UnitLabel*

## Type

The Type metaclass contains properties that control which subjects and metaclasses are displayed for user-defined types when you use the Common filter for the properties.

## Ada\_CommonProperties

*The Ada\_CommonProperties property specifies which Rational Rhapsody Developer for Ada-specific properties are displayed when you select the Common filter for the properties. (Default = Ada\_CG::Type::Visibility, Ada\_CG::Type::DeclarationPosition)*

## C\_CommonProperties

*The C\_CommonProperties property specifies which Rational Rhapsody Developer for C-specific properties are displayed when you select the Common filter for the properties. (Default = C\_CG::Type::AnimSerializeOperation, C\_CG::Type::AnimUnserializeOperation)*

## CPP\_CommonProperties

*The CPP\_CommonProperties property specifies which Rational Rhapsody Developer for C++-specific properties are displayed when you select the Common filter for the properties. (Default = CPP\_CG::Type::AnimSerializeOperation, CPP\_CG::Type::AnimUnserializeOperation)*

## UseCaseDiagram

The UseCaseDiagram metaclass contains a property that controls which subjects and metaclasses are displayed for UCDs when you use the Common filter for the properties.

### CommonProperties

*The CommonProperties property specifies which properties are displayed when you select the Common filter for the properties. (Default = General::Graphics::ShowLabels, UseCaseGe::UseCase::ShowName, UseCaseGe::UseCase::ShowStereotype, UseCaseGe::Complete::Complete\_Relation, UseCaseGe::Package::ShowName)*

# Eclipse

---

The Eclipse subject contains metaclasses that contain properties that affect which properties are displayed in the Properties tab.

## Configuration

The Configuration metaclass contains properties that control the display of the Rational Rhapsody browser.

### DefaultProjectLocation

When the property `Eclipse::Project::AutomaticProjectCreation` is set to `True`, you can use the `DefaultProjectLocation` property to specify the location that should be used for the Eclipse project that is automatically created when you create an Eclipse configuration in Rhapsody.

You can use the following variables when specifying the project location:

- `$WorkspaceLocation` - the folder representing the current Eclipse workspace
- `$EclipseProjectName` - the name of the Eclipse project that was created using the format specified by the value of `Eclipse::Configuration::DefaultProjectName`.

### DefaultProjectName

When the property `Eclipse::Project::AutomaticProjectCreation` is set to `True`, you can use the `DefaultProjectName` property to specify the name format to use for the Eclipse project that is automatically created when you create an Eclipse configuration in Rhapsody.

You can use the following variables when specifying the name format: `$ProjectName`, `$ComponentName`, `$ConfigurationName` (referring to the Rhapsody project, component, and configuration).

### InvokeExecutable

The `InvokeExecutable` property (under `Eclipse::Configuration` ) points to the executable program.  
Keywords: `$executable` - the IDE executable program as read from `Rhapsody.ini`.

The possible values are as follows:

- `Always` - Rational Rhapsody displays a confirmation window each time you try to delete an item from the model.
- `Never` - Confirmation is not required to delete an element.
- `WhenNeeded` - Rational Rhapsody displays a message asking for confirmation if there are references to the element (or for some other reason).

*(Default = \$executable)*

## InvokeParameters

The InvokeParameters property (under Eclipse::Configuration ) control the parameters for the command line.

Keywords:

\$workspace as specified in the Rational Rhapsody Tags for the Eclipse Configuration.

\$RhpClientPort: the port number that Rational Rhapsody uses to be a client to Eclipse, as specified by using the Rational Rhapsody Code IDE menu command options .

*\$RhpServerPort: the port number that Rational Rhapsody uses to be a server to Eclipse (Default = -data \$workspace -vmargs -DRhpClientPort= \$RhpClientPort -DRhpServerPort= \$RhpServerPort)*

## DefaultEnvironments

A default Rational Rhapsody environment is chosen according to the type of project that the user creates in an IDE.

The following are examples of situations (that change the type of projects) that would affect the choice of the Rational Rhapsody environment:

- The user creates a new Eclipse configuration in Rational Rhapsody.
- Workbench is brought forward and the "Create new project" wizard is displayed.
- The user creates a Workbench Real Time Project in Workbench.
- Rational Rhapsody is notified that an active Eclipse configuration is coupled with a Workbench project.

## Eclipse

The Eclipse property is the default environment in the settings tab for generic Eclipse (CDT) projects.

*(Default = Cygwin)*

## Workbench

The Workbench property is the default environment in the settings tab for generic Eclipse (CDT) projects.

*(Default = WorkbenchManaged)*

## WorkbenchKernel

The WorkbenchKernel property is set if the user creates a Downloadable Kernel module project in Workbench and wants the default environment to be "WorkbenchManaged."

## WorkbenchRTP

The WorkbenchRTP property is set if the "DefaultEnvironments" for a Workbench Real Time project is mapped to Eclipse:DefaultEnvironments:WorkbenchKernel property and its default value is Workbenchmanaged\_RTP.

## Export

The metaclass Export contains properties related to the exporting of Eclipse projects to create Rational Rhapsody models.

### AssociateWithOriginalProjectOnExport

When you export an Eclipse project to Rational Rhapsody, the Export to Rational Rhapsody Model window contains an option "Associate Rational Rhapsody model with original Eclipse project", which is selected by default. This means that after the contents of the Eclipse project are brought into a Rational Rhapsody model, the resulting model is linked to the original Eclipse project, and when code is generated from the model, it is stored in the original Eclipse project.

In some cases, however, you might want to sever any connection between the original Eclipse project and the Rational Rhapsody model after the initial import into Rational Rhapsody. If you clear the "Associate Rational Rhapsody model with original Eclipse project" check box, then the original Eclipse project is only used for the initial import into Rational Rhapsody. A window is opened where you can indicate that you want the code that is generated from the Rational Rhapsody model to be stored in a new Eclipse project, or select an existing Eclipse project that you would like to use to house the code that is generated from Rhapsody.

*Default = Checked*

## Project

The Project metaclass contains properties that relate to the handling of Eclipse projects associated with Rhapsody projects.

### AllowRoundtripForAllFiles

The `AllowRoundtripForAllFiles` property is used to specify that Rhapsody should roundtrip changes made to source files that are part of Eclipse projects associated with Rhapsody projects even if the file modified does not belong to the active Rhapsody configuration.

If you set the value of this property to `False`, Rhapsody will not roundtrip changes made to source files that do not belong to the active Rhapsody configuration.

*Default = True*

## **AutomaticProjectCreation**

The `AutomaticProjectCreation` property can be used to specify that a new Eclipse project should be created automatically when you create an Eclipse configuration in Rhapsody, rather than having Rhapsody launch the Rhapsody Project wizard in Eclipse.

If you set the value of this property to `True`, use the properties `Eclipse::Configuration::DefaultProjectName` and `Eclipse::Configuration::DefaultProjectLocation` to specify the name format to use for the Eclipse project created and the location where the project should be saved.

*Default = False*



# General

---

The General subject contains metaclasses that contain properties that control the general aspects of the Rational Rhapsody display.

## Attribute

Contains properties related to the display of attribute information.

### SignaturePluginFunction

The property SignaturePluginFunction can be used to specify the name of a plugin method that returns a string which should be displayed in the Features dialog instead of the default attribute declaration displayed by Rhapsody.

The value of the property should be the name of a plugin method.

*Default = Blank*

## Graphics

The Graphics metaclass contains properties that determine the general behavior of graphic editors, such as whether you can drag a graphic element by clicking on its bounding box.

### ActivityDiagramToolbar

Use the ActivityDiagramToolbar property to set or re-order the tools on the drawing toolbar for activity diagrams.

You can set the property to:

- Show all the default drawing tools by using the "RpyDefault" value.
- Show a new tool in addition to the default drawing tools.
- Show only those drawing tools that you want to appear on the drawing toolbar.

To enter a new tool in addition to the default drawing tools:

- Create a stereotype for the new tool and associate it with an "Applicable to" element. For example, create a stereotype called "MyActor" with an "Applicable to" element set to "Actor."
- For the ActivityDiagramToolbar property, set the value to "RpyDefault,MyActor." Always use a comma to separate values for the property.

To enter only those drawing tools that you want to appear on the drawing toolbar, enter the drawing tool names separated by a comma. For example, if you want only the Swimlane Frame, Swimlane Divider, Initial Flow, ControlFlow, and ActivityFinal tools on the drawing toolbar, enter "Swimlane Frame,SwimlaneDivider,InitialFlow,ControlFlow,ActivityFinal" as the value for the property.

For the current list of tool names for a drawing toolbar, set the property to "RpyDefault", which is the default value. With the property set to the default value, all the available drawing tools with names appear on its drawing toolbar.

Note that this property only affects the drawing toolbar for activity diagrams. It does not affect the drawing toolbar for "new term" stereotypes that are applicable to activity diagrams.

*Default = RpyDefault*

## **AdditionalCompartments**

Various types of graphic elements in diagrams, such as classes and objects, have compartments that can display elements that belong to that element, for example, attributes and operations of a class. There are a number of predefined compartments that you can choose to show or hide. In addition, you can use the property AdditionalCompartments to define additional compartments to display, for any "new terms" in your project. The value of the property should be a comma-separated list of the names of the "new terms" that you want to have compartments for.

*Default = Blank*

## **AllowObjectReparenting**

The AllowObjectReparenting property controls whether moving an object on a diagram into another element (for example, a class), the element gets to be the owner of this object (moving it in the browser). This property takes effect only when the UpdateModelFromGraphicChange property is set to TRUE.

*Default = Cleared*

## **AllowRequirementReparenting**

The AllowRequirementReparenting property controls whether moving a requirement on a diagram into another element (for example, a class), the element gets to be the owner of this requirement (moving it in the browser). This property takes effect only when the UpdateModelFromGraphicChange property is set to TRUE.

*Default = Cleared*

## **AutoScrollMargin**

The AutoScrollMargin controls how responsive the auto scrolling functionality is. The auto scroll begins scrolling when the mouse pointer enters the auto scroll margins, which are virtual margins that define a virtual region around the drawing area (starting from the window frame and going X number of points into the drawing area).

The AutoScrollMargin property defines the X number of points the margins enter into the drawing area. If you specify a large number for this property, the margin becomes bigger, thereby making the auto scroll more sensitive. Set this property to 0 to disable auto scrolling.

*(Default = 50)*

## AutoScrollOnSelecting

Rational Rhapsody includes an autoscroll feature that runs when you are selecting objects within a given region and you approach the edge of the diagram window. This allows you to extend your selection to include objects that are currently outside the viewable area of the window.

The AutoScrollOnSelecting property can be used to disable or enable this feature.

You can use the AutoScrollMargin property to control when the autoscrolling runs by changing the size of the region considered to be the edge of the viewable area.

*Default = Checked*

## ClassBoxFont

*The ClassBoxFont property specifies the default font for new class attributes and operations. To change the font used for the class itself, use the Format window. (Default = Arial 10 NoBold NoItalic)*

## CompartmentsTitleFont

When classes are displayed in Specification view, compartments are displayed for elements such as attributes and operations.

If you have used the Display Options window or the ShowCompartmentsTitle property to specify that headings should be displayed to identify the different compartments, you can use the CompartmentsTitleFont property to choose the font that Rational Rhapsody should use for these compartment headings.

When you click the ".." button next to the property value, a font chooser window is displayed.

*Default = Arial 14 NoBold Italic*

## CommunicationDiagramToolbar

Use the CommunicationDiagramToolbar property to set or re-order the tools on the drawing toolbar for communication diagrams.

You can set the property to:

- Show all the default drawing tools by using the "RpyDefault" value.
- Show a new tool in addition to the default drawing tools.

- Show only those drawing tools that you want to appear on the drawing toolbar.

To enter a new tool in addition to the default drawing tools:

- Create a stereotype for the new tool and associate it with an "Applicable to" element. For example, create a stereotype called "MyActor" with an "Applicable to" element set to "Actor."
- For the CommunicationDiagramToolbar property, set the value to "RpyDefault,MyActor." Always use a comma to separate values for the property.

To enter only those drawing tools that you want to appear on the drawing toolbar, enter the drawing tool names separated by a comma. For example, if you want only the Object, Actor and Link tools on the drawing toolbar, enter "Object,Actor,Link" as the value for the property.

For the current list of tool names for a drawing toolbar, set the property to "RpyDefault", which is the default value. With the property set to the default value, all the available drawing tools with names appear on its drawing toolbar.

Note that this property only affects the drawing toolbar for communication diagrams. It does not affect the drawing toolbar for "new term" stereotypes that are applicable to communication diagrams.

*Default = RpyDefault*

## ComponentDiagramToolbar

Use the ComponentDiagramToolbar property to set or re-order the tools on the drawing toolbar for component diagrams.

You can set the property to:

- Show all the default drawing tools by using the "RpyDefault" value.
- Show a new tool in addition to the default drawing tools.
- Show only those drawing tools that you want to appear on the drawing toolbar.

To enter a new tool in addition to the default drawing tools:

- Create a stereotype for the new tool and associate it with an "Applicable to" element. For example, create a stereotype called "MyComponent" with an "Applicable to" element set to "Component".
- For the ComponentDiagramToolbar property, set the value to "RpyDefault,MyComponent". Always use a comma to separate values for the property.

To enter only those drawing tools that you want to appear on the drawing toolbar, enter the drawing tool names separated by a comma. For example, if you want only the Component, File, and Dependency tools on the drawing toolbar, enter "Component,File,Dependency" as the value for the property.

For the current list of tool names for a drawing toolbar, set the property to "RpyDefault", which is the default value. With the property set to the default value, all the available drawing tools with names appear on its drawing toolbar.

Note that this property only affects the drawing toolbar for component diagrams. It does not affect the drawing toolbar for "new term" stereotypes that are applicable to component diagrams.

*Default = RpyDefault*

## ConnectorHighlighting

The property ConnectorHighlighting can be used to specify what graphic elements should be highlighted when you hover over a connector. The possible values for this property are:

- ConnectorOnly - only the connector is highlighted
- ConnectorAndNodes - the connector and the nodes on either side of the connector are highlighted

*Default = ConnectorAndNodes*

## ContextByTopology

This property applies to block definition diagrams. When you create new elements in the Rational Rhapsody browser, the property allows you to view the metaclasses of those elements you have added as you drag and drop them onto the diagram. The context of each is set automatically according to their graphic topology.

*Default = Cleared*

## CreateRequirementUnderDiagram

This property affects all diagrams (except statechart and activity diagrams that are always set to use this approach).

If set to Checked, the common elements created by a drag-and-drop operation from the toolbar are created under the diagram and not in the default package of the diagram.

*Default = Cleared*

## CRTerminator

The CRTerminator property specifies how multiline fields in notes and statechart names should interpret a carriage return (CR). Note that single-line fields, such as relation and role names and messages in sequence diagrams, always interpret a CR as a command to finish editing. The possible values are as follows:

- Checked - Multiline fields interpret a CR as a command to finish editing. Use Ctrl+CR to insert a new line.
- Cleared - Multiline fields interpret a CR as a new line. Use Ctrl+CR to exit from Edit mode.

*(Default = Cleared)*

## DefaultBoxView

The DefaultBoxView property determines how new classes and objects are displayed in object model diagrams - "Specification" view (with compartments) or "Structured" view (without compartments).

The property can take any of the following values:

- **Specification** - When you create a class or object on an object model diagram, or drag a class or object from the browser, it is opened on the diagram by using the Specification view. Also, if you right click a class and select &Make an Object&, the object created is opened by using the Specification view (regardless of the view that was used previously for the class it is based on).
- **Structured** - When you create a class or object on an object model diagram, or drag a class or object from the browser, it is opened on the diagram by using the Structured view. Also, if you right click a class and select "Make an Object", the object created is opened by using the Structured view (regardless of the view that was used previously for the class it is based on).
- **Default** - When you create a class or object on an object model diagram, or drag a class or object from the browser, it is opened on the diagram by using the Specification view. If you select a class, and select "Make an Object" from the context menu, the object created is opened by using the view that was used previously for the class it is based on.

*Default = Default*

## DefaultPackageLocation

The DefaultPackageLocation property controls what will be the owner of new packages created from the applicable diagram toolbar.

The following values are available:

- **Project**.
- **DiagramPackage**

*Default = Project*

## DeleteConfirmation

The DeleteConfirmation property specifies whether confirmation is required before deleting a graphical element from the model. Note that this property does not apply to statechart elements, which have a separate DeleteConfirmation property.

The possible values are as follows:

- **Always** - Rational Rhapsody displays a confirmation window each time you try to delete an item from the model.
- **Never** - Confirmation is not required to delete an element.
- **WhenNeeded** - Rational Rhapsody displays a message asking for confirmation if there are references to the element (or for some other reason).

*(Default = Always)*

## DelegateNewDependenciesToType

The DelegateNewDependenciesToType property specifies the classification (type or part) of two elements when you draw a dependency between them. If enabled, elements are classified as types; if disabled,

elements are classified as parts.

*(Default = Cleared)*

## DeploymentDiagramToolbar

Use the DeploymentDiagramToolbar property to set or re-order the tools on the drawing toolbar for deployment diagrams.

You can set the property to:

- Show all the default drawing tools by using the "RpyDefault" value.
- Show a new tool in addition to the default drawing tools.
- Show only those drawing tools that you want to appear on the drawing toolbar.

To enter a new tool in addition to the default drawing tools:

- Create a stereotype for the new tool and associate it with an "Applicable to" element. For example, create a stereotype called "MyNode" with an "Applicable to" element set to "Node".
- For the DeploymentDiagramToolbar property, set the value to "RpyDefault,MyNode". Always use a comma to separate values for the property.

To enter only those drawing tools that you want to appear on the drawing toolbar, enter the drawing tool names separated by a comma. For example, if you want only the Node, Component, and Dependency tools on the drawing toolbar, enter "Node,Component,Dependency" as the value for the property.

For the current list of tool names for a drawing toolbar, set the property to "RpyDefault", which is the default value. With the property set to the default value, all the available drawing tools with names appear on its drawing toolbar.

Note that this property only affects the drawing toolbar for deployment diagrams. It does not affect the drawing toolbar for "new term" stereotypes that are applicable to deployment diagrams.

*Default = RpyDefault*

## DiagramFrameMargin

The DiagramFrameMargin property defines the margin size (in pixels) between the frame of a diagram and the elements contained within the frame.

- Default - 10 pixels

## DiagramHeader

The DiagramHeader property defines how the frame header of a diagram is displayed.

- SysML Default - \$DiagramKind [\$OwnerStereotype \$OwnerType] \$OwnerName [\$Name]
- UML Default - \$Name

## DiagramOriginPolicy

The DiagramOriginPolicy property defines the diagram origin policy.

- ZeroBased - The top left corner of the diagram is fixed and set to coordinates (0,0)
- ByComponentsBounds - The origin of the diagram is dynamic and is set by the diagram content

*(Default = ByComponentsBounds)*

## EnableImageView

The EnableImageView property specifies whether images that are associated with graphical elements are displayed by default (Checked) instead of the standard geometric shape for the element.

This property has a user interface component in the element Display Options window.

*(Default = Cleared)*

## EnableQuickNavigation

Graphic elements in diagrams can have a list of shortcuts that are accessible from the title bar of the element. The property EnableQuickNavigation is used to toggle the visibility of this navigation list.

The categories of elements included in the navigation list, and the order of the categories, can be customized with the property QuickNavigationCategories.

If you want greater control over the content of the navigation list (such as showing/hiding specific elements within a given category, or changing the order of elements within a specific category), use the Compartments list, which can be found under "Display Options".

*Default = False*

## ExportedDiagramScale

This property specifies how an exported diagram is scaled and whether it can be split into separate pages for better readability. The possible values are as follows:

- FitToOnePage - Scale the exported diagram as necessary so it can fit on one page.
- NoPagination - Export the diagram as a whole, at 100% scaling. This option is for users who use HTML viewers.
- UsePrintLayout - Export the diagram by using the same settings as specified in the diagram print settings. In essence, the exported diagram is the same as if it were printed.
- For example, if the diagram print scale is 200% and the orientation is Landscape, the diagram is exported in the same way. The diagram is split as per the print page bounds (dashed lines) shown on the diagram.

In previous versions, this property included a zoom percentage (40 to 100, 150, and 200), which zoomed



the diagram to the specified percentage during export. These values were removed in Version 6.0. However, if you previously used these values in your model, they still work.

*(Default = FitToOnePage)*

## **FitBoxToItsTextuals**

This property specifies whether boxes should be automatically resized to fit their text content (such as names, attributes, and operations).

Note that if the property General::Graphics::WrapNameCompartmentText is set to True, then wrapping takes precedence and boxes are not resized, regardless of the value of FitBoxToItsTextuals.

*Default = True*

## **FixedConnectionPoints**

When you attach a connector to a diagram element, Rational Rhapsody treats the connection point as a flexible connection point. If you move a connected element, Rational Rhapsody might change the position of the connection point if it improves the appearance of the diagram.

The FixedConnectionPoints property makes it possible to create connection points that are fixed and are not to be adjusted by Rational Rhapsody when elements are moved. When this property is set to True, if you draw a connector to the edge of a diagram element, the connection point is a fixed point. If you attach the connector to the middle of an element when drawing the connector, it is a flexible connection point.

*Default = Cleared*

## **FlickerFree**

This property is currently unused.

## **GradientBoxShading**

When this property is set to true, the GradientBoxShading (default color shading) is not shown for all diagram elements.

*(Default = Not cleared)*

## **GradientNameCompartment**

When this property is set to true, the GradientNameCompartment (default color shading) is not shown for the name compartment of an element.

*(Default = Not cleared)*

## **grid\_color**

*The grid\_color property specifies the default color used for the grid lines. (Default = 0,0,0)*

## **grid\_display**

The grid\_display property currently has no effect.

## **grid\_snap**

The grid\_snap property specifies whether the Snap to Grid feature is enabled for new diagrams, regardless of whether the grid is actually displayed. The possible values are as follows:

- Checked - Objects are forced to align with the grid when you draw, move, or stretch them.
- Cleared - Objects are not forced to align with the grid when you draw, move, or stretch them.

*(Default = Cleared)*

## **grid\_spacing\_horizontal**

The grid\_spacing\_horizontal property specifies the spacing, in world coordinates, between grid lines along the X-axis when the grid is enabled for diagrams. Note that you can set this value in the GUI by selecting Layout Grid Grid Properties and changing the value of the Grid Spacing, Horizontal field.

*(Default = 0.125)*

## **grid\_spacing\_vertical**

The grid\_spacing\_vertical property specifies the spacing, in world coordinates, between grid lines along the Y-axis when the grid is enabled for diagrams. Note that you can set this value in the GUI by selecting Layout Grid Grid Properties and changing the value of the Grid Spacing, Vertical field.

*(Default = 0.125)*

## **grid\_unit\_of\_measure**

The grid\_unit\_of\_measure property specifies the minimum distances, in world coordinates, between grid points when the grid is enabled for diagrams.

*(Default = inches)*

## **HighlightSelection**

*The HighlightSelection property specifies whether items should be highlighted when you move the cursor*

*over them in a diagram. (Default = Checked)*

## ImageViewLayout

Specifies how to show as associated image. The user can select from three different options:

- ImageOnly - Displays only a large image in the diagram.
- Structured - Displays the image in addition to the top of the diagram structure.
- Compartment - Displays the image in addition to the entire diagram structure.

*(Default = ImageOnly)*

## KeepEndpoints

Beginning in release 8.2 of Rhapsody, for rectilinear connectors in diagrams, it is possible to specify that the connection points to elements should stay fixed when the elements are moved. To use this option, set the value of the property KeepEndpoints to True.

*Default = False*

## LayoutsXMLFile

The property LayoutsXMLFile is for internal Rhapsody use.

## LocateInDiagramBlinkingFreq

When you select Locate in Diagram, the relevant element in the diagram is highlighted with a flashing color outline for a few seconds. The property LocateInDiagramBlinkingFreq can be used to control the flash frequency. The value of the property should be a number of milliseconds.

*Default = 500*

## LocateInDiagramBlinkingTimes

When you select Locate in Diagram, the relevant element in the diagram is highlighted with a flashing color outline for a few seconds. The property LocateInDiagramBlinkingTimes can be used to control how many times the element flashes after it is located.

*Default = 3*

## LocateInDiagramColor

The property LocateInDiagramColor can be used to customize the color that is used to highlight the target element when you select the Locate on Diagram option.

*Default = 234,26,26*

## MaintainWindowContent

The MaintainWindowContent property specifies whether the viewport (the part of a diagram displayed in the window) is kept for window resizing operations when you change the zoom level, providing additional space in the diagram in a smooth manner. The possible values of the property are as follows:

- Checked - The elements are scaled according to the zoom factor so you see the same elements in the window, regardless of scaling.
- Cleared - As the diagram is scaled, some elements are hidden or revealed, depending on the zoom. This is the behavior provided by previous versions of Rational Rhapsody.

The following operations change the window size:

- Maximize/restore
- Tile
- Cascade
- Manual resizing by dragging the edge of the window

*You can also access this functionality in the GUI by selecting View Maintain Window Content. (Default = Cleared)*

## MarkMisplacedElements

The MarkMisplacedElements property specifies whether misplaced elements are marked in a special way. Previously, misplaced elements were shown with a small X in the upper corner.

In Rational Rhapsody 6.0, misplaced elements are marked by default with red, cross-hatched lines in the diagram. However, you can change the marking used by way of the Format window (select the project, package, or diagram, select Format, then select the misplaced element).

A misplaced element is one that looks different ly in the diagram than its actual definition in the model. For example, in a diagram, it might look as though object A is contained by object B, although that is not how object A is actually defined in the model.

Therefore, object A would be marked as misplaced that uses red, cross-hatched lines.

*(Default = Checked)*

## MaximumNumberOfPortsToDisplay

The property MaximumNumberOfPortsToDisplay is used to limit the number of ports that are displayed on a single element. If you have created more ports on the element than the number specified with this property, you will see a message indicating that not all of the ports are shown.

*Default = 100*

## MoveConnectorWithElement

The MoveConnectorWithElement property controls the sliding of a connector on the element to which the connector is attached. When this property is set to Cleared, any connector (association, dependency, transition, and so on) will stay in its global position when you move the element (source or target node) that it is connected to. This is the case until the global position becomes outside the element region, and then the connector jumps back to its original position.

By default this property is set to Checked, which means that the connector moves when the element it is connected to moves (no sliding happens).

Note that this property has no effect for connectors that have a rectilinear line shape, they always try to stay in their global position (slide on the source/target element).

*Default = Checked*

## MoveWithoutContainedElements

The MoveWithoutContainedElements property controls whether moving an element on the diagram moves the children for that element as well.

*Default = Cleared*

## MultiScaleOneByOne

*The MultiScaleOneByOne property specifies whether objects in the diagram are scaled as a group (Cleared) or each independently (Checked). (Default = Cleared)*

## ObjectModelDiagramToolbar

Use the ObjectModelDiagramToolbar property to set or re-order the tools on the drawing toolbar for object model diagrams.

You can set the property to:

- Show all the default drawing tools by using the "RpyDefault" value.
- Show a new tool in addition to the default drawing tools.
- Show only those drawing tools that you want to appear on the drawing toolbar.

To enter a new tool in addition to the default drawing tools:

- Create a stereotype for the new tool and associate it with an "Applicable to" element. For example, create a stereotype called "MyObject" with an "Applicable to" element set to "Object".
- For the ObjectModelDiagramToolbar property, set the value to "RpyDefault,MyObject". Always use a comma to separate values for the property.

To enter only those drawing tools that you want to appear on the drawing toolbar, enter the drawing tool

names separated by a comma. For example, if you want only the Object, Class, AssociationEnd, Link, and Dependency tools on the drawing toolbar, enter "Object,Class,AssociationEnd,Link,Dependency" as the value for the property.

For the current list of tool names for a drawing toolbar, set the property to "RpyDefault", which is the default value. With the property set to the default value, all the available drawing tools with names appear on its drawing toolbar.

Note that this property only affects the drawing toolbar for object model diagrams. It does not affect the drawing toolbar for "new term" stereotypes that are applicable to object model diagrams.

*Default = RpyDefault*

## PanelControlTypeBindingDisplay

This property determines which attribute types to display in the browser on the Element Binding tab for a panel diagram control. The possible values are:

- PredefinedOnly - show only the predefined types
- All - show all types

*Default = PredefinedOnly*

## PanelDiagramToolbar

Use the PanelDiagramToolbar property to set or re-order the tools on the drawing toolbar for panel diagrams.

You can set the property to:

- Show all the default drawing tools by using the "RpyDefault" value.
- Show a new tool in addition to the default drawing tools.
- Show only those drawing tools that you want to appear on the drawing toolbar.

To enter a new tool in addition to the default drawing tools:

- Create a stereotype for the new tool and associate it with an "Applicable to" element. For example, create a stereotype called "MyPanelActor" with an "Applicable to" element set to "Actor."
- For the PanelDiagramToolbar property, set the value to "RpyDefault,MyPanelActor." Always use a comma to separate values for the property.

To enter only those drawing tools that you want to appear on the drawing toolbar, enter the drawing tool names separated by a comma. For example, if you want only the Knob, Guage, and Slider tools on the drawing toolbar, enter "Knob,Guage,Slider" as the value for the property.

For the current list of tool names for a drawing toolbar, set the property to "RpyDefault", which is the default value. With the property set to the default value, all the available drawing tools with names appear on its drawing toolbar.

Note that this property only affects the drawing toolbar for panel diagrams. It does not affect the drawing

toolbar for "new term" stereotypes that are applicable to panel diagrams.

*Default = RpyDefault*

## PopulateClassSize

The PopulateClassSize property determines the size of the class or object when Rational Rhapsody autopopulates an OMD. The possible values are:

- OldStyle - uses the class size that was used for autopopulate prior to version 6.1 MR-1: greater in width than in height
- DefaultSize - uses the default class size that is used for creating ordinary OMDs: greater in height than in width.

*Default = DefaultSize*

## PopulateExpandedSelection

The PopulateExpandedSelection property enables a larger selection ability in the populate window. If the check box is checked, when you right-click an item in the tree-list of the populate window is, a menu lists with the following options:

- Select Only
- Select with decedents
- Select with Base classes

*(Default = Cleared)*

## PopulateHierarchyStyle

The PopulateHierarchyStyle property determines the Hierarchy layout style. Users can select one of two options:

- Top-Bottom: In the case of inheritance, the base class is at the top and the derived class at the bottom (Default style). This also applies to other links, like association, where the target class of the association is at the top, and the source class at the bottom
- Bottom-Top: Opposite of Top-Bottom, that is, the base class is on the bottom and the derived class on the top (this was the previous layout style)

*(Default = Top-Bottom)*

## PopulateMaxBoxSize

The PopulateMaxBoxSize property is used to limit the size of diagram elements when a diagram is auto-populated.

The value of the property is a comma-separated list of four integers. The maximum width in pixels is the difference between the third number and the first number. The maximum height in pixels is the difference

between the fourth number and the second number.

For example, if you entered 0,0,200,200, the maximum size would be 200 pixels by 200 pixels.

If the value is 0,0,0,0 then no size limit is applied to elements during auto-population of diagrams - elements display as large as necessary to accommodate the contained text.

*Default = 0,0,0,0*

## PortHighlighting

The property PortHighlighting can be used to specify what graphic elements should be highlighted when you hover over a port. The possible values for this property are:

- PortOnly - only the port itself is highlighted
- WithConnectors - the port and any connectors attached to the port are highlighted
- WithConnectorsAndPorts - the port, any connectors attached to the port, and ports on the other side of the connectors are highlighted

*Default = WithConnectorsAndPorts*

## PrintLayoutExportScale

The PrintLayoutExportScale property specifies the factor by which the Windows metafile format (WMF) files are scaled down in order to fit on one page. The default value, 75, guarantees that the diagram fits into a single page in Microsoft Word.

## RepeatedDrawing

The RepeatedDrawing property specifies whether repetitive drawing mode (stamp mode) is enabled. You can use repetitive drawing mode to create a box element with a single click; double-clicking produces two of the same box elements.

By default, each time you want to add an element to a diagram, you must first click the appropriate icon in the drawing toolbar.

In some cases, however, you might want to add a number of elements of the same type. To facilitate this, Rational Rhapsody includes a "repetitive drawing mode."

To enter repetitive drawing mode, click the Stamp icon in the Layout toolbar. After selecting a tool in the drawing toolbar, you are able to continue drawing elements of that type without selecting the tool again each time.

If you choose a different tool from the toolbar, then Rational Rhapsody allows you to draw multiple elements of the newly selected type.

After you click the icon, Rational Rhapsody remains in repetitive drawing mode until you turn it off. To turn off the repetitive mode, just click the Stamp icon a second time.



*(Default = Cleared)*

## **ResizeWithoutContainedElements**

The `ResizeWithoutContainedElements` property controls whether resizing an element on the diagram resizes the children for that element as well.

*Default = Cleared*

## **RotateDiagramOnExport**

The `RotateDiagramsOnExport` property determines whether a diagram should be rotated in a report generated with Rational Rhapsody ReporterPLUS. This property takes effect only when `General::Graphics::ExportedDiagramScale` is set to `UsePrintLayout`. The possible values are `RotateLeft`, `RotateRight`, and `No` (Default value).

This property replaced the `LandscapeRotateOnExport` property. If an older model overrode the `LandscapeRotateOnExport` property, it overrides this property as well.

## **ScaleToFitExportedDiagram**

*The `ScaleToFitExportedDiagram` property specifies whether the diagram is scaled to fit in a report generated with Rational Rhapsody ReporterPLUS. (Default = Checked)*

## **SearchInDiagramShortcut**

When a diagram has focus, you can search for specific text in the diagram. You can use the property `SearchInDiagramShortcut` to select the keyboard shortcut to use for this search - `Ctrl+F` or `Ctrl+Shift+F`.

*Default = `Ctrl+Shift+F`*

## **SequenceDiagramToolbar**

Use the `SequenceDiagramToolbar` property to set or re-order the tools on the drawing toolbar for sequence diagrams.

You can set the property to:

- Show all the default drawing tools by using the "RpyDefault" value.
- Show a new tool in addition to the default drawing tools.
- Show only those drawing tools that you want to appear on the drawing toolbar.

To enter a new tool in addition to the default drawing tools:

- Create a stereotype for the new tool and associate it with an "Applicable to" element. For example, create a stereotype called "MyMessage" with an "Applicable to" element set to "Message".

- For the SequenceDiagramToolbar property, set the value to "RpyDefault,MyMessage". Always use a comma to separate values for the property.

To enter only those drawing tools that you want to appear on the drawing toolbar, enter the drawing tool names separated by a comma. For example, if you want only the Message, Reply Message, and Timeout tools on the drawing toolbar, enter "Message,ReplyMessage,Timeout" as the value for the property.

For the current list of tool names for a drawing toolbar, set the property to "RpyDefault", which is the default value. With the property set to the default value, all the available drawing tools with names appear on its drawing toolbar.

Note that this property only affects the drawing toolbar for sequence diagrams. It does not affect the drawing toolbar for "new term" stereotypes that are applicable to sequence diagrams.

*Default = RpyDefault*

## ShowActivityFrame

*Select this property to add an activity frame automatically to a new activity diagram when it is created.  
(Default = Cleared)*

## ShowCommonToolbox

The property ShowCommonToolbox can be used to hide the "common" toolbox which is ordinarily included on the Drawing toolbar. This toolbox includes tools such as Requirements and Notes.

The default value of the property is provided to help you hide/show these tools if you are defining "user-perspectives" for different types of users. For example, you can define a property called Basic\_ShowCommonToolbox and set its value to "No" if you want to hide these tools in the perspective that you named "Basic".

For more information on defining user-perspectives in Rhapsody, see the topic titled "Configuring the user interface with perspectives" in the Rhapsody Knowledge Center.

*Default =*

*?<IsConditionalProperty>?<begin>\${General::Graphics::\${General::Model::SelectedPerspective}\_ShowCommonToolb*

## ShowCompartmentsTitle

When classes are displayed in Specification view, compartments are displayed for elements such as attributes and operations.

The ShowCompartmentsTitle property can be used to specify that headings should be displayed to identify the different compartments.

The property can be set at the diagram level or higher.

When you change the value of the property, it affects the appearance of any classes subsequently added to the diagram, but does not affect the appearance of classes already on the diagram.

When this property is set to True, you can use the CompartmentsTitleFont property to choose the font that Rational Rhapsody should use for these compartment headings.

*Default = Cleared*

## ShowDiagramFrame

This property controls whether diagram frames (the line border around diagrams) is shown by default or not. This feature can be overridden by right-clicking anywhere in the in the diagram canvas and selecting "Show/Hide Diagram Frame."

*(Default = Cleared)*

## ShowEdgeTracking

*The ShowEdgeTracking property specifies whether to show the "ghost" edges of a linked element when you move it. This is set to True by default, which means you can see the edges. (Default = Checked)*

## ShowFreeShapesToolbox

The property ShowFreeShapesToolbox can be used to hide the "free shapes" toolbox which is ordinarily included on the Drawing toolbar. This toolbox includes drawing tools such as Rectangle and Ellipse.

The default value of the property is provided to help you hide/show these tools if you are defining "user-perspectives" for different types of users. For example, you can define a property called Basic\_ShowFreeShapesToolbox and set its value to "No" if you want to hide these tools in the perspective that you named "Basic".

For more information on defining user-perspectives in Rhapsody, see the topic titled "Configuring the user interface with perspectives" in the Rhapsody Knowledge Center.

*Default =*

*?<IsConditionalProperty>?<begin>\$<General::Graphics::<General::Model::SelectedPerspective>\_ShowFreeShapesTo*

## ShowInheritedNotation

This property determines how to show the inherited attributes and operations of a class in a diagram.

*Default = Enabled*

## ShowLabels

The ShowLabels property is a Boolean value that specifies whether to display labels instead of names in the browser or diagrams, depending on which property is set.

*(Default = Cleared)*

## ShowMultipleStereotypes

For graphic elements in diagrams, the ShowMultipleStereotypes property can be used to determine whether all of the applied stereotypes should be displayed above the element name, or just the first stereotype.

*Default = True*

## ShowStereotypes

The property ShowStereotypes is used to control whether or not stereotypes are displayed for items in compartments, and how the stereotypes should be displayed. The possible values for this property are:

- No - stereotypes are not displayed
- Prefix - stereotypes are displayed to the left of the element name
- Suffix - stereotypes are displayed to the right of the element name

The property is set at the project level.

Default = Prefix

## SnapEdgeAlignmentDistance

The property SnapEdgeAlignmentDistance is used to make it easier to straighten connectors that have bends in them. When the distance required to straighten such a connector is smaller than the value set for this property, you only have to move the connector slightly in the appropriate direction, and the connector will "snap" to the point required to make it straight.

The value of the property should be a number of pixels. To disable the feature, set the value of the property to 0.

*Default = 10*

## SmartRouting

Set the property SmartRouting to True if you want Rhapsody to find the optimal path to use for connectors when they are added to a diagram. This feature takes a number of factors into account, for example, preventing situations where connectors cut across intermediate nodes.

*Default = False*

## SnapToObjects

When moving or resizing elements in a diagram, you can have the element "snap" to points where they will be aligned with neighboring diagram elements. This feature works like the "snap to grid" feature, but the anchor points are based on the position of the other elements rather than the grid. Elements will jump

to these points when they are within the range defined by the property `SnapToObjectsDistance`.

You can use the `SnapToObjects` property to disable/enable this feature.

Note that if the "snap to grid" feature is enabled, the grid will be used as the anchor points rather than the neighboring elements, regardless of the value of the property `SnapToObjects`.

*Default = True*

## **SnapToObjectsCenterLine**

Use the `SnapToObjectsCenterLine` property to determine whether or not Rhapsody should use the "snap to object" feature to center-align elements with other elements in the diagram. If the property is set to `False`, Rhapsody will only use the "snap to object" feature for top/bottom and left/right alignment.

*Default = True*

## **SnapToObjectsDistance**

Use the `SnapToObjectsDistance` property to set the threshold (in pixels) for "snapping" elements to be aligned with other diagram elements.

*Default = 10*

## **SnapToObjectsLineColor**

If the property `SnapToObjectsShowLines` is set to `True`, you can use the `SnapToObjectsLineColor` property to customize the color of the guide lines that are displayed.

*Default = 180,0,0*

## **SnapToObjectsShowLines**

If you are using the "snap to object" feature for moving and resizing diagram elements, you can use the property `SnapToObjectsShowLines` to determine whether or not Rhapsody displays lines that show how the element will be aligned with the neighboring elements in the diagram.

*Default = True*

## **StatechartToolbar**

Use the `StatechartToolbar` property to set or re-order the tools on the drawing toolbar for statecharts.

You can set the property to:

- Show all the default drawing tools by using the "RpyDefault" value.

- Show a new tool in addition to the default drawing tools.
- Show only those drawing tools that you want to appear on the drawing toolbar.

To enter a new tool in addition to the default drawing tools:

- Create a stereotype for the new tool and associate it with an "Applicable to" element. For example, create a stereotype called "MyState" with an "Applicable to" element set to "State".
- For the StatechartToolbar property, set the value to "RpyDefault,MyState". Always use a comma to separate values for the property.

To enter only those drawing tools that you want to appear on the drawing toolbar, enter the drawing tool names separated by a comma. For example, if you want only the State, Transition, Default Transition, and Termination State tools on the drawing toolbar, enter "State,Transition,DefaultTransition,TerminationState" as the value for the property.

For the current list of tool names for a drawing toolbar, set the property to "RpyDefault", which is the default value. With the property set to the default value, all the available drawing tools with names appear on its drawing toolbar.

Note that this property only affects the drawing toolbar for statecharts. It does not affect the drawing toolbar for "new term" stereotypes that are applicable to statecharts.

*Default = RpyDefault*

## StereotypeBitmapTransparentColor

The StereotypeBitmapTransparentColor property creates a “transparent” background for bitmaps associated with stereotypes (so only the graphics are displayed in the class box).

To create a transparent background:

At the project level, set this property to the RGB value of the bitmap background.

See the Rational Rhapsody help for information on associating bitmaps with stereotypes.

*Default = 255, 0, 255*

Note: When creating transparent backgrounds, do not use default RGB value. This color value is ignored by Rhapsody and therefore can not be set to transparent.

## StructureDiagramContext

The StructureDiagramContext property controls the context (to where elements are placed when created on the diagram) of a structure diagram. This property has the following values:

- Default - The default context of any static diagrams, which is the default package (or any other package the user assigned it by way of the Features window for the diagram).
- ClassOwner - The context of the diagram is the class owner of the structure diagram. This means that any elements created in this diagram will be placed inside the class owner. In addition, if a DiagramFrame exists in the diagram, the user can display the port for the class on the DiagramFrame.

*Default = Default. Note that in SysML, the InternalBlockDiagram (a new term on StructureDiagram) has ClassOwner as the default value for this property.*

## StructureDiagramToolbar

Use the StructureDiagramToolbar property to set or re-order the tools on the drawing toolbar for structure diagrams.

You can set the property to:

- Show all the default drawing tools by using the "RpyDefault" value.
- Show a new tool in addition to the default drawing tools.
- Show only those drawing tools that you want to appear on the drawing toolbar.

To enter a new tool in addition to the default drawing tools:

- Create a stereotype for the new tool and associate it with an "Applicable to" element. For example, create a stereotype called "MyObject" with an "Applicable to" element set to "Object".
- For the StructureDiagramToolbar property, set the value to "RpyDefault,MyObject". Always use a comma to separate values for the property.

To enter only those drawing tools that you want to appear on the drawing toolbar, enter the drawing tool names separated by a comma. For example, if you want only the Object, Port, and Link tools on the drawing toolbar, enter "Object,Port,Link" as the value for the property.

For the current list of tool names for a drawing toolbar, set the property to "RpyDefault", which is the default value. With the property set to the default value, all the available drawing tools with names appear on its drawing toolbar.

Note that this property only affects the drawing toolbar for structure diagrams. It does not affect the drawing toolbar for "new term" stereotypes that are applicable to structure diagrams.

*Default = RpyDefault*

## TableHeaderMenuAlignment

The TableHeaderMenuAlignment property can be used to specify the location of the menu button (for sorting and filtering) in table header cells - at the left edge of the header cell or the right edge.

*Default = Right*

## TemplateParamsLayout

In version 7.2 of Rational Rhapsody, a change was made to the way that template parameters are displayed in object model diagrams.

Previously, if you changed the size of the template element in the diagram, the size of the box that displays the template parameters would also change proportionately.

Beginning with 7.2, the size of the box containing the template parameters does not change when you change the size of the template element in the diagram.

In order to maintain the previous behavior for pre-7.2 models, the `TemplateParamsLayout` property was added to the C++ and Java backward compatibility settings with a value of `Regular`. The possible values for the property are:

- `Regular` - the size of the template parameter box changes together with the template element
- `FixedSize` - the size of the template parameter box remains a fixed size when you change the size of the template element

*Default = Regular*

## TimingDiagramToolbar

You can use the `TimingDiagramToolbar` property to customize the tools that are included on the drawing toolbar for timing diagrams.

When the value of the property is set to `RpyDefault`, the toolbar displays the default drawing tools for timing diagrams.

If you have created custom diagram elements (as described in the topic "Creating customized diagram elements" in the help), you can add them to the drawing toolbar by adding the name of the new stereotype to the value of this property.

If the value of the property contains more than one item, use commas to separate the items.

If you do not want all of the default drawing tools to appear in the drawing toolbar, you can replace `RpyDefault` with a list of the drawing tools that you want to include. The names of the drawing tools can be taken from the table that is included in the topic "Diagram elements" in the help.

Note that this property only affects the drawing toolbar for timing diagrams. It does not affect the drawing toolbar for "new term" stereotypes that are applicable to timing diagrams.

*Default = RpyDefault*

## Tool\_tips

The `Tool_tips` property enables the display of tooltips.

*Default = Checked*

## ToolTipFormat

The `ToolTipFormat` property lets you set the format for the content of a tooltip.

*Default = \$name\$description*



## UpdateModelFromGraphicChange

This property determines whether or not the model hierarchy changes correspondingly to a graphical hierarchy change.

The following two properties only take effect when this property is set to TRUE:

- General::Graphics::AllowObjectReparenting
- General::Graphics::AllowRequirementReparenting

Default value = Cleared

## UseCaseDiagramToolbar

Use the UseCaseDiagramToolbar property to set or re-order the tools on the drawing toolbar for use case diagrams.

You can set the property to:

- Show all the default drawing tools by using the "RpyDefault" value.
- Show a new tool in addition to the default drawing tools.
- Show only those drawing tools that you want to appear on the drawing toolbar.

To enter a new tool in addition to the default drawing tools:

- Create a stereotype for the new tool and associate it with an "Applicable to" element. For example, create a stereotype called "MyActor" with an "Applicable to" element set to "Actor."
- For the UseCaseDiagramToolbar property, set the value to "RpyDefault,MyActor." Always use a comma to separate values for the property.

To enter only those drawing tools that you want to appear on the drawing toolbar, enter the drawing tool names separated by a comma. For example, if you want only the Use Case, Actor, and Association tools on the drawing toolbar, enter "UseCase,Actor,Association" as the value for the property.

For the current list of tool names for a drawing toolbar, set the property to "RpyDefault", which is the default value. With the property set to the default value, all the available drawing tools with names appear on its drawing toolbar.

Note that this property only affects the drawing toolbar for use case diagrams. It does not affect the drawing toolbar for "new term" stereotypes that are applicable to use case diagrams.

*Default = RpyDefault*

## WarnForReferencing

The WarnForReferencing property specifies whether Rational Rhapsody should issue a notification message to notify the user that the graphical element is about to be re-referenced.

Available for requirement graphical elements only.

*Default = cleared*

## WrapNameCompartmentText

The property WrapNameCompartmentText controls how text appears in a name compartment. When the property is set to True, the text wraps according to the size of the compartment.

Note that if the property WrapNameCompartmentText is set to True, then Rhapsody will wrap the element name rather than enlarging the box containing it, regardless of the value of the property General::Graphics::FitBoxToItsTextuals.

*Default = False*

## HTTPServer

The HTTPServer metaclass contains properties that control the Rational Rhapsody Web server.

### FirstPortSearch

When you use the Webify feature, Rational Rhapsody starts a web server and then looks for an available port to connect to. The FirstPortSearch property specifies the first port to check.

*Default = 27463*

### LastPortSearch

When you use the Webify feature, Rational Rhapsody starts a web server and then looks for an available port to connect to. The search for an available port starts at the value specified by the FirstPortSearch property. The LastPortSearch property specifies the last port to check.

*Default = 27473*

### LaunchFile

*This property is a path pointing at the batch file used to launch the Web server. (Default = \$BatchPath/startserver.bat)*

## Message

Contains properties relating to messages in sequence diagrams.

## DataFlowElementPattern

When you add a dataflow to a sequence diagram, the Features window for the dataflow contains a list of the ports that you can select. The property DataFlowElementPattern can be used to control what type of ports are displayed in this list. For more information on the options that can be included in the syntax for the property value, see the topic "Context patterns" in the Rhapsody help.

*Default = flowPort || flowPort, type:Attribute || proxyPort, type:Attribute*

## Model

The Model metaclass contains properties that control the general features of model elements, such as the format of element names.

## ActiveCodeViewSensitivity

The ActiveCodeViewSensitivity property controls the update rate of the active code view (ACV). The possible values are as follows:

- ElementSelection - The ACV is updated whenever you modify the selection and whenever there are changes in the model.
- OnFocus - The ACV is updated only when you set it as the focused view (by clicking on the ACV view pane).

*(Default = ElementSelection)*

## ActualCallRegExp

The ActualCallRegExp property specifies the regular expression describing the format of a legal actual call to an operation when the call is part of a transition. Usually, it is the action part of a transition. The default regular expression `^(.+)\(.*)$` is evaluated as follows:

- The circumflex character (^) means that matching should begin from the start of the string. This matches a prefix of the string. The dollar sign (\$) matches the NULL character at the end of the input string.
- The sequence `((.+)\(` means to match one or more characters until an open parenthesis is found. This implies that if there are no characters before the opening parenthesis in the input string, there is no match.
- The period matches any single character. For example, the expression `"..."` would match any three characters.
- The opening parenthesis (`((`) has a special meaning in regular expressions. Therefore, it is preceded with the backslash escape character, which tells the parser to ignore the usual meaning of the opening parenthesis and look for a literal "(" character in the string. For example, to match the string "(a)", you would use the regular expression `"\((a)"`.

*(Default = ^(.+)\)(.\*\)\$)*

## AdditionalHelpersFiles

The AdditionalHelpersFiles property can be used to specify additional .hep files that should be loaded for a project, beyond the .hep file specified by using the HelpersFile property.

The value of this property should be a comma-separated list of the additional .hep files you would like to associate with the project.

*Default = Blank*

## AdditionalLanguageKeywords

The AdditionalLanguageKeywords property specifies a comma-separated list of language-specific keywords to be color-coded by the Rational Rhapsody internal editor, in addition to the default keywords (such as "class" and "public").

*(Default = empty string)*

## AddNewMenuStructure

The AddNewMenuStructure property is used to define the structure of the "Add New" menu that opens when you right-click an item on the Rational Rhapsody browser.

The structure of the property is: Metaclass name, Submenu name/Metaclass name

For a list of the available metaclasses, see the metaclasses.txt file in <Rational Rhapsody installation path>\Doc. You can use rpy\_separator to insert a separator bar between sections.

Example value: Package, Class, rpy\_separator, Stereotype, rpy\_separator, Annotations/Constraint, Annotations/Comment, rpy\_separator, TableMatrix/TableLayout, TableMatrix/TableView, TableMatrix/rpy\_separator, TableMatrix/MatrixLayout, TableMatrix/MatrixView

With this example value, when you right-click a package on the Rational Rhapsody browser and select "Add New", the menu opens and looks like this:

Package Class ----- Stereotype ----- Annotations Constraint Comment ----- TableMatrix TableLayout  
TableView ----- MatrixLayout MatrixView

## ApplyNewTermSemantic

The ApplyNewTermSemantic property applies to NewTerms. The NewTerm feature is used to define new types based on existing out-of-the-box types. Once done, a new type exists. If the ApplyNewTermSemantic property is checked, the NewTerms work. If cleared, the NewTerms feature does not work.

*(Default = Checked)*

## **AutoCascadeAddNewMenu**

The AutoCascadeAddNewMenu property, automatically cascades the "add new" menu (checked) according to the profiles that contain each NewTerm. The NewTerm feature is used to define new types based on existing out-of-the-box types. Once done, a new type exists.

*(Default = Checked)*

## **AutoSaveInterval**

The AutoSaveInterval property specifies the interval, in minutes, at which Rational Rhapsody automatically saves your project. The following naming scheme applies:

- Project file config.rpy is copied to the file \_auto.rpy.
- Repository \config\_rpy is copied to the directory \config\_auto\_rpy.

*Default = 10*

## **AutoSynchronize**

The AutoSynchronize property is a Boolean value that determines whether Rational Rhapsody runs synchronization.

When this property is set to Checked, each time Rational Rhapsody gets the focus (for example, if you leave the product to read e-mail, then switch back to Rational Rhapsody), the product runs the Synchronize functionality.

The started synchronize can be a synchronization with the files on the file system, view, or configuration management archive, depending on the environment.

Use this property along with the ConfigurationManagement::General::UseUnitTimeStamps property.

*Default = Cleared*

## **AvailableMetaclasses**

Rational Rhapsody allows you to hide any out-of-the-box element types that your users do not need. The availability of metaclasses is determined by the AvailableMetaclasses property. This property is defined by using a comma-separated list of strings.

To keep all of the out-of-the box metaclasses, leave this property blank.

To limit the availability of certain metaclasses, use this property to indicate only the metaclasses that you would like to have available. The strings to use to represent the different metaclasses can be found in the metaclasses.txt file in the Doc directory of your Rational Rhapsody installation.

## AvailableNewTerms

The availability of new terms is determined by the AvailableNewTerms property. This property is defined by using a comma-separated list of strings.

To keep all of the out-of-the box terms, leave this property blank.

To limit the availability of certain new terms, use this property to indicate only the new terms that you would like to have available.

## BackUps

The BackUps property specifies the maximum number of backups created when you save.

The possible values are None, One, and Two.

*(Default = None)*

## BlockIsSavedUnit

*The BlockIsSavedUnit property determines whether new blocks are saved as units (separate files) by default. (Default = Cleared)*

## CheckRoundtrip

The CheckRoundtrip property determines whether roundtrip is enabled after performing a Check Model operation. Sometimes, code generation unexpectedly displays a message saying that files have been externally changed and that roundtrip might be needed, even if this is not the case.

To disable roundtrip (and this message), set this property to Cleared. When roundtrip is enabled, a shortcut confirmation option lets you select a Yes to All or No to All button.

*(Default = Checked)*

## CheckSpelling

When you enter a description for an element in its Features window, Rational Rhapsody automatically checks the text for spelling errors when you click Apply. If you want to turn off this behavior, set the value of the property CheckSpelling to False.

*Default = True*

## CheckSpellingOfCamelCaseWords

Since CamelCase is often an indication that a word is some sort of code element, such as a variable name,

the spellcheck feature does not check CamelCase words by default. If you want spell checking to include words written in CamelCase, set the value of the property `CheckSpellingOfCamelCaseWords` to `True`.

*Default = False*

## CheckSpellingOfUpperCaseWords

Since the spelling of a word in upper case characters is often an indication that a word is some sort of code element, such as a constant, the spellcheck feature does not check words that are all upper case by default. If you want spell checking to include words written entirely in upper case, set the value of the property `CheckSpellingOfUpperCaseWords` to `True`.

*Default = False*

## ClassCodeEditor

The `ClassCodeEditor` property specifies which kind of editor is started when editing classes. The editor can be displayed in either a modal or modeless window. A modeless window does other work in other windows while it is open, whereas a modal window does not allow you to select any other window while it is open.

The possible values are as follows:

- `Internal` - Use the Rational Rhapsody internal editor for both modal and modeless editing.
- `Associate` - Use the editor that is associated with `.h` and `.cpp` files as set in the Windows registry for modeless editing, and use the internal editor for modal editing.
- `CommandLine` - Use the editor specified in the `EditorCommandLine` property for both modal and modeless editing.

*(Default = Internal)*

## ClassIsSavedUnit

*The `ClassIsSavedUnit` property determines whether new classes are saved as units (separate files) by default. (Default = Cleared)*

## CommonClassifiers

The `CommonClassifiers` property can be used to control what packages should be used for populating the Realization drop-down list on the General tab of the features window for instance lines in sequence diagrams. Use of this property can improve GUI response time for large models.

For the value of this property, enter a comma-separated list of package names, for example: `pkg1,pkg2`. (The list should not contain spaces.) Only the classifiers from the specified packages is included in the drop-down list.

If the property is left empty, classifiers from all the packages is included in the drop-down list.

If you choose to use this property to specify a list of packages, the drop-down list includes a "Select" option that allows the user to select classifiers that are not displayed in the list by default.

## CommonList

The CommonList property controls which elements appear in the top section of the Add New menu (referred to as the common list), when applicable. You can re-order, remove, or re-add any of these elements by doing so through the CommonList property.

Note the following:

- Whatever element that is removed from CommonList is displayed in the middle portion of the Add New menu if that element is relevant for your project. The element must appear somewhere if it is a valid element.
- The General::Model::AddNewMenuStructure property overrides this property.

*Default =*

*Function, Variable, Attribute, PrimitiveOperation, TriggeredOperation, Reception, Constructor, Initializer, Destructor, CleanUp,*

## CommonTypes

The CommonTypes property specifies which types are listed as alternatives for attributes, variables, and arguments to make commonly used types easily accessible.

For example, you can omit the types listed in the Predefined package from the list of alternative types, and instead specify a list of packages that contain the types defined according to your design and code standards. When you use the CommonTypes property, you can specify a list of packages and files that contain types that you use often, or types that are "basic" types for the project.

The value of the property is a comma-separated list of full paths of packages; the types are listed in the order specified in the property. If the value is \$ALL, all types from all packages are included in the list.

If this property is empty, Rational Rhapsody uses Version 5.2 behavior. Note that if you do not explicitly include the PredefinedType package, its contents are not included in the list (although the package is loaded and visible in the browser). As with previous versions, you can use select to navigate and select any type defined in the model.

*(Default = empty string)*

## CompareBuildNumberInRepository

The CompareBuildNumberInRepository property is a Boolean value that can be used to prevent Rational Rhapsody from opening a model that was developed with a different Rational Rhapsody build. This prevents accidental "upgrading" of models.

When this property is set to True, Rational Rhapsody does not open models developed with previous builds, and displays a message to this effect if the user tries to open such a model.

*Default = False*



## ComponentFileIsSavedUnit

The ComponentFileIsSavedUnit property determines whether or not new component files added to the model are automatically saved as units.

*Default = Cleared*

## ComponentIsSavedUnit

The ComponentIsSavedUnit property determines whether new components are saved as units (separate files) by default.

*(Default = Checked)*

## CSVImportPolicy

The CSVImportPolicy property indicates whether to prefer model values or imported values during CSV import.

*Default = PreferFile*

## DefaultDirectoryName

The DefaultDirectoryName property defines the directory name used for newly created packages that are stored in a separate directory.

A package can be stored in a separate directory by selecting the "Store in separate directory" check box on the Unit Information window or by setting the General::Model::DefaultDirectoryScheme property to "PackageAsDirectory."

The possible values are:

- AsFileName - The name of the directory where the .sbs file is stored is the name of the package.
- AsDirectoryName - The name of the directory where the .sbs file is stored is the name of the sbs file derived from the package.

*Default = AsFileName*

## DefaultDirectoryScheme

When a model element is defined as a "unit", it is saved as a separate file in the file system. The DefaultDirectoryScheme property determines the directory structure used for such files.

If the property value is set to Flat, all new units are saved in a single directory, regardless of where they are in your model hierarchy.

If the property value is set to `PackageAsDirectory`, a new directory is created for each package you create in your model. The name of the directory is the same as the name of the package, and it contains the .sbs file that represents the package as well as the files that represent any other units contained in the package, such as classes or subpackages. As a result, the directory structure for the units reflects the package hierarchy in your model.

Note that even when the value of the property is set to `Flat`, you can override this behavior for a specific package by selecting the "Store in separate Directory" option in the Unit Information window for the package.

*Default = Flat*

## DefaultType

The `DefaultType` property specifies the default data type to be used for any newly created attribute, variable, or argument.

Note: The setting must include the `package::usertype` for the code to be generated.

*You can set the default data type to any type defined by your design and coding standards. For example, you could set the default data type to "package::int32" or to a user-defined type called "not\_defined" to emphasize that the type for this attribute, variable, or argument was not yet defined. (Default = empty string)*

## DefaultUnitFileName

If you merge a unit into a model, and the model already contains a unit with the same filename, Rational Rhapsody automatically adds a numerical suffix to the filename in order to differentiate it from the existing file. This is problematic in scenarios where it is important that the original name of the unit be maintained.

The `General::Model::DefaultUnitFileName` property can be used to define a "template" for creating filenames on the basis of element names. In cases where there is a possibility that units is merged from other models, this property can be used to establish unique file-naming schemes before development of the models begins.

For the value of this property, you can combine a model-unique string with the keywords `$name`, `$owner` and `$pid`. If you want the filename to reflect the element it represents, make sure to include the keyword `$name` in the property value.

For example, you can specify the value of the property as `modelA_$name`, and this prevents any naming conflicts if you later bring in elements with the same name from a different model.

If the value of the property is left blank, then Rational Rhapsody uses its default naming scheme in which filenames are taken from the name of the element they represent.

Note that this property only affects filenames for elements that are automatically saved as units as soon as you create them. If you take an element that is not a unit, and select `Create Unit` from the context menu, the value of this property is not to be used.

*Default = Blank*

## DescriptionEditor

*The DescriptionEditor property specifies the editor to use when editing element descriptions (for example, Word). See the EditorCommandLine property for information on specifying the editor to use for external code. (Default = empty string)*

## DescriptionEditorSupportsRTF

*The DescriptionEditorSupportsRTF property specifies whether the editor specified in the DescriptionEditor property supports rich text format (RTF). (Default = Cleared)*

## DescriptionTextLimit

The DescriptionTextLimit property determines the maximum length of the descriptions that can be entered for elements in a Rational Rhapsody model.

The value of the property represents the number of bytes that should be allocated for the description.

*Default = 32768*

## DiagramIsSavedUnit

The DiagramIsSavedUnit property determines whether diagrams are saved as units (separate files) by default.

*Default = Cleared*

## DiagramsToolbar

If you create a custom diagram type, you also have the option of including an icon for the new diagram type in the Diagrams toolbar.

To add the new type of diagram to the Diagrams toolbar you must modify the value of the DiagramsToolbar property to include the name of the new diagram type in the comma-separated list, for example, OV-1,RpySeparator,RpyDefault.

The strings to use in this list are as follows:

- ActivityDiagram
- ClassDiagram
- CollaborationDiagram
- ComponentDiagram
- DeploymentDiagram

- ObjectModelDiagram
- PanelDiagram
- SequenceDiagram
- Statechart
- StructureDiagram
- UseCaseDiagram
- RpyDefault - Use this string if you want to add the default diagram icons in addition to the icon for your customized diagram type.
- RpySeparator - Use this string if you want to add a divider between icons; for example, ActivityDiagram,RpySeparator,PanelDiagram.

*Default = Empty string. (The toolbar automatically shows the default icons.)*

## **DisableIgnoreReadOnlyFilesOption**

If you try to delete elements that have references in read-only files, Rational Rhapsody issues a warning that the deletion will result in unresolved references in the model unless the files are made writable. The dialog that is displayed lets you choose from among three possible actions: check out the read-only files, make the files read/write, or ignore the fact that there are references in read-only files. You can disable the "ignore" option by setting the value of the DisableIgnoreReadOnlyFilesOption property to True.

*Default = False*

## **DMProjectArea**

The DMProjectArea property specifies the project area on the Rational Design Manager server to which you want to connect using the Rational Rhapsody Design Manager collaboration view.

## **DMServerURL**

The DMServerURL property specifies the URL of the Rational Design Manager server to which you want to connect with using the Rational Rhapsody Design Manager collaboration view.

## **DMWorkspace**

After you have connected to an imported model, the property DMWorkspace is used to save the name of the workspace on the DM server so that relevant information can be retrieved the next time that you open your model.

*Default = Blank*

## **EditorCommandLine**

The EditorCommandLine property specifies which external editor is started when you edit code. If this

property is empty, Rational Rhapsody runs the internal editor by default. When you use this property, keep in mind the following:

- You can also use the Browse button to locate the text editor.
- The ClassCodeEditor property must be set to CommandLine for this property to take effect.

*Alternatively, if you associate your text editor with the file types .h and .cpp in Windows and set the ClassCodeEditor property to Associate, that editor is started when you edit these files. The advantage to this approach is that only one editor session is started. See the DescriptionEditor property for information on specifying the editor to use for element descriptions. (Default = empty string)*

## EnvironmentVariables

The EnvironmentVariables property specifies environment variables for Rational Rhapsody to execute when your project is opened. You can use environment variables to specify various file paths in your project.

For example, you could use environment variables to specify the location of legacy source files or the location of referenced units. This capability provides the benefit of storing your model and environment in one location (the Rational Rhapsody project) so you can more easily share and distribute complex projects.

You can use environment variables to specify various file paths in your project. For example, you could use environment variables to specify the location of legacy source files or the location of referenced units. This capability provides the benefit of storing your model and environment in one location (the Rational Rhapsody project) so you can more easily share and distribute complex projects.

Rational Rhapsody parses the content of the EnvironmentVariables property and executes the specified environment variables. This execution occurs when a project is opened and after all project-level properties are applied, but before any additional units (packages, components, and diagrams) are read.

When you open a project and all the properties overridden at the project level are applied (but before any additional units are read), Rational Rhapsody parses the contents of this property and sets the relevant environment variables. This setting affects every place Rational Rhapsody searches for an environment variable, which means that it affects the logical path. You can override this property only at the site.prp or project level.

For example, consider the case where you include legacy files in your project, and use the variable LEGACY\_DIR in your makefile to specify the location of those files. If you set the EnvironmentVariables property to include the path for LEGACY\_DIR, Rational Rhapsody executes the variable when the model is opened so the make utility can expand the LEGACY\_DIR variable. In essence, your “environment” is contained in the Rational Rhapsody model.

As another example, consider the case where you have reference units in your model (added by using the option Add to Model As Reference). You can edit the location of a reference unit by using the Directory field of the Unit Information window, and use an environment variable as part of that location.

If you set the EnvironmentVariables property to include the path of this environment variable, Rational Rhapsody parses and executes that environment variable when it opens the project, and then searches for the reference unit in the specified location. The value of this property is a MultiLine in the following format (with one variable definition per line):

Rational Rhapsody parses the content of the EnvironmentVariables property and executes the specified environment variables. This execution occurs when a project is opened and after all project-level properties are applied, but before any additional units (packages, components, and diagrams) are read.

“Execution” means iterating over the lines and setting the environment variable through the operating system API—there is no “source.”

When you open a project and all the properties overridden at the project level are applied (but before any additional units are read), Rational Rhapsody parses the contents of this property and sets the relevant environment variables. This setting affects every place Rational Rhapsody searches for an environment variable, which means that it affects the logical path.

You can override this property only at the site.prp or project level.

Variable name one=path Variable name two=path ...

For example: COMMON\_BASE=C:\SomeDirectory\SomeSubdirectory  
ANOTHER\_BASE=E:\SomeOtherDirectory\SomeOtherSubdirectory

Note: You can override this property only at the site.prp or project level.

For example, consider the case where you include legacy files in your project, and use the variable LEGACY\_DIR in your makefile to specify the location of those files. If you set the EnvironmentVariables property to include the path for LEGACY\_DIR, Rational Rhapsody executes the variable when the model is opened so the make utility can expand the LEGACY\_DIR variable. In essence, your "environment" is contained in the Rational Rhapsody model.

The following restrictions and limitations apply to this property:

- Comments are not supported.
- There is no way to “unset” the variables, other than exiting Rational Rhapsody.
- Changes in the property setting execute the next time the project is loaded.
- If you check out a different version of the . rpy file, the environment variables are reset (they are not “unset” first).
- If you read another project where this property is overridden in the project context, that setting executes on top of the previous settings (the settings are not “unset” first).
- Environment variables defined in “included” property (. prp) files are not supported.
- The value of the EnvironmentVariables property cannot be based on the value of the same property at a higher level. That is, subsequent definitions of the property cannot be based upon the previous definition. Therefore, you cannot define a \$BLK\_VAR in the site.prp file and then have a new variable \$APP\_VAR defined in the project (. rpy) file with a value of \$BLK\_VAR\SubDir, where the \$BLK\_VAR value is used from the site.prp file EnvironmentVariables definition.
- Rational Rhapsody does not expand environment variables into Rational Rhapsody generated makefiles or build files. Rhapsody does execute the variable when the model is opened and before makefile and build file generation, which enables the make or build utility to assume the responsibility of expanding the environment variable. Most but not all make and build utilities can expand environment variables.

*(Default = empty MultiLine)*

## ExcludedMetaClasses

The ExcludedMetaClasses property can be used to hide out-of-the-box element types. The availability of element types is determined by the value of this property and the value of the AvailableMetaClasses property. In the event of a conflict between the two properties, the ExcludedMetaClasses property takes precedence.

The value of the property should be a comma-separated list of metaclass names. The strings to use to represent the different metaclasses can be found in the metaclasses.txt file in the Doc directory of your Rational Rhapsody installation.

## ExcludedNewTerms

Rational Rhapsody allows you to hide any new terms that your users do not need. The availability of excluded new terms is determined by the ExcludedNewTerms property. This property is defined by using a comma-separated list of strings.

To keep all of the new terms visible, leave this property blank.

To limit the availability of certain new terms, use this property to indicate the new terms that you do not want to be available. If a NewTerm name appears in both as Available and as Excluded, it will be Excluded.

## Extension

*The Extension property specifies the extension appended to the project file name. For example, Rational Rhapsody saves the project file for the project Pager as Pager.rpy, and its repository as \Pager\_rpy. (Default = rpy)*

## ExternalImageEditorCommand

The ExternalImageEditorCommand property defines the command line to be used in order to run the user-defined image editor (ex. Microsoft Paint, Paint Shop Pro, and so on).

## FileIsSavedUnit

*The FileIsSavedUnit property determines whether new files are saved as units (separate files) by default. (Default = Cleared)*

## Filter

The Filter property contains a list of metaclasses that are filtered from the toolbars. A type that is in this list does not appear on a tool bar.

*(Default = empty string)*

## FolderIsSavedUnit

The FolderIsSavedUnit property determines whether or not new folders added to the model are automatically saved as units.

*Default = Cleared*

## GeneralElementMenuName

The GeneralElementMenuName property is for internal use only. You should not make any changes to this property unless directed by someone from IBM Rational Rhapsody.

*Default = General Elements*

## HelpersFile

The HelpersFile property can be used to associate a .hep file with a model. You can type in the full path of the .hep file or you can use the "..." button to select the .hep file. .hep files are used to store the details of helper applications that have been developed to facilitate working in Rational Rhapsody.

Note that if you specify a .hep file by using this property, Rational Rhapsody does not recognize the helper applications defined in the profile-specific .hep file if one is provided for the profile you are using.

*(Default = Blank)*

## HideToolbars

The HideToolbars property can be used to have certain toolbars initially hidden when a project is opened. This can be useful for profiles where you don't want to display toolbars that users are unlikely to need.

The value of the property should be a comma-separated list of toolbar names, using the strings listed in the View > Toolbar menu.

## HighlightElementsInActiveComponentScope

When this property is Checked, elements within the scope of the active component and configuration are highlighted as bold in the browser.

*(Default = Cleared)*

## HintMessageForPerspective

If you are using the user-perspective feature to hide specific options in the user interface for different types of users, and you are using the ShowHintForPerspectiveName property to show a message for "limited" perspectives, you can use the property HintMessageForPerspective to customize the message



that is displayed to the user.

If you do not use the `HintMessageForPerspective` property to provide a custom message, Rhapsody uses a default message.

*Default = Blank*

## ImageEditor

Defines which image editor to use when opening an associated image. Available values are as follows:

- `AssociatedApplication` - Allows the OS choose according to the extension (Default)
- `External` - Uses user-defined editor

If the `ImageEditor` property is set to `External`, then the `ExternalImageEditorCommand` property must be defined.

## LocateInModelDependencyRecursionLimit

When you use the `Locate in Model` feature to find the model element represented by an element name in a snippet of code in the `Features` dialog, the context that is searched includes model elements that are related through dependencies, even elements related through recursive dependencies.

To prevent situations where such recursivity results in long delays, you can use the property `LocateInModelDependencyRecursionLimit` to limit the number of levels of recursion that are taken into account.

The first level is considered to be the model element itself, so if you want to limit the list to direct dependencies, you would enter 2 as the value of the property.

If you don't want to limit the level of recursion, set the value of the property to 0.

*Default = 0 (default value is 2 when using Rhapsody in Java)*

## MergeElementsAPIPolicy

The property `MergeElementsAPIPolicy` is used to control the behavior of the Rhapsody API operation `IRPApplication.mergeElements`.

If set to `All`, the merge process will both add and delete information.

If set to `NoDelete`, the merge process will add information but not delete information.

In addition, if you are importing ARXML into a Rhapsody model, the `MergeElementsAPIPolicy` property works in combination with the `AUTOSAR::ARXML::MergeImportPolicy` property. While the `MergeImportPolicy` property defines the "preferred" side to choose if conflicts are found during a merge, the `MergeElementsAPIPolicy` property sets what elements remain in the model after the merge.

To see a table that shows combination possibilities and results when MergeElementsAPIPolicy and MergeImportPolicy are set, see the online help. Do a search for MergeElementsAPIPolicy or MergeImportPolicy.

- All - Anything that is not in the imported ARXML is omitted or deleted from the model.
- NoDelete - Elements that are already in the model and elements that are in the ARXML will remain in the model.

*Default = NoDelete*

## ModelCodeAssociativityFineTune

The ModelCodeAssociativityFineTune property changes the default DMCA mode in the site.prp file. However, you usually set this property by using the GUI (by selecting Code > Dynamic model code associativity). The possible values are as follows:

- Bidirectional - Both code generation and round trip are started automatically for the code view window.
- Roundtrip - Only roundtrip is started automatically in the online code view windows.
- Code Generation - Only code generation is started automatically in the online code view windows.
- None - Disables DMCA entirely. The online code view windows become simple text editors.

*(Default = Bidirectional)*

## NamesForbiddenCharacters

The NamesForbiddenCharacters property is used to specify characters that cannot be used in the names given to the following types of elements:

- Actors
- Configurations
- Dependencies
- Diagrams (all types)
- Flows
- Links
- Requirements
- Statecharts
- Stereotypes
- Use cases

The names permitted for other types of elements are limited by the property NamesRegExp. See the property description for NamesRegExp for the exact list of elements that are affected by that property.

*Default = \[\*?'"`{}[]()<>\/\n*

## NamesRegExp

The NamesRegExp property specifies the regular expression that limits the format of the name that can be used for a model element. For example, using the default value of the property, you could call an element Class1 but not 1Class.

Note that the property NamesRegExp affects only the following types of elements:

- Arguments (for operations and events)
- Attributes
- Classes
- Events
- Global variables
- Objects
- Operations (all types)
- Projects
- Transitions
- Types

*Default =  $\wedge([a-zA-Z_][a-zA-Z0-9_]*)(operator.+))\$$*

## NotifyPluginOnElementChange

If you have an application that extends the RPApplicationListener class, Rhapsody will notify the application when model elements are changed. Similarly, if you have added an OnElementsChanged method to a Rhapsody plug-in, the plugin will be notified when model elements are changed.

Because these notifications results in a large degree of overhead, the property NotifyPluginOnElementChange was added with a default value of False.

Your application or plug-in will be notified of element changes only if you change the value of the property to True.

In the case of plugins (which don't extend the RPApplicationListener class), you also must change the value of the property NotifyPluginOnElementChangeForApplicationListenerOnly to False in order to receive notifications.

For both of these properties, if you modify the property value, you must reload the model in order to have the new value take effect.

Note that you can also turn on these notifications by calling the method IRPProject.setNotifyPluginOnElementsChanged in your application.

*Default = False*

## NotifyPluginOnElementChangeForApplicationListenerOnly

If you have an application that extends the `RApplicationListener` class, Rhapsody will notify the application when model elements are changed. Similarly, if you have added an `OnElementsChanged` method to a Rhapsody plug-in, the plug-in will be notified when model elements are changed.

Because these notifications results in a large degree of overhead, your application or plug-in will be notified of element changes only if you change the value of the property `NotifyPluginOnElementChange` to `True`.

In the case of plugins (which don't extend the `RApplicationListener` class), you also must change the value of the property `NotifyPluginOnElementChangeForApplicationListenerOnly` to `False` in order to receive notifications.

For both of these properties, if you modify the property value, you must reload the model in order to have the new value take effect.

*Default = True*

## ObjectIsSavedUnit

*The `ObjectIsSavedUnit` property determines whether new packages are saved as units (separate files) by default. (Default = Cleared)*

## OutputWindowFont

*The `OutputWindowFont` property specifies the font used for messages displayed in the Output window. (Default = Courier New 9 NoBold NoItalic)*

## PackagesSavedUnit

*The `PackageIsSavedUnit` property determines whether new packages are saved as units (separate files) by default. (Default = Checked)*

## PathInProjectList

When a project is added to a project list, the path to the project is added to the project list file (.rpl). The `PathInProjectList` property can be used to specify whether an absolute or relative path should be used when the project is added to a project list.

The possible values for the property are `Absolute` and `Relative`.

Note that when you change the value of this property for a project after the project has already been included in project lists, you have to open the relevant project lists in Rational Rhapsody and select `Save All` in order to update the project path in the project list files.

*Default = Absolute*

## Perspectives

If you are using the perspectives feature to filter the types of elements that users can create, use the Perspectives property to list the names of the different perspectives that will be included in the Perspective toolbar.

The value of the property should be a comma-separated list of strings. For example, the SysMLPerspectives settings use the value:  
Novice,Intermediate-Requirements-Engineer,Intermediate-Architect,Intermediate-Trade-Analyst,Advanced

When the value of this property is left blank, the Perspective toolbar is not displayed.

*Default = Blank*

## PredefinedTypesInComboBox

The PredefinedTypesInComboBox property is a comma separated list of the predefined types included in the types combo box. The PredefinedTypesInComboBox property only affects predefined types. Types from other packages are not affected.

*(Default = empty string)*

## PredefinedTypesToExclude

The property PredefinedTypesToExclude is used to specify a number of predefined types that should not be displayed in the browser or in the list of types. This property is for internal Rhapsody use, so its value should not be modified.

*Default =*

*SearchRelRefAmountTypeEnum,SearchRelRefHowRelatedEnum,SearchTypeEnum,SearchUnresolvedEnum,MetricsChartType*

## RefactorRenameRegExp

This property is used as part of the regular expression string to search for user code instances of the element you are renaming. If you are renaming an attribute, the program needs to find any instances in the user code (such as the operation of a body, action on entry/exit, reaction in a station, overridden properties, and configuration initialization).

The program only inspects the user code because everything else (dependencies and other relations/references) are automatically updated upon renaming the element. However, this feature also performs some additional refactoring. Therefore, the program searches for all instances of the element in user code and shows them in the preview window to the user.

For example, if a user created functions related to a changed attribute and if the new name of the attribute is "attribute\_0123," then the program renames those instances in user code to "get\_attribute\_0123" and

"set\_attribute\_0123" with the "get\_" and "set\_" prefixes used as they are set by default in this property.

The user might change these default settings, but they need to be compatible with the regular expression syntaxes of both the Rational Rhapsody search and replace, as well as the internal code editor.

*Default = (\$keyword)/((([?]\*\_)/(its))\$keyword)/(\$keyword\_([?]\*))*

## ReferenceUnitPath

The ReferenceUnitPath property defines how to save a reference unit path.

The property can be set to "Absolute" or "Relative" to specify whether units that are added to the model by reference to use an absolute path or by the relative path.

If the property is set to "Relative," then newly added referenced units contain a path relative to the project directory.

Note:

The correct way to change to a relative path is to set this property and then add the unit to the model again.

If the ReferenceUnitPaths property is set to "Absolute" when the unit is loaded, then Rational Rhapsody continues to expect an "Absolute" path when the model is loaded. Editing the path of the unit to change it from "Absolute" to "Relative" does not work.

## RenameUnusedFiles

By default, when you use "Delete from Model" to delete a unit in Rational Rhapsody, the element is removed from the browser but the unit file remains in the project directory. This is also true for actions such as rename and move. The RenameUnusedFiles property is a Boolean value that allows you to specify that Rational Rhapsody should add an additional file extension to the names of files that remain in the project directory after one of these actions.

To use this feature, set the value of this property to Checked.

Use of this feature makes it easy to identify the unused files in the file system if you would like to delete them at some stage.

By default, the extension added when the property is set to Checked is ".keep". This extension can be changed by modifying the value of the General::Model::RenameUnusedFiles propertyWith.

*Default = Cleared*

## RenameUnusedFilesWith

If the General::Model::RenameUnusedFiles property is set to True, then Rational Rhapsody adds an additional file extension to the names of files that remain in the file system after actions such as rename and "delete from model" in Rational Rhapsody. The RenameUnusedFilesWith property allows you to

specify the extension that you would like Rational Rhapsody to use for this feature.

*Default = .keep*

## ReservedWords

The ReservedWords property is a string that specifies the list of Rational Rhapsody reserved words. Reserved words cannot be used as names of classes, attributes, and so on. To specify additional reserved words for your environment, use the property lang\_CG::Environment::AdditionalReservedWords.

The default value is as follows:

asm auto bad\_cast bad\_typeid break case catch char class const const\_cast continue default delete do double dynamic\_cast else enum except extern finally float for friend goto if inline int long namespace new operator private protected public register reinterpret\_cast return short signed sizeof static static\_cast struct switch template this throw try type\_info typedef typeid union unsigned using virtual void volatile while xalloc

## SAExternalID

This read-only property displays the identity in the encyclopedia of the Rational System Architect imported elements .

## SAExternalType

This read-only property displays the type in the encyclopedia of the Rational System Architect imported elements.

## SAPropertyValue

This read-only property relates to importing Rational System Architect data into Rational Rhapsody. When creating an element in Rational Rhapsody that is representing an imported Rational System Architect property, this Rational Rhapsody property holds the corresponding value from Rational System Architect.

## SaveBeforeCodeGeneration

The SaveBeforeCodeGeneration property determines whether a model is to be saved before code generation. Note that this behavior does not apply for the code generation "Selected classes" command.

*Default = Cleared*

## SearchPath

*The SearchPath property is currently unused. (Default = .)*

## SelectedPerspective

If you are using the perspectives feature to filter the types of elements that users can create, the SelectedPerspective property is used to store the value of the currently selected perspective.

You can use this property to specify the perspective that should be used when Rhapsody is opened. The value of the property should be the relevant perspective name from the list defined with the property General::Model::Perspectives

## ShowHintForPerspectiveName

The user-perspective feature can be used to hide specific options in the user interface for different types of users. You can use the property ShowHintForPerspectiveName to have a message displayed when a model uses a "limited" perspective, so that there is no confusion as to why certain options are not available.

The value of the property should be a comma-separated list of names of user-perspectives that you have defined. These are the perspectives for which the message will be displayed. For example, in the SysMLPerspectives settings, the value of the property is set to "Basic,Intermediate-Requirements-Engineer,Intermediate-Architect,Intermediate-Trade-Analyst".

*Default = Blank*

## ShowHoveringToolbar

Beginning in release 8.2.1, in addition to the standard Drawing toolbar, Rhapsody provides a hovering Drawing toolbar, which is displayed near the current location of the cursor on the drawing canvas. This toolbar is displayed when you click the canvas or a diagram element with the left mouse button.

You can disable this feature temporarily by selecting View > Toolbars > Hovering Drawing Toolbar from the main menu.

If you want to disable the hovering toolbar for a specific project, you can change the value of the property General::Model::ShowHoveringToolbar to False. You will have to re-open the project to have the new value take effect.

If you want to disable the hovering toolbar for all projects, you can change the value of ShowHoveringToolbar in your site.prp file. You will have to restart Rhapsody to have the new value take effect.

*Default = True*

## ShowModelTooltipInBrowser

This property determines how tooltip content is displayed in the Rational Rhapsody browser.

- Simple: uses the default settings for tooltip content
- Enhanced: uses custom settings for tooltip content



*Default = Enhanced*

## **ShowModelTooltipInDescription**

This property determines how tooltip content is displayed in the Description tab of the Features window.

- Simple: uses the default settings for tooltip content
- Enhanced: uses custom settings for tooltip content

*Default = Enhanced*

## **ShowModelTooltipInFeaturesDialog**

The property ShowModelTooltipInFeaturesDialog controls whether tooltips are displayed in the Features dialog for elements where such tooltips are supported (for example, the Element Types & Criteria tab of the Features dialog for Table Layouts).

The possible values are:

- Simple - tooltip is not displayed
- Enhanced - tooltip is displayed

*Default = Simple*

## **ShowModelTooltipInGE**

This property determines how tooltip content is displayed in the graphical editors toolbar.

- Simple: uses the default settings for tooltip content
- Enhanced: uses custom settings for tooltip content

*Default = Simple*

## **ShowModelTooltipInGETab**

This property determines how tooltip content is displayed in each editable tab of the Features window.

- Simple: uses the default settings for tooltip content
- Enhanced: uses custom settings for tooltip content

*Default = Simple*

## **ShowModelTooltipInGETabMenu**

The property ShowModelTooltipInGETabMenu controls whether additional information is displayed when you hover over an item in the menu that lists the open tabs.

The possible values are:

- Simple - tooltip is not displayed
- Enhanced - tooltip is displayed

*Default = Enhanced*

## **ShowModelTooltipInGrid**

This property determines how tooltip content is displayed in a free form diagram.

- Simple: uses the default settings for tooltip content
- Enhanced: uses custom settings for tooltip content

*Default = Enhanced*

## **ShowModelTooltipInMenu**

The property ModelTooltipTemplateHTML allows you to define a template for displaying detailed model element information in tooltips. A number of properties are provided to allow you to show/hide these enhanced tooltips in different contexts.

The property ShowModelTooltipInMenu is used to determine whether such enhanced tooltips should be displayed for "new terms" that are included in pop-up menus. Set the property to Enhanced to have these enhanced tooltips displayed. Set the property to Simple in order to prevent these tooltips from being displayed.

*Default = Enhanced*

## **ShowModelTooltipInSearchResult**

This property determines how tooltip content is displayed in the search results window.

- Simple: uses the default settings for tooltip content
- Enhanced: uses custom settings for tooltip content

*Default = Enhanced*

## **ShowPotentialUnresolvedReferences**

If you delete a model element that is referenced by other elements that are read-only, Rational Rhapsody displays a window listing the relevant read-only files so that you can change them to read/write. This is designed to prevent situations where element deletions result in unresolved references.

If you do not want Rational Rhapsody to display this window, you can set the value of the ShowPotentialUnresolvedReferences property to Cleared.

*Default = Checked*

## ShowUnitChangeNotification

When the property ShowUnitChangeNotification is set to True, if the file for a loaded model element is modified by a different program, Rhapsody notifies you of the change, and asks you if you want to refresh the relevant information in Rhapsody.

*Default = True*

## SourceFont

The SourceFont property determines which font is used for source code in the browser and graphic editor windows. You can use only fonts that are actually installed on your system. For example, Courier is a fixed-size font that is available only in certain sizes, but not the 5-point size.

However, Courier New is a TrueType font, which can be used in any size-integer or floating point because it is provided in vector format rather than as a bitmap.

*To see which sizes are available on your system, click Choose Font in the specification window for the SourceFont property. Available fonts are listed in the resulting Font window. Note that italics and bold are ignored. (Default = Courier New 9 NoBold NoItalic)*

## Submenu1List

The Submenu1List property controls which elements populate the menu for the Submenu1Name property. By default, Submenu1 concerns diagrams. Therefore, by default, the default values for Submenu1List are the diagrams available in Rational Rhapsody. You can re-order, remove, or re-add any of these elements by doing so through the Submenu1List property.

Note the following:

- Whatever element that is removed from Submenu1List is displayed in the middle portion of the Add New menu if that element is relevant for your project. The element must appear somewhere if it is a valid element.
- The Submenu1List and Submenu1Name properties are also used by Tools > Diagrams. When you make a change to Submenu1List, to have it take effect on the Tools menu, you must save your project, close it, and then open it again. In addition, if you delete the Submenu1 value from the SubmenuList property, all the Rational Rhapsody diagram choices appear in the Tools menu, instead of under Tools > Diagrams (after you save your project and then re-open it again).
- The General::Model::AddNewMenuStructure property overrides this property.

*Default = Class*

*Diagram, ObjectModelDiagram, SequenceDiagram, UseCaseDiagram, ComponentDiagram, DeploymentDiagram, Collaborati*

## Submenu1Name

The Submenu1Name property controls the name that displays for the Submenu1 group in the Add New menu. Use this property in conjunction with Submenu1List to populate the elements that should appear for the Submenu1 group. The SubmenuList property controls whether Submenu1Name is displayed on the

Add New menu.

Note the following:

- The Submenu1List and Submenu1Name properties are also used by Tools > Diagrams. When you make a change to Submenu1List, to have it take effect on the Tools menu, you must save your project, close it, and then open it again. In addition, if you delete the Submenu1 value from the SubmenuList property, all the Rational Rhapsody diagram choices appear in the Tools menu, instead of under Tools > Diagrams (after you save your project and then re-open it again).
- The General::Model::AddNewMenuStructure property overrides this property.

*Default = Diagrams*

## Submenu2List

The Submenu2List property controls which elements populate the menu for the Submenu2Name property. By default, Submenu2 concerns relations. Therefore, by default, the default values for Submenu2List are the relations available in Rational Rhapsody. You can re-order, remove, or re-add any of these elements by doing so through the Submenu2List property.

Note the following:

- Whatever element that is removed from a group is displayed in the middle portion of the Add New menu if that element is relevant for your project. The element must appear somewhere if it is a valid element.
- The General::Model::AddNewMenuStructure property overrides this property.

*Default = Dependency, Derivation, Flow, AssociationEnd, Generalization, Realization, Hyperlink*

## Submenu2Name

The Submenu2Name property controls the name that displays for the Submenu2 group in the Add New menu. Use this property in conjunction with Submenu2List to populate the elements that should appear for the Submenu2 group. The SubmenuList property controls whether Submenu2Name is displayed on the Add New menu.

Note: The General::Model::AddNewMenuStructure property overrides this property.

*Default = Relations*

## Submenu3List

The Submenu3List property controls which elements populate the menu for the Submenu3Name property. By default, Submenu3 concerns tables and matrices. Therefore, by default, the default values for Submenu3List are the elements available in Rational Rhapsody for tables and matrices. You can re-order, remove, or re-add any of these elements by doing so through the Submenu3List property.

Note the following:

- Whatever element that is removed from a group is displayed in the middle portion of the Add New menu if that element is relevant for your project. The element must appear somewhere if it is a valid element.
- The General::Model::AddNewMenuStructure property overrides this property.

*Default = TableLayout,TableView,MatrixLayout,MatrixView*

## Submenu3Name

The Submenu3Name property controls the name that displays for the Submenu3 group in the Add New menu. Use this property in conjunction with Submenu3List to populate the elements that should appear for the Submenu3 group. The SubmenuList property controls whether Submenu3Name is displayed on the Add New menu.

Note: The General::Model::AddNewMenuStructure property overrides this property.

*Default = Table\Matrix*

## Submenu4List

The Submenu4List property controls which elements populate the menu for the Submenu4Name property. By default, Submenu4 concerns annotations. Therefore, by default, the default values for Submenu4List are the elements available for annotations/requirements in Rational Rhapsody. You can re-order, remove, or re-add any of these elements by doing so through the Submenu4List property.

Note the following:

- Whatever element that is removed from a group is displayed in the middle portion of the Add New menu if that element is relevant for your project. The element must appear somewhere if it is a valid element.
- The General::Model::AddNewMenuStructure property overrides this property.

*Default = Constraint,Component,ControlledFile*

## Submenu4Name

The Submenu4Name property controls the name that displays for the Submenu4 group in the Add New menu. Use this property in conjunction with Submenu4List to populate the elements that should appear for the Submenu4 group. The SubmenuList property controls whether Submenu4Name is displayed on the Add New menu.

Note: The General::Model::AddNewMenuStructure property overrides this property.

*Default = Annotations*

## SubmenuList

The SubmenuList property controls which submenu groups appear at the bottom portion of the Add New

menu. Use this property in conjunction with its corresponding Submenu#Name and Submenu#List properties (for example, Submenu1Name and Submenu1List).

For example, if you remove the "Submenu1," value, which is related to diagrams (see the Submenu1Name and Submenu1List properties); then the "Diagrams" group does not appear on the Add New menu.

Note the following:

- Removing the "Submenu1" value has an effect on Tools > Diagrams. After doing so, after you save your project and then open it again, all the Rational Rhapsody diagram choices appear in the Tools menu, instead of under Tools > Diagrams (which is what happens when the "Submenu1" value is set in the SubmenuList property).
- The General::Model::AddNewMenuStructure property overrides this property.

*Default = Submenu1,Submenu2,Submenu3,Submenu4*

## SupportIncrementalModelSynchronization

The SupportIncrementalModelSynchronization property specifies whether you want to save \_modifiedTimeWeak for all elements in order to support model synchronization.

*Default = Checked*

## TrackChangesAutoRun

If you want the track changes feature to be turned on automatically any time that a specific project is opened, set the value of the property TrackChangesAutoRun to Normal or Minimized for the project. Note that changes to the project are tracked only if the track changes window is open or minimized.

- Normal - the track changes window will be opened when the project is opened
- Minimized - the track changes window will be opened and then automatically minimized when the project is opened
- None - the track changes window will not be opened automatically

*Default = None*

## TypeComboBoxSort

The TypeComboBoxSort property determines how the Type ComboBox is sorted. Listed values can be sorted one of two ways: Alphabetically or ByPackage. If the listed values are sorted ByPackage, the contents of the list are sorted by their respective packages and within each package hierarchy, the types are sorted alphabetically.

## UndoBufferSize

*The UndoBufferSize property is an integer that specifies how many undo transactions are remembered by Rational Rhapsody. A value of "0" means no undo transactions are remembered, "1" means one undo transaction is remembered, and so on. (Default = 20)*

## UnresolvedSymbol

By default, Rational Rhapsody displays "(U)" next to unresolved model elements in the browser and in diagrams.

The UnresolvedSymbol property allows you to specify a different symbol to use to indicate unresolved model elements. Just enter the string that you would like Rational Rhapsody to use.

*Default = (U)*

## UseIncrementalSave

By default, the Rational Rhapsody Save option saves only the units that have been modified. The UseIncrementalSave property allows you to specify that the Save option should save the entire model.

To have Rational Rhapsody save the entire model, set this property to Cleared.

*Default = Checked*

## UseTimestampInDefaultAnnotationName

When this property is set to True, the integer count appended to a default name for a Constraint, Comment or Requirement element (Constraint\_4, for example) is replaced with a unique time stamp. By using the time stamp in a collaborative environment, you can avoid the possibility of two elements being checked in to source control with the same name. In these situations, the DiffMerge utility attempts to merge the two elements.

*Default = Cleared*

## VolatileProperties

The VolatileProperties property can be used to list properties whose value should not be saved in the model when the value is modified.

The value of the property should be a comma-separated list of full property names, using the format Subject::Metaclass::Property.

## WarnForDuplicates

The WarnForDuplicates property specifies whether Rational Rhapsody should issue a warning message when you add an element to the model with a name that is identical to an already existing name for the same kind of element.

Within a given project, you can have two packages with the same name, or two classes or objects with the same name (for example, P1::p and P2::p), provided they are in different scopes.

*(Default = Checked)*

## ModelLibraries

The metaclass ModelLibraries contains properties related to reference models such as the Java reference model.

### JavaAPIPackage

The JavaAPIPackage property is used to specify the location of the Java reference model that is included with Rational Rhapsody. This is a model of the classes contained in Java SE 6. Rhapsody uses the property when you select the "Add Java API Library" item from the File menu.

*Default = \$OMROOT/LangJava/JDKRefModel/JDKModel\_rpy/java.sbs*

## Operation

Contains properties related to the display of operation information.

### SignaturePluginFunction

The property SignaturePluginFunction can be used to specify the name of a plugin method that returns a string which should be displayed in the Features dialog instead of the default signature displayed by Rhapsody for operations.

The value of the property should be the name of a plugin method.

*Default = Blank*

## Profile

The Profile metaclass contains properties that specify the behavior of profiles.

### AutoCopied

*The AutoCopied property specifies a comma-separated list of physical paths of profiles that are automatically copied into a newly created project (by using Add To Model with As Unit). (Default = empty string)*



## AutoReferences

For any profile, you can specify a list of additional profiles that should be added automatically as references when you add the primary profile to a model. If you want these additional profiles added with their subunits, use the property `AutoReferences`. If you want the other profiles added without their subunits, use the property `AutoReferencesWithoutSubUnits`. The value of the property should be a comma-separated list of the paths to the profiles that should be added.

*Default = Blank*

## AutoReferencesWithoutSubUnits

For any profile, you can specify a list of additional profiles that should be added automatically as references when you add the primary profile to a model. If you want these additional profiles added without their subunits, use the property `AutoReferencesWithoutSubUnits`. If you want the other profiles added with their subunits, use the property `AutoReferences`. The value of the property should be a comma-separated list of the paths to the profiles that should be added.

*Default = Blank*

## Project

The Project metaclass contains properties that specify the behavior of projects.

## AddToModelDefaultPolicy

When you use the "Add to Model" feature, the dialog that is displayed provides three options: As Reference, As Unit (without copy into model), and As Unit (with copy into model). The `AddToModelDefaultPolicy` property can be used to specify which of these options should be selected by default when the dialog is displayed.

*Default = AsReference*

## ConditionalPropertyCyclicValue

When using the conditional property feature, if there is a cyclical relationship between the values of two or more properties, the conditional expression cannot be evaluated. In order to prevent such problems, Rhapsody checks for cyclical relationships between properties. When Rhapsody encounters such problematic use of conditional properties, it does not try to evaluate the expression but instead returns the string specified as the value of the property `ConditionalPropertyCyclicValue`.

For example, if the expression is "`?<IsConditionalProperty>some text $<C_CG::Package::ImplementationEpilog> some more text`", and the property `C_CG::Package::ImplementationEpilog` has a cyclical relationship with other properties, Rhapsody will

return "some text {Cyclic Property Reference for \$<property>} some more text".

*Default = {Cyclic Property Reference for \$<property>}*

## ConditionalPropertyPrefix

Rhapsody allows you to make the value of a property dependent upon the value of one or more other properties. This feature is described in the help topic titled "Conditional Properties".

The property ConditionalPropertyPrefix can be used to specify the prefix that should be used at the beginning of the value of a property to indicate that the property is to be interpreted as a conditional property.

Note that the value of this property is read when a project is opened. If you change the value, you must close and then reopen the project in order for the new value to take effect.

*Default = ?<IsConditionalProperty>*

## UnitSetAsReadOnlyUnitsAddedOnAddWithoutCopy

The property UnitSetAsReadOnlyUnitsAddedOnAddWithoutCopy allows you to specify that existing units added to a model should be set to read-only if they are added to the project without being copied to the project location.

*Default = False*

## Relations

The Relations metaclass contains a property that controls the default multiplicity of relations.

### DefaultMultiplicity

*The DefaultMultiplicity property specifies the default multiplicity for relations for which the multiplicity is not specified. (Default = 1)*

## Report

The Report metaclass contains properties that control the attributes of the Rational Rhapsody internal reporter.

## ExternalViewerCommand

The ExternalViewerCommand property supplies the command to use to run an external RTF viewer. The command must take the format: "<executable name>" "\$fileName"

This property is only used if the ReportViewer property is set to "External."

## ReportViewer

The ReportViewer property specifies which RTF viewer to use in order to show the generated "Report on model" RTF document. Available options are as follows:

- Rhapsody - Uses the Rational Rhapsody internal view
- Associated - Allows the OS to choose the right viewer according to the file extension
- External - Uses the command line set in the ExternalViewerCommand to run an external viewer program

*(Default = Rhapsody)*

## RTFCharacterSet

The RTFCharacterSet property defines the necessary character set used by the RTF format file created by the Rational Rhapsody internal reporter.

The character set is used in the RTF multilanguage and description styles, which are used for the Name Label and Description fields of the report. The RTF file created by the Rational Rhapsody internal reporter should include a specific character set for each language.

For example, set this property to “\fcharset128” for Japanese.

*The default value, an empty string, preserves the current behavior. You can define this property on the project and higher (site or factory) level. (Default = empty string)*

## ReporterPLUS

The ReporterPLUS metaclass contains properties that control the behavior of Rational Rhapsody ReporterPLUS.

## ReportAll

The ReportAll property is used by the ReporterPLUS tool. Do not change the value of this property.

The default value is as follows:

"\$OMROOT/../../Reporter/Reporter.exe" /m "\$modelname" /l "reg")

## ReportSelected

The ReportSelected property is used by the ReporterPLUS tool. Do not change the value of this property.

The default value is as follows:

"\$OMROOT/../../Reporter/Reporter.exe" /m "\$modelname" /s "\$scope" /l "reg"

## SplitDiagrams

The SplitDiagrams property specifies whether to split large reports across pages when they are exported (as metafiles) to other tools, such as the Rational Rhapsody internal reporter or the API.

By splitting large diagrams across multiple pages, you improve their readability.

This feature does not apply to the following operations:

- Printing diagrams
- Copying diagrams into the clipboard
- Pasting them from the clipboard

It applies only to exporting diagrams (as metafiles) to other tools. The possible values are as follows:

- True - Divide the diagram vertically or horizontally as needed when exporting. The diagram is split if fitting the diagram onto a single page requires a zoom factor of 65% or lower.
- False - Zoom out as necessary to fit the diagram on a single page. This is the default behavior for Rational Rhapsody Version 4.0 and its point releases.

If this property is set to True, the diagram is first split by column (top to bottom), then by row. The following figure shows the order in which the pages are created.

For example, the following figure shows a sequence diagram forced onto a single page. The following figures show how this sequence diagram would be split into multiple pages. The following sections describe implementation-specific behavior.

When a sequence diagram is split into multiple pages, the names of instances (the upper pane of the SD) is added to the top of each page. Rational DOORS If you have opened a diagram and want to see all of the pages, select Edit OLE Object Document Object Open.

API In previous versions of Rational Rhapsody, you exported a diagram in one metafile by using the following call:

HRESULT getPicture ([in] BSTR fileName); Version 4.1 introduces a new method, getPictureAsDividedMetafiles. The syntax is as follows: HRESULT getPictureAsDividedMetafiles ( [in] BSTR firstFileName, [out, retval] IRPCollection\*\* fileNames); In the call, firstFileName specifies the naming convention for the created files.

For example, if you passed the value “Foo” as the firstFileName:

- If the diagram can be drawn on one page, the name of the metafile is Foo.
- If the diagram is split into multiple pages, the first file is named FooZ\_X\_Y. The variables used in the name are as follows:
- Z - The number of the created file
- X - The number of the page along the X vector
- Y - The number of the page along the Y vector
- For example, the file Foo2\_1\_2 means that this is the second metafile created and it contains one page, which is the second page along the Y vector (the X vector is 1).

*All the file names is inserted in the sent strings list ( fileNames). (Default = Checked)*

## TemplateEditor

The TemplateEditor property is used by the ReporterPLUS tool. Do not change the value of this property.

The default value is as follows:

```
"$OMROOT/../../Reporter/Reporter.exe" /l "pro"
```

## RPE

The RPE metaclass contains properties that control the settings for Rational Publishing Engine in Rational Rhapsody.

## Invocation

The Invocation property contains the path of Rational Publishing Engine Launcher. This property is used while invoking Rational Publishing Engine Launcher from Rational Rhapsody.

## InvokeParams

The InvokeParams property specifies the parameters that should be passed to Rational Publishing Engine Launcher

## LaunchXML

The LaunchXML property specifies an XML file used to set the default parameters in the Rational Publishing Engine report wizard.

## ReportGeneratorInvocation

The ReportGeneratorInvocation property specifies the path of Rational Rhapsody Report Generator.

## ReportGeneratorParams

The ReportGeneratorParams property specifies the parameters that can be passed to Rational Rhapsody Report Generator.

## ReportImageFormat

The property ReportImageFormat can be used to specify the graphic format that should be used for images when an RPE report is generated for the model.

*Default = JPG*

## TableView

The TableView metaclass contains properties that control table layouts and views.

## ExtendedTableLayoutMetaClasses

The ExtendedTableLayoutMetaClasses property defines a list of metaclasses with which to filter the content of the Layout list on the General tab of the Features window for a table view.

If the value for the property is empty, no filtering is done and all layout choices appear in the list.

*Default = TableLayout,ExtendedTableLayout*

## ExtendedTableLayout

The ExtendedTableLayout property enables extended table layouts.

*Default = Checked*

If the value is set to Cleared, then the regular table layout is enabled instead.

## ExtendedTableView

The ExtendedTableView property enables extended table views.

*Default = Checked*

If the value is set to Cleared, then the regular table view is enabled instead.

## ScopeMetaClasses

The ScopeMetaClasses property defines a list of the metaclasses with which to filter the content of the Scope list on the General tab of the Features window for a table view.

If the value of the property is empty, no filtering is done and all elements based on the project and the packages in the model appear in the list.

*Default = CompositionTypeee*

## Visibility

The Visibility metaclass contains properties related to menu visibility.

## CustomizeMenusFile

If you want to customize the options that are available from the main menu for a given project, you can use the property CustomizeMenusFile to point to an XML file that defines the menu items to include.

For example, the SysMLPerspectives settings includes customized menu files such as Basic\_customize\_menu.xml and Advanced\_customize\_menu.xml.

The value of the property CustomizeMenusFile can use the conditional property syntax. In the SysMLPerspectives settings, the value of this property is:

?<IsConditionalProperty>\$OMROOT/Settings/SysMLPerspectives/\$<General::Model::SelectedPerspective>\_customize\_me

*Default = Blank*

## Workspace

The Workspace metaclass contains properties that control the behavior of the Rational Rhapsody workspace.

## AnimationOutputBufSize

*The AnimationOutputBufSize property specifies the size, in bytes, of the output buffer used by animation. (Default = 65536 bytes)*

## ApplyHiddenSubjects

This property allows the user to hide subjects specified in the "HiddenSubjects" property. However, when the user is running the Rational Rhapsody Modeler or Rhapsody Corporate, this property is ignored.

*(Default = Cleared)*

## DoubleClickOnRelationsShould

The DoubleClickOnRelationShould specifies what action to perform when double-clicking on an item in the Relations window. Possible values are as follows:

- OpenFeatures - Open the Features Dialog of the item.
- LocateTheElement - Highlight the item in the browser. If the item is a diagram, the diagram is opened.
- DoBoth - Open the Features Dialog Box and highlight the item in the browser.

## GenerateNameFromLabelInLabelMode

This property indicates whether the element name should be generated from its label when working in the "Label" mode. Label mode can be set by selecting the View > Label Mode menu option.

*(Default = Checked)*

## HiddenSubjects

This is a comma separated list of subjects that need to be hidden from display in the features window. The specified subjects are only hidden if the "ApplyHiddenSubjects" property is set to Checked.

However, when the user is running the product Modeler or Rhapsody Corporate, the value of the "ApplyHiddenSubjects" property is ignored and by default all of the subjects specified in this list are hidden in the Features window.

## OkMayDockFeatures

The OkMayDocFeatures property specifies whether the features window is set to Show Mode ( Checked). In Show Mode, if the features window is docked and you double-click an element (in either the browser or drawing area), the window floats. Pressing OK docks it again instead of closing it.

*(Default = Cleared)*

## OpenDiagramWithLastPlacement

The OpenDiagramWithLastPlacement property is a Boolean value that determines whether Rational Rhapsody displays your diagrams by using the last values of the following diagram properties:



- Size
- Position (relative to the upper, left-hand corner)
- Status (maximized, minimized, and so on)
- Zoom factor
- Scroll location

*(Default = Checked)*

## **OpenWindowsWhenLoadingProject**

The OpenWindowsWhenLoadingProject project is a Boolean value that determines whether Rational Rhapsody should load the window configuration information saved from a previous session.

Rational Rhapsody saves the window configuration for a user for a given project in the workspace file (.rpw) each time a project is closed. The information saved includes window size, position, status of feature windows, and the scaling or zoom factor of open diagrams.

To prevent Rational Rhapsody from loading that information the next time the project is opened, set this property to Cleared.

This property used to be called General::Workspace::OpenWorkspaceWhenLoadingProject. It was changed because workspaces now store information on loaded units (for the partial load feature), as well as window preferences. This property affects only the windows.

*(Default = Checked)*

## **OpenWorkspaceWhenLoadingProject**

The OpenWorkspaceWhenLoadingProject project (under General::Workspace in Rational Rhapsody Developer for C and J) is a Boolean value that determines whether Rational Rhapsody should automatically load the workspace when it loads the project.

*(Default = Checked)*

## **ShowLabelInFeaturesDialog**

*The property indicates whether the Features window should display the Label instead of the Name.*  
*(Default = Cleared)*

# IntelliVisor

---

The IntelliVisor subject contains metaclasses that contain properties that control the IntelliVisor tool, which provides suggestions during common tasks.

## General

The General metaclass contains properties that determines when the IntelliVisor is enabled in Rational Rhapsody.

### ActivateOnCode

The ActivateOnCode property determines whether the Intellivisor is enabled (Checked) or disabled (Cleared) for code.

*Default = Checked*

### ActivateOnGe

The ActivateOnGe property determines whether the Intellivisor is enabled (Checked) or disabled (Cleared) in the drawing area.

*Default = Checked*

### DependencyRecursionLimit

When you use the autocompletion feature to include model elements in your code, Rhapsody also includes model elements that are available as a result of dependency relationships. The list of elements presented includes those that are the result of recursive dependencies. To prevent situations where such recursivity results in long delays or very long lists of elements, you can use the property DependencyRecursionLimit to limit the number of levels of recursion that are taken into account.

The first level is considered to be the model element itself, so if you want to limit the list to direct dependencies, you would enter 2 as the value of the property.

If you don't want to limit the level of recursion, set the value of the property to 0.

*Default = 0 (default value is 2 when using Rhapsody in Java)*

### ShowPredefineMacros

The ShowPredefineMacros property determines whether the values defined in the IntelliVisor::PredefineMacros properties (and their corresponding tooltips defined in

IntelliVisor::PredefineMacrosTooltip) are included in the lists generated by the IntelliVisor. The default items defined in PredefineMacros and PredefineMacrosTooltip (GEN, IS\_IN, and OPORT) are commonly used values. However, you can expand this list by doing the following:

- Add a property to the predefined macros list. The name of the property is visible in the list control; the value of the property is placed inside the code when the macro is selected.
- Add a tooltip to the tooltip list. This tooltip is visible when the item is selected from the list.

*Default = Checked*

## PredefineMacros

The PredefineMacros metaclass contains properties that specify the syntax of the predefined macros that are included in the lists generated by the IntelliVisor.

### CGEN

The CGEN property specifies the syntax of the macro that generates an event.

*Default = CGEN*

### CIS\_IN

The CIS\_IN property specifies the syntax of the macro that determines whether a statechart is in the specified state.

*Default = CIS\_IN*

### GEN

The GEN property specifies the syntax of the macro that generates an event.

*Default = GEN*

### IS\_IN

The IS\_IN property specifies the syntax of the macro that determines whether a statechart is in the specified state.

*Default = IS\_IN*

### OPORT

The OPORT property is a shortcut for OUT\_PORT. This macro relays messages through the port. For example: OPORT(p)-foo(); // calls foo() by using the port // OPORT(p)-GEN(evt); // sends event evt by using the port

*Default = OPORT*

## PredefineMacroTooltip

The PredefineMacroTooltip metaclass contains properties that specify the tooltips displayed for the predefined macros that are included in the lists generated by the IntelliVisor.

### CGEN

The CGEN property specifies the tooltip displayed by the IntelliVisor for the CGEN macro.

*Default = CGEN(<<instance>>,<<event>>)*

### CIS\_IN

The CIS\_IN property specifies the tooltip displayed by the IntelliVisor for the CIS\_IN macro.

*Default = CIS\_IN(<<me>>,<<state>>)*

### GEN

The GEN property specifies the tooltip displayed by the IntelliVisor for the GEN macro.

*Default = Event generation macro:GEN(<<event>>)*

### IS\_IN

The IS\_IN property specifies the tooltip displayed by the IntelliVisor for the IS\_IN macro.

*Default = Statechart test macro:IS\_IN(<<state>>)*

### OPORT

The OPORT property specifies the tooltip displayed by the IntelliVisor for the OPORT macro.

*Default = Port macro:OPORT(<<p>>)*

# Java(1.1)Containers

---

Rational Rhapsody uses containers to implement to-many relations between objects. These include relationships of one object to many, or many objects to many.

The Java(1.1)Containers subject contain metaclasses that contain properties that control implementing relations.

Each property in this subject includes the default value for each container type and relation type. For easier readability, the placeholder `RelationType` in these values represents all the other relation types that are not explicitly detailed. For example, the relation type `User` might have the default value of an empty string, whereas all the other relation types have the value of an empty `MultiLine`.

## BoundedOrdered

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed sequentially.

### Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter: `$cname-push_back($item)`

For maps (qualified relations), the following command inserts an item into the map based on a key: `$cname-insert(map$keyType,$target*::value_type( $keyName,$item))`

The default is `$cname.addElement($item)`.

### Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector: `new vector$target*`

The default is `$cname = new $CType()`.

### CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it

the name stored in \$cname: vector\$target\* \$cname()

The default is \$Create.

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

The default is new \$CType().

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector\$target\* collection type determines the type of the variable cl on the left side of the assignment:

```
vectorClient*>* cl = new vectorClient*>(2);
```

In this case, the class Client is the \$target of vector operations.

The default is Vector.

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item: \$cname-find(\$item)

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about Composite Types.

The default is \$CType \$cname.

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation by using an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the at() operation for the container to retrieve the item at the indexed position: \$cname-at(\$index)

The default is (\$RelationTargetType)(\$cname.elementAt(\$index)).

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end() This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps: \$cname-operator[](\$keyName)

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

(Default = strong)

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: \$cname()

*Default = new \$CType()*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: pos=0; pos<\$multiplicity; pos++; \$cname[pos]=NULL

The default is \$Create.

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

The default is \$CreateStatic.

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: vector\$target\*::const\_iterator \$iterator; \$iterator=\$cname-begin()

*(Default = \$IterType \$iterator = 0;)*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

The default is \$IterCreate.

## IterGetCurrent



The `IterGetCurrent` property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator` This value is the same as that for `IterTest` only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for `IterGetCurrent` returns a pointer to the current item in the collection.

The default is `($RelationTargetType)($cname.elementAt($iterator))`.

## **IterIncrement**

The `IterIncrement` property specifies the code that increments the iterator. For example, the following command moves the `$iterator` ahead one item:

`$iterator++` (Default)

## **IterIncrementForCleanUp**

The `IterIncrementForCleanUp` property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

The default is `$IterIncrement`.

## **IterIncrementForInit**

The `IterIncrementForInit` property specifies the code that increments the iterator, for certain initialization cases.

The default is `$IterIncrement`.

## **IterInit**

The `IterInit` property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

## **IterReset**

The `IterReset` property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection: `$iterator=$cname-begin()`

The default is `$iterator = 0`.

## **IterReturnType**

The `IterReturnType` property specifies the return type for functions such as getters for the various

containers that Rational Rhapsody uses.

The default is \$IterType.

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname-end() With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

The default is \$iterator < \$cname.size().

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<, as defined in the STL. vector\$target\*::const\_iterator You can change the iterator type to one of your own choice.

The default is int.

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$cname. The default value for all other subjects is \$cname.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

The default is \$(constant)\$target.

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to be removed and then call erase() to remove it. `vector$target*::iterator pos=find($cname-begin(), $cname-end(), $item); $cname-erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair `p`, whose second element, `p.second`, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair$keyType, $target* p; p.second=$item; map$keyType, $target*::iterator pos=find($cname-begin(), $cname-end(), p); $cname-erase(pos)`

The default is `$cname.removeElement($item)`.

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items: `$cname-clear()`

The default is `$cname.removeAllElements()`.

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the `$keyName`, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, `$cname`, is a role name of the relation itself, because there is only one class involved: `$cname = $item`

## BoundedUnordered

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed sequentially.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter: `$cname-push_back($item)`

For maps (qualified relations), the following command inserts an item into the map based on a key: `$cname-insert(map$keyType,$target*::value_type($keyName,$item))`

The default is `$cname.addElement($item)`.

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector: `new vector$target*`

The default is `$cname = new $CType()`.

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`: `vector$target* $cname()`

The default is `$Create`.

## CreateStatic

The CreateStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to `Reference`.

The default is `new $CType()`.

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the `vector$target*` collection type determines the type of the variable `cl` on the left side of the assignment:

```
vectorClient*>* cl = new vectorClient*>(2);
```

In this case, the class `Client` is the `$target` of vector operations.

The default is `Vector`.

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item: \$cname-find(\$item)

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about Composite Types.

The default is \$CType \$cname.

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the at() operation for the container to retrieve the item at the indexed position: \$cname-at(\$index)

The default is (\$RelationTargetType)(\$cname.elementAt(\$index)).

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end() This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name that uses the subscript operator[], which has been overloaded according to the STL definition for maps:  
\$cname-operator[](\$keyName)

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file. (Default)
- weak - The #include directives are added to the source file with forward declarations in the header file.

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: \$cname()

*Default = new \$CType()*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: pos=0; pos<\$multiplicity; pos++; \$cname[pos]=NULL

The default is \$Create.

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

The default is \$CreateStatic.

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: `vector$target*::const_iterator $iterator; $iterator=$sname-begin()`

The default is `$IterType $iterator = 0;`.

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

The default is \$IterCreate.

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

The default is `($RelationTargetType)($sname.elementAt($iterator))`.

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item: `$iterator++` (Default)

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

The default is \$IterIncrement.

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization

cases.

The default is \$IterIncrement.

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname-begin()
```

The default is `$iterator = 0`.

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

The default is `$IterType`.

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname-end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as `IterGetCurrent`).

The default is `$iterator < $cname.size()`.

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<`, as defined in the STL. `vector$target*::const_iterator` You can change the iterator type to one of your own choice.

The default is `int`.

## Loop



The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

The default value for Java is as follows: `for (int pos = 0; pos $multiplicity; pos++)`

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$name. The default value for all other subjects is \$name.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

The default is `$(constant)$target`.

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector$target*::iterator pos=find($cname-begin(), $cname-end(),$item);$cname-erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair$keyType,$target* p; p.second=$item; map$keyType,$target*::iterator pos=find($cname-begin(), $cname-end(),p); $cname-erase(pos)`

The default is `$cname.removeElement($item)`

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items: `$cname-clear()`

The default is `$cname.removeAllElements()`.

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: \$cname-erase(\$keyName)

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved: \$cname = \$item

## Type

The Type property specifies the type of the container as a pointer to the relation.

## Fixed

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed sequentially.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter: \$cname-push\_back(\$item)

For maps (qualified relations), the following command inserts an item into the map based on a key: \$cname-insert(map\$keyType,\$target\*::value\_type( \$keyName,\$item))

The default is \$cname.addElement(\$item).

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector: new vector\$target\*

The default is \$cname = new \$CType().

## CreateByValue

The `CreateByValue` property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`: `vector$target* $cname()`

The default is `$Create`.

## CreateStatic

The `CreateStatic` property defines the initialization of a relation when `CG::Relation::Containment` is set to `Reference`.

The default is `new $CType()`

## CType

The `CType` property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the `vector$target*` collection type determines the type of the variable `cl` on the left side of the assignment:

```
vectorClient*>* cl = new vectorClient*>(2);
```

In this case, the class `Client` is the `$target` of vector operations.

The default is `Vector`.

## Find

The `Find` property specifies the command used to locate an item in a container. For example, the following command calls the `Find()` operation for container `$cname` to locate the `$item`: `$cname-find($item)`

## FullTypeDefinition

The `FullTypeDefinition` property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about Composite Types.

The default is `$CType $cname`.

## Get

The `Get` property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: `$me$cname` The variable `$me` is replaced with the object context variable as specified by the `Me` property. The variable

\$cname is replaced with the name of the container, which is the role name for the relation.

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the at() operation for the container to retrieve the item at the indexed position: \$cname-at(\$index)

The default is (\$RelationTargetType)(\$cname.elementAt(\$index)).

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end() This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps: \$cname-operator[](\$keyName)

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file. (Default)
- weak - The #include directives are added to the source file with forward declarations in the header file.

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: \$cname()

*Default = new \$CType()*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: pos=0; pos<\$multiplicity; pos++; \$cname[pos]=NULL The default is \$Create.

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

The default is \$CreateStatic.

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: vector\$target\*::const\_iterator \$iterator; \$iterator=\$cname-begin()

The default is \$IterType \$iterator = 0;.

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

The default is \$IterCreate.

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For

example, the following command returns a pointer to the current item in the collection: `*$iterator` This value is the same as that for `IterTest` only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for `IterGetCurrent` returns a pointer to the current item in the collection.

The default is `($RelationTargetType)($cname.elementAt($iterator))`.

## **IterIncrement**

The `IterIncrement` property specifies the code that increments the iterator. For example, the following command moves the `$iterator` ahead one item: `$iterator++` (Default)

## **IterIncrementForCleanUp**

The `IterIncrementForCleanUp` property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

The default is `$IterIncrement`.

## **IterIncrementForInit**

The `IterIncrementForInit` property specifies the code that increments the iterator, for certain initialization cases.

The default is `$IterIncrement`.

## **IterInit**

The `IterInit` property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

## **IterReset**

The `IterReset` property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection: `$iterator=$cname-begin()`

The default is `$iterator = 0`.

## **IterReturnType**

The `IterReturnType` property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname-end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

The default is `$iterator < $cname.size()`.

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<`, as defined in the STL. `vector$target*::const_iterator` You can change the iterator type to one of your own choice.

The default is `int`.

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

The default value for Java is as follows: `for (int pos = 0; pos $multiplicity; pos++)`

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is `$me$cname`. The default value for all other subjects is `$cname`.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

The default is `$(constant)$target`.

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the `find()` operation to point the iterator to the item to item to be removed and then call `erase()` to remove it. `vector$target*::iterator pos=find($cname-begin(), $cname-end(),$item);$cname-erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair `p`, whose second element, `p.second`, is the item to be removed. The `find()` operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair$keyType,$target* p; p.second=$item; map$keyType,$target*::iterator pos=find($cname-begin(), $cname-end(),p); $cname-erase(pos)`

The default is `$cname.removeElement($item)`.

## RemoveAll

The `RemoveAll` property specifies the command used to remove all items from the container. For example, the following command calls the `clear()` operation for the container to remove all items: `$cname-clear()`

The default is `$cname.removeAllElements()`.

## RemoveKey

The `RemoveKey` property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the `erase()` operation for the container, passing it the `$keyName`, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

## Set

The `Set` property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, `$cname`, is a role name of the relation itself, because there is only one class involved: `$cname = $item`

# General

The `General` metaclass contains properties that enable you to set the directives and include files for the container.

## ContainerDirectives

The `ContainerDirectives` property specifies the preprocessor directives that are necessary when compiling code that uses a particular container library.

No additional directives are required when you use `OMContainers`.



## ContainerIncludes

The ContainerIncludes property specifies header files that must be included when you use a particular container library.

For example, when you use STLContainers, the following string causes one #include directive per container to be added to generated files (such as #include string): string, algorithm, vector, list, map, iterator Whether the #include directives are added to source or header files depends on the value of the IncludeDirective property.

*(Default = java.util.\*)*

## Qualified

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed sequentially.

### Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter: \$cname-push\_back(\$item)

For maps (qualified relations), the following command inserts an item into the map based on a key: \$cname-insert(map\$keyType,\$target\*::value\_type( \$keyName,\$item))

The default is \$cname.put(\$keyName,\$item).

### Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector: new vector\$target\*

The default is \$cname = new \$CType().

### CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname: vector\$target\* \$cname()

The default is \$Create.

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

The default is new \$CType().

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector\$target\* collection type determines the type of the variable cl on the left side of the assignment:

```
vectorClient*>* cl = new vectorClient*>(2);
```

In this case, the class Client is the \$target of vector operations.

The default is Hashtable.

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item: \$cname-find(\$item)

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about Composite Types.

The default is \$CType \$cname.

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the at() operation for the container to retrieve the item at the indexed position: \$cname-at(\$index)

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end() This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps: \$cname-operator[](\$keyName)

The default is (\$RelationTargetType)(\$cname.get(\$keyName)).

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file. (Default)
- weak - The #include directives are added to the source file with forward declarations in the header file.

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that

describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: \$cname()

*Default = new \$CType()*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: pos=0; pos<\$multiplicity; pos++; \$cname[pos]=NULL The default is \$Create

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

The default is \$CreateStatic.

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: vector\$target\*::const\_iterator \$iterator; \$iterator=\$cname-begin()

The default is \$IterType \$IterReset.

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

The default is \$IterCreate.

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: \*\$iterator This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

The default is (\$RelationTargetType)(\$iterator.nextElement()).

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item: `$iterator++`

The default is `$iterator.nextElement()`.

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

The default is `$IterCreate`.

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

The default is `$IterCreate`.

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection: `$iterator=$cname-begin()`

The default is `$iterator = $cname.elements()`.

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

The default is `$IterType`.

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname-end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as `IterGetCurrent()`).

The default is `$iterator.hasMoreElements()`.

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<, as defined in the STL. vector$target*::const_iterator You can change the iterator type to one of your own choice.`

The default is Enumeration.

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use `$Loop` expand the specified Loop property body.

The default value for Java is as follows: `for (int pos = 0; pos $multiplicity; pos++)`

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for `RiCContainers::EmbeddedScalar` is `$me$cname`. The default value for all other subjects is `$cname`.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

The default is `$(constant)$target`.

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the `find()` operation to point the iterator to the item to item to be removed and then call `erase()` to remove it. `vector$target*::iterator pos=find($cname-begin(), $cname-end(),$item);$cname-erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. pair\$keyType,\$target\* p; p.second=\$item; map\$keyType,\$target\*::iterator pos=find(\$cname-begin(), \$cname-end(),p); \$cname-erase(pos)

The default is as follows:

```
$IterCreate; while (iter.hasMoreElements()) { Object key = iter.nextElement(); if ($cname.get(key).equals($item)) { $cname.remove(key); break; } }
```

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container.

The default is \$cname.clear()

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: \$cname-erase(\$keyName)

The default is \$cname.remove(\$keyName).

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved: \$cname = \$item

## Type

The Type property specifies the type of the container as a pointer to the relation.

## Scalar

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed sequentially.

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector\$target\* collection type determines the type of the variable cl on the left side of the assignment:

```
vectorClient*>* cl = new vectorClient*>(2);
```

In this case, the class Client is the \$target of vector operations.

The default is \$RelationTargetType.

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item: \$cname-find(\$item)

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about Composite Types.

The default is \$CType \$cname

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

The default is \$cname.

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the at() operation for the container to retrieve the item at the indexed position: \$cname-at(\$index)

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example,



the following command calls the `end()` operation for the container to locate the last item in the collection:  
`$cname-end()` This property and `GetEndGenerate` were created to adhere to the standard library convention for “finding” where iteration should end. `GetEnd` is generated where `Get` is generated. The method name is defined by using the `CG::Relation::GetEnd` and `CG::Relation::GetEndGenerate` properties.

## GetKey

The `GetKey` property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator `[]`, which has been overloaded according to the STL definition for maps:  
`$cname-operator[]($keyName)`

## IncludeDirective

The `IncludeDirective` property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- `strong` - The `#include` directives are added to the header file.
- `weak` - The `#include` directives are added to the source file with forward declarations in the header file. (Default)

## IncludeFiles

The `IncludeFiles` property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the `Containment` property is set to `Reference`, a forward declaration of the container is added to the class specification file, and the `#include` is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

## IterReturnType

The `IterReturnType` property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

The default is `$RelationTargetType`.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

The default is `$(constant)$target`.

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, `$cname`, is a role name of the relation itself, because there is only one class involved: `$cname = $item` (Default)

## StaticArray

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed sequentially.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter: `$cname-push_back($item)`

For maps (qualified relations), the following command inserts an item into the map based on a key: `$cname-insert(map$keyType,$target*::value_type( $keyName,$item))`

The default is as follows:

```
$Loop { if($cname[pos] == null) { $cname[pos] = $item; break; } }
```

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector: `new vector$target*`

The default is `$cname = $CreateStatic`.

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname: `vector$target* $cname()`

The default is \$Create.

## CreateStatic

The CreateStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to `Reference`.

The default is `new $target[$multiplicity]`.

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the `vector$target*` collection type determines the type of the variable `cl` on the left side of the assignment:

```
vectorClient*>* cl = new vectorClient*>(2);
```

In this case, the class `Client` is the \$target of vector operations.

The default is `$RelationTargetType[] $cname`.

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the `Find()` operation for container \$cname to locate the \$item: `$cname-find($item)`

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about Composite Types.

The default is \$CType.

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a

particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: `$me$cname` The variable `$me` is replaced with the object context variable as specified by the `Me` property. The variable `$cname` is replaced with the name of the container, which is the role name for the relation.

## GetAt

The `GetAt` property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The `ContainerTypes::RelationType::GetAt` property specifies a template for the body of the operation. For example, the following command generates code that calls the `at()` operation for the container to retrieve the item at the indexed position: `$cname-at($index)`

The default is `$cname[$index]`.

## GetEnd

The `GetEnd` property specifies the command used to retrieve the last item in the container. For example, the following command calls the `end()` operation for the container to locate the last item in the collection: `$cname-end()` This property and `GetEndGenerate` were created to adhere to the standard library convention for “finding” where iteration should end. `GetEnd` is generated where `Get` is generated. The method name is defined by using the `CG::Relation::GetEnd` and `CG::Relation::GetEndGenerate` properties.

## GetKey

The `GetKey` property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator `[]`, which has been overloaded according to the STL definition for maps: `$cname-operator[]($keyName)`

## IncludeDirective

The `IncludeDirective` property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The `#include` directives are added to the header file. (Default)
- weak - The `#include` directives are added to the source file with forward declarations in the header file.

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: \$cname()

*Default = new \$target[\$multiplicity]*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: pos=0; pos<\$multiplicity; pos++; \$cname[pos]=NULL The default is as follows:

```
$Create; $Loop { $cname[pos] = null; }
```

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

The default is \$CreateStatic.

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: vector\$target\*::const\_iterator \$iterator; \$iterator=\$cname-begin()

The default is \$IterType \$iterator = 0;.

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

The default is \$IterCreate.

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: \*\$iterator This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

The default is \$cname[\$iterator].

## IterIncrement

The `IterIncrement` property specifies the code that increments the iterator. For example, the following command moves the `$iterator` ahead one item: `$iterator++` (Default)

## **IterIncrementForCleanUp**

The `IterIncrementForCleanUp` property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

The default is `$IterIncrement`.

## **IterIncrementForInit**

The `IterIncrementForInit` property specifies the code that increments the iterator, for certain initialization cases.

The default is `$IterIncrement`.

## **IterReset**

The `IterReset` property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection: `$iterator=$cname-begin()`

The default is `$iterator = 0`.

## **IterReturnType**

The `IterReturnType` property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

The default is `$IterType`.

## **IterTest**

The `IterTest` property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname-end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, `IterTest` retrieves the current item in the collection (the same as `IterGetCurrent`).

The default is `$iterator < $multiplicity`.

## **IterType**

The `IterType` property specifies the iterator type. For example, the following command denotes that the

iterator in use is the parameterized type `vector<`, as defined in the STL. `vector$target*::const_iterator` You can change the iterator type to one of your own choice.

The default is `int`.

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the `OMContainers.StaticArray` metaclass) can use any other property in the same metaclass. Therefore, other properties that use `$Loop` expand the specified Loop property body.

The default value for Java is as follows: `for (int pos = 0; pos < $multiplicity; pos++)`

## RelationTargetType

The `RelationTargetType` property specifies the return type for relation getters, as part of the relation implementation properties.

The default is `$(constant)$target`.

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the `find()` operation to point the iterator to the item to item to be removed and then call `erase()` to remove it. `vector$target*::iterator pos=find($cname-begin(), $cname-end(),$item);$cname-erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair `p`, whose second element, `p.second`, is the item to be removed. The `find()` operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair$keyType,$target* p; p.second=$item; map$keyType,$target*::iterator pos=find($cname-begin(), $cname-end(),p); $cname-erase(pos)` The default is as follows:

```
$Loop { if($cname[pos] == $item) { $cname[pos] = null; break; } }
```

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the `clear()` operation for the container to remove all items: `$cname-clear()`

## SetAt

The SetAt property specifies how code is generated for the body of the mutator for a scalar container.

The default is `$cname[$index] = $item`.

## UnboundedOrdered

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed sequentially.

### Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter: `$cname-push_back($item)`

For maps (qualified relations), the following command inserts an item into the map based on a key: `$cname-insert(map$keyType,$target*::value_type( $keyName,$item))`

The default is `$cname.addElement($item)`.

### Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector: `new vector$target*`

The default is `$cname = new $CType()`.

### CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`: `vector$target* $cname()`

The default is `$Create`.

### CreateStatic

The CreateStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to `Reference`.

The default is `new $CType()`.



## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector\$target\* collection type determines the type of the variable cl on the left side of the assignment:

```
vectorClient*>* cl = new vectorClient*>(2);
```

In this case, the class Client is the \$target of vector operations.

The default is Vector.

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about Composite Types.

The default is \$CType \$cname.

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the at() operation for the container to retrieve the item at the indexed position: \$cname-at(\$index)

The default is (\$RelationTargetType)(\$cname.elementAt(\$index)).

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file. (Default)
- weak - The #include directives are added to the source file with forward declarations in the header file.

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: \$cname()

*Default = new \$CType()*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: pos=0; pos<\$multiplicity; pos++; \$cname[pos]=NULL The default is \$Create.

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

The default is \$CreateStatic.

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: vector\$target\*::const\_iterator \$iterator; \$iterator=\$cname-begin()

The default is \$IterType \$IterReset;.

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

The default is \$IterCreate.

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: \*\$iterator This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

The default is (\$RelationTargetType)(\$cname.elementAt(\$iterator)).

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item: \$iterator++ (Default)

## IterIncrementForCleanup

The `IterIncrementForCleanUp` property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

The default is `$IterIncrement`.

## **IterIncrementForInit**

The `IterIncrementForInit` property specifies the code that increments the iterator, for certain initialization cases.

The default is `$IterIncrement`.

## **IterReset**

The `IterReset` property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection: `$iterator=$cname-begin()`

The default is `$iterator = 0`.

## **IterReturnType**

The `IterReturnType` property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

The default is `$IterType`.

## **IterTest**

The `IterTest` property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname-end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, `IterTest` retrieves the current item in the collection (the same as `IterGetCurrent`).

The default is `$iterator < $cname.size()`.

## **IterType**

The `IterType` property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<`, as defined in the STL. `vector$target*::const_iterator` You can change the iterator type to one of your own choice.

The default is `int`.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

The default is `$(constant)$target`.

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the `find()` operation to point the iterator to the item to be removed and then call `erase()` to remove it. `vector$target*::iterator pos=find($cname-begin(), $cname-end(),$item);$cname-erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair `p`, whose second element, `p.second`, is the item to be removed. The `find()` operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair$keyType,$target* p; p.second=$item; map$keyType,$target*::iterator pos=find($cname-begin(), $cname-end(),p); $cname-erase(pos)`

The default is `$cname.removeElement($item)`.

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the `clear()` operation for the container to remove all items: `$cname-clear()` The default is `$cname.removeAllElements()`.

# UnboundedUnordered

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed sequentially.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter: `$cname-push_back($item)`

For maps (qualified relations), the following command inserts an item into the map based on a key: `$cname-insert(map$keyType,$target*::value_type( $keyName,$item))`

The default is `$cname.addElement($item)`.

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector: `new vector$target*` The default is `$cname = new $CType()`.

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname: `vector$target* $cname()` The default is `$Create`.

## CreateStatic

The CreateStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to `Reference`.

The default is `new $CType()`.

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the `vector$target*` collection type determines the type of the variable `cl` on the left side of the assignment:

```
vectorClient*>* cl = new vectorClient*>(2);
```

In this case, the class `Client` is the \$target of vector operations.

The default is `Vector`.

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about Composite Types.

The default is `$CType $cname`.

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the

targets of a to-many relation that uses an index.

The `ContainerTypes::RelationType::GetAt` property specifies a template for the body of the operation. For example, the following command generates code that calls the `at()` operation for the container to retrieve the item at the indexed position: `$cname-at($index)` The default is `($RelationTargetType)($cname.elementAt($index))`.

## IncludeDirective

The `IncludeDirective` property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The `#include` directives are added to the header file. (Default)
- weak - The `#include` directives are added to the source file with forward declarations in the header file.

## Init

The `Init` property specifies the command used to initialize the container. For example, the following command calls the constructor for the container `$cname` to initialize it: `$cname()`

*Default = new \$CType()*

## InitInCtorBody

The `InitInCtorBody` property specifies code to initialize containers for arrays inside the class constructor body. For example: `pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL` The default is `$Create`.

## InitStatic

The `InitStatic` property defines the initialization of a relation when `CG::Relation::Containment` is set to `Value`.

The default is `$CreateStatic`.

## IterCreate

The `IterCreate` property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: `vector$target*::const_iterator $iterator; $iterator=$cname-begin()` The default is `$IterType $IterReset;`

## IterCreateByValue

The `IterCreateByValue` property is the same as `IterCreate`, but instantiates the iterator by value.

The default is `$IterCreate`.

## **IterGetCurrent**

The `IterGetCurrent` property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator` This value is the same as that for `IterTest` only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for `IterGetCurrent` returns a pointer to the current item in the collection.

The default is `($RelationTargetType)($cname.elementAt($iterator))`.

## **IterIncrement**

The `IterIncrement` property specifies the code that increments the iterator. For example, the following command moves the `$iterator` ahead one item: `$iterator++` (Default)

## **IterIncrementForCleanUp**

The `IterIncrementForCleanUp` property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

The default is `$IterIncrement`.

## **IterIncrementForInit**

The `IterIncrementForInit` property specifies the code that increments the iterator, for certain initialization cases.

The default is `$IterIncrement`.

## **IterReset**

The `IterReset` property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection: `$iterator=$cname-begin()` The default is `$iterator = 0`.

## **IterReturnType**

The `IterReturnType` property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

The default is `$IterType`.

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname-end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

The default is `$iterator < $cname.size()`.

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<T>`, as defined in the STL. `vector$target*::const_iterator` You can change the iterator type to one of your own choice.

The default is `int`.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

The default is `$(constant)$target`.

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the `find()` operation to point the iterator to the item to be removed and then call `erase()` to remove it. `vector$target*::iterator pos=find($cname-begin(), $cname-end(),$item);$cname-erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair `p`, whose second element, `p.second`, is the item to be removed. The `find()` operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair$keyType,$target* p; p.second=$item; map$keyType,$target*::iterator pos=find($cname-begin(), $cname-end(),p); $cname-erase(pos)` The default is `$cname.removeElement($item)`.

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the `clear()` operation for the container to remove all items: `$cname-clear()` The default is `$cname.removeAllElements()`.



## User

Defines the properties for user-defined implementations of relations.

You can create your own implementations for relations by defining a new set of properties under the User metaclass. Once these are defined, you can give them permanent status by manually saving them in the factory.prp file under any other name, for example MyFaves. To complete their installation, you must add the new name as an enumerated value to the CG::Relation::Implementation property.

For example, you can change the definition of the Implementation property as follows:

```
Subject CG Metaclass Relation Property Implementation Enum "Default,Scalar,Fixed,BoundedOrdered,
BoundedUnordered,UnboundedOrdered, UnboundedUnordered,Qualified,MyFaves, User" "Default" end
end
```

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter: \$cname-push\_back(\$item)

For maps (qualified relations), the following command inserts an item into the map based on a key: \$cname-insert(map\$keyType,\$target\*::value\_type( \$keyName,\$item))

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector: new vector\$target\*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname: vector\$target\* \$cname()

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector\$target\* collection type determines the type of the variable cl on the left side of the assignment:

```
vectorClient*>* cl = new vectorClient*>(2);
```

In this case, the class Client is the \$target of vector operations.

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item: \$cname-find(\$item)

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about Composite Types.

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the at() operation for the container to retrieve the item at the indexed position: \$cname-at(\$index)

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end() This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The

method name is defined by using the `CG::Relation::GetEnd` and `CG::Relation::GetEndGenerate` properties.

## GetKey

The `GetKey` property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator `[]`, which has been overloaded according to the STL definition for maps:  
`$cname-operator[]($keyName)`

## IncludeDirective

The `IncludeDirective` property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The `#include` directives are added to the header file. (Default)
- weak - The `#include` directives are added to the source file with forward declarations in the header file.

## IncludeFiles

The `IncludeFiles` property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the `Containment` property is set to `Reference`, a forward declaration of the container is added to the class specification file, and the `#include` is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

## Init

The `Init` property specifies the command used to initialize the container. For example, the following command calls the constructor for the container `$cname` to initialize it: `$cname()`

## InitInCtorBody

The `InitInCtorBody` property specifies code to initialize containers for arrays inside the class constructor body. For example: `pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL`

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: `vector$target*::const_iterator $iterator; $iterator=$cname-begin()`

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator` This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item: `$iterator++`

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection: \$iterator=\$cname-begin()

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname-end() With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<, as defined in the STL. vector\$target\*::const\_iterator You can change the iterator type to one of your own choice.

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$cname. The default value for all other subjects is \$cname.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation

implementation properties.

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector$target*::iterator pos=find($cname-begin(), $cname-end(),$item);$cname-erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair$keyType,$target* p; p.second=$item; map$keyType,$target*::iterator pos=find($cname-begin(), $cname-end(),p); $cname-erase(pos)`

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items: `$cname-clear()`

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved: `$cname = $item`

# Java(1.2)Containers

---

Rational Rhapsody uses containers to implement to-many relations between objects. These include relationships of one object to many, or many objects to many.

The Java(1.2)Containers subject contains metaclasses that contain properties that control implementing relations.

Each property in this subject includes the default value for each container type and relation type. For easier readability, the placeholder `RelationType` in these values represents all the other relation types that are not explicitly detailed. For example, the relation type `User` might have the default value of an empty string, whereas all the other relation types have the value of an empty `MultiLine`.

## BoundedOrdered

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed sequentially.

### Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = \$cname.add(\$item)*

### Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = \$cname = new \$CType()*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector<$target*> $cname()
```

*Default = \$Create*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default = new \$CType()*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = ArrayList*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default =*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.



*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

`$cname->at($index)`

*Default = (\$RelationTargetType)(\$cname.get(\$index))*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the container end() operation to locate the last item in the collection:  
`$cname-end()`

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:  
`$cname-operator[]($keyName)`

*Default =*

## **IncludeDirective**

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## **IncludeFiles**

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default =*

## **Init**

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

```
$cname()
```

*Default = new \$CType()*

## **InitInCtorBody**

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL
```

*Default = \$Create*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default = \$CreateStatic*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = \$IterType \$iterator = 0*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = (\$RelationTargetType)(\$cname.get(\$iterator))*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default = \$iterator++*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = \$IterIncrement*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterIncrement*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = \$iterator = 0*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end()

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With

OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = \$iterator < \$cname.size()*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<>, as defined in the STL.

```
vector<$target*>::const_iterator
```

You can change the iterator type to one of your own choice.

*Default = int*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$target*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. vector<\$target\*>::iterator pos=find(\$cname>begin(),

`$cname>end(),$item);$cname>erase(pos)` This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair `p`, whose second element, `p.second`, is the item to be removed. The `find()` operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find(
$cname->begin(), $cname->end(),p); $cname->erase(pos)
```

```
Default = $IterType _pos = $cname.indexOf($item); if (_pos != -1) { $cname.remove(_pos); }
```

## RemoveAll

The `RemoveAll` property specifies the command used to remove all items from the container. For example, the following command calls the `clear()` operation for the container to remove all items:

```
$cname->clear()
```

```
Default = $cname.clear()
```

## RemoveKey

The `RemoveKey` property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the `erase()` operation for the container, passing it the `$keyName`, which maps into a dictionary used to locate the item to be removed:

```
$cname-erase($keyName)
```

```
Default =
```

## Set

The `Set` property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, `$cname`, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

```
Default =
```

## Type

The `Type` property specifies the type of the container as a pointer to the relation.

```
Default =
```

# BoundedUnordered

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed randomly.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = \$cname.add(\$item)*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = \$cname = new \$CType()*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`:

```
vector<$target*> $cname()
```

*Default = \$Create*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default = new \$CType()*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = ArrayList*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default =*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*



## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

```
$cname->at($index)
```

```
Default = ($RelationTargetType)($cname.get($index))
```

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the container end() operation to locate the last item in the collection:

```
$cname-end()
```

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

```
Default =
```

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:

```
$cname-operator[]($keyName)
```

```
Default =
```

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

```
Default = strong
```

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default =*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

`$cname()`

*Default = new \$CType()*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

`pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL`

*Default = \$Create*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default = \$CreateStatic*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the

container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = \$IterType \$iterator = 0*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = (\$RelationTargetType)(\$cname.get(\$iterator))*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default = \$iterator++*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

*Default = \$IterReset*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterReset*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = \$iterator = 0*

## IterReturntype

The IterReturntype property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()`

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = \$iterator < \$cname.size()*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<>`, as defined in the STL.

```
vector<$target*>::const_iterator
```

You can change the iterator type to one of your own choice.

*Default = int*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$target*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. vector<\$target\*>::iterator pos=find(\$cname>begin(), \$cname>end(),\$item);\$cname->erase(pos) This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find(
$cname->begin(), $cname->end(),p); $cname->erase(pos)
```

*Default = \$IterType \_pos = \$cname.indexOf(\$item); if (\_pos != -1) { \$cname.remove(\_pos); }*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For

example, the following command calls the `clear()` operation for the container to remove all items:

```
$cname->clear()
```

*Default = \$cname.clear()*

## RemoveKey

The `RemoveKey` property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the `erase()` operation for the container, passing it the `$keyName`, which maps into a dictionary used to locate the item to be removed:

```
$cname-erase($keyName)
```

*Default =*

## Set

The `Set` property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, `$cname`, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

*Default =*

## Type

The `Type` property specifies the type of the container as a pointer to the relation.

*Default =*

## Fixed

Defines properties for implementing embedded fixed relations.

## Add

The `Add` property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

```
Default = $cname.add(0, $item)
```

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector:

```
new vector<$target*>
```

```
Default = $cname = new $CType()
```

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector<$target*> $cname()
```

```
Default = $Create
```

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

```
Default = new $CType()
```

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = ArrayList*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

`$cname->find($item)`

*Default =*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

`$cname->at($index)`

*Default = (\$RelationTargetType)(\$cname.get(\$index))*

## GetEnd



The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the container end() operation to locate the last item in the collection:  
\$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:  
\$cname-operator[](\$keyName)

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default =*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

```
$cname()
```

*Default = new \$CType()*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL
```

*Default = \$Create*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default = \$CreateStatic*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = \$IterType \$iterator = 0*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The `IterGetCurrent` property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for `IterTest` only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for `IterGetCurrent` returns a pointer to the current item in the collection.

*Default = (\$RelationTargetType)(\$cname.get(\$iterator))*

## IterIncrement

The `IterIncrement` property specifies the code that increments the iterator. For example, the following command moves the `$iterator` ahead one item:

`$iterator++`

*Default = \$iterator++*

## IterIncrementForCleanUp

The `IterIncrementForCleanUp` property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

*Default = \$IterReset*

## IterIncrementForInit

The `IterIncrementForInit` property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterReset*

## IterInit

The `IterInit` property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the `relation.getRelation>()` generated operation.

*Default =*

## IterReset

The `IterReset` property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection:

`$iterator=$cname->begin()`

*Default = \$iterator = 0*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()`

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = \$iterator < \$cname.size()*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<>`, as defined in the STL.

```
vector<$target*>::const_iterator
```

You can change the iterator type to one of your own choice.

*Default = int*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: `int pos; for (pos = 0; pos $multiplicity; ++pos)` The default value for Java is as follows: `for (int pos = 0; pos $multiplicity; pos++)`

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$target*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname>erase(pos)` This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find(
$name->begin(), $name->end(),p); $name->erase(pos)
```

*Default = \$IterType \_pos = \$cname.indexOf(\$item); if (\_pos != -1) { \$cname.remove(\_pos); }*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

```
$cname->clear()
```

*Default = \$cname.clear()*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

\$cname = \$item

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## General

Defines properties that enable you to set the directives and include files for the container.

### ContainerDirectives

The ContainerDirectives property specifies the preprocessor directives that are necessary when compiling code that uses a particular container library.

No additional directives are required when you use OMContainers.

*Default = Empty string*

### ContainerIncludes

The ContainerIncludes property specifies header files that must be included when you use a particular container library.

For example, when you use STLContainers, the following string causes one #include directive per container to be added to generated files (such as #include string): string, algorithm, vector, list, map, iterator Whether the #include directives are added to source or header files depends on the value of the IncludeDirective property.

*Default = java.util.\**

## Qualified

Defines properties for implementing qualified relations, which are accessed by way of a key.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = \$cname.put(\$keyName,\$item)*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = \$cname = new \$CType()*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`:

```
vector<$target*> $cname()
```

*Default = \$Create*

## CreateStatic

The CreateStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to `Reference`.

*Default = new \$CType()*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = HashMap*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default =*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.



The `ContainerTypes::RelationType::GetAt` property specifies a template for the body of the operation. For example, the following command generates code that calls the `container at()` operation to retrieve the item at the indexed position:

```
$cname->at($index)
```

```
Default = ($RelationTargetType)($cname.get($index))
```

## GetEnd

The `GetEnd` property specifies the command used to retrieve the last item in the container. For example, the following command calls the `container end()` operation to locate the last item in the collection:

```
$cname-end()
```

This property and `GetEndGenerate` were created to adhere to the standard library convention for "finding" where iteration should end. `GetEnd` is generated where `Get` is generated. The method name is defined by using the `CG::Relation::GetEnd` and `CG::Relation::GetEndGenerate` properties.

```
Default =
```

## GetKey

The `GetKey` property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator `[]`, which has been overloaded according to the STL definition for maps:

```
$cname-operator[]($keyName)
```

```
Default = ($RelationTargetType)($cname.get($keyName))
```

## IncludeDirective

The `IncludeDirective` property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- `strong` - The `#include` directives are added to the header file.
- `weak` - The `#include` directives are added to the source file with forward declarations in the header file.

```
Default = strong
```

## IncludeFiles

The `IncludeFiles` property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default =*

## **Init**

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

`$cname()`

*Default = new \$CType()*

## **InitInCtorBody**

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

`pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL`

*Default = \$Create*

## **InitStatic**

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default = \$CreateStatic*

## **IterCreate**

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

`vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()`

*Default = \$IterType \$IterReset*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: \*\$iterator

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = (\$RelationTargetType)(\$cname.get(\$iterator.next()))*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

`$iterator++`

*Default = \$iterator.next()*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = \$IterCreate*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterCreate*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = \$iterator = \$cname.keySet().iterator()*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end()

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = \$iterator.hasNext()*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<>, as defined in the STL.

```
vector<$target*>::const_iterator
```

You can change the iterator type to one of your own choice.

*Default = Iterator*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$target*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. vector<\$target\*>::iterator pos=find(\$cname>begin(), \$cname>end(), \$item); \$cname->erase(pos) This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find(
$name->begin(), $name->end(),p); $name->erase(pos)
```

*Default = \$IterCreate; while(iter.hasNext()) { Object key = iter.next(); if (\$cname.get(key).equals(\$item))  
{ \$cname.remove(key); break; } }*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

```
$cname->clear()
```

*Default = \$cname.clear()*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: \$cname-erase(\$keyName)

*Default = \$cname.remove(\$keyName)*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

\$cname = \$item

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## Scalar

Defines properties for implementing scalar relations.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter:

\$cname->push\_back(\$item)

For maps (qualified relations), the following command inserts an item into the map based on a key:

\$cname->insert(map<\$keyType,\$target\*>::value\_type( \$keyName,\$item))

*Default =*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector:

```
new vector<$target*>
```

*Default =*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector<$target*> $cname()
```

*Default =*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default =*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = \$RelationTargetType*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following

command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default =*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

```
$cname->at($index)
```

*Default =*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the container end() operation to locate the last item in the collection:  
\$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by



using the `CG::Relation::GetEnd` and `CG::Relation::GetEndGenerate` properties.

*Default =*

## GetKey

The `GetKey` property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator `[]`, which has been overloaded according to the STL definition for maps:  
`$cname-operator[]($keyName)`

*Default =*

## IncludeDirective

The `IncludeDirective` property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- `strong` - The `#include` directives are added to the header file.
- `weak` - The `#include` directives are added to the source file with forward declarations in the header file.

*Default = strong*

## IncludeFiles

The `IncludeFiles` property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the `Containment` property is set to `Reference`, a forward declaration of the container is added to the class specification file, and the `#include` is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default =*

## Init

The `Init` property specifies the command used to initialize the container. For example, the following command calls the constructor for the container `$cname` to initialize it:

\$cname()

*Default =*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL
```

*Default =*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default =*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default =*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default =*

## **IterIncrement**

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default =*

## **IterIncrementForCleanUp**

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default =*

## **IterIncrementForInit**

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default =*

## **IterInit**

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

*Default =*

## **IterReset**

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default =*

## **IterReturnType**

The IterReturnType property specifies the return type for functions such as getters for the various

containers that Rational Rhapsody uses.

*Default = \$RelationTargetType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end()

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default =*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<>, as defined in the STL.

vector<\$target\*>::const\_iterator

You can change the iterator type to one of your own choice.

*Default =*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$target*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname>erase(pos)` This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find(
$name->begin(), $name->end(),p); $name->erase(pos)
```

*Default =*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

```
$cname->clear()
```

*Default =*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

`$cname = $item`

*Default = \$cname = \$item*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## StaticArray

Defines properties for implementing static arrays.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = \$Loop { if(\$cname[pos] == null) { \$cname[pos] = \$item; break; } }*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = \$cname = \$CreateStatic*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector<$target*> $cname()
```

*Default = \$Create*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default = new \$target[\$multiplicity]*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = \$RelationTargetType[] \$cname*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default =*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a

particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$name

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$name is replaced with the name of the container, which is the role name for the relation.

*Default = \$name*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

\$cname->at(\$index)

*Default = \$cname[\$index]*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the container end() operation to locate the last item in the collection:

\$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:

\$cname-operator[](\$keyName)

*Default =*

## IncludeDirective



The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## **IncludeFiles**

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default =*

## **Init**

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

`$cname()`

*Default = new \$target[\$multiplicity]*

## **InitInCtorBody**

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

`pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL`

*Default = \$Create; \$Loop { \$cname[pos] = null; }*

## **InitStatic**

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default = \$CreateStatic*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = \$IterType \$iterator = 0;*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate;*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: \*\$iterator

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = \$cname[\$iterator]*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default = \$iterator++*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = \$IterIncrement*

## **IterIncrementForInit**

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterIncrement*

## **IterInit**

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## **IterReset**

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection:

`$iterator=$cname->begin()`

*Default = \$iterator = 0*

## **IterReturnType**

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## **IterTest**

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()`

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = \$iterator < \$multiplicity*

## **IterType**

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<>, as defined in the STL.

```
vector<$target*>::const_iterator
```

You can change the iterator type to one of your own choice.

*Default = int*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

```
for (int pos = 0; pos < $multiplicity; pos++)
```

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$target*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname>erase(pos)` This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find(
```

```
$cname->begin(), $cname->end(),p); $cname->erase(pos)
```

```
Default = $Loop { if($cname[pos] == $item) { $cname[pos] = null; break; } }
```

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

```
$cname->clear()
```

```
Default = Empty MultiLine
```

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed:

```
$cname-erase($keyName)
```

```
Default =
```

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

```
Default =
```

## SetAt

The SetAt property specifies how code is generated for the body of the mutator for a scalar container.

```
Default = $cname[$index] = $item
```

## Type

The Type property specifies the type of the container as a pointer to the relation.

```
Default =
```

# UnboundedOrdered

Defines properties for implementing relations whose multiplicity is unbounded (\*) and that should be accessed sequentially.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = \$cname.add(\$item)*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = \$cname = new \$CType()*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`:

```
vector<$target*> $cname()
```

*Default = \$Create*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default = new \$CType()*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = LinkedList*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default =*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

```
$cname->at($index)
```

```
Default = ($RelationTargetType)($cname.get($index))
```

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the container end() operation to locate the last item in the collection:

```
$cname-end()
```

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

```
Default = $cname.listIterator($cname.lastIndexOf($cname.getLast()))
```

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:

```
$cname-operator[]($keyName)
```

```
Default =
```

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

```
Default = strong
```



## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default =*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

`$cname()`

*Default = new \$CType()*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

`pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL`

*Default = \$Create*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default = \$CreateStatic*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the

container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = \$IterType \$IterReset*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = (\$RelationTargetType)(\$iterator.next())*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default = Empty string*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = \$IterReset*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterReset*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = \$iterator = \$cname.listIterator(0)*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()`

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = \$iterator.hasNext()*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<>`, as defined in the STL.

```
vector<$target*>::const_iterator
```

You can change the iterator type to one of your own choice.

*Default = ListIterator*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$target*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. vector<\$target\*>::iterator pos=find(\$cname>begin(), \$cname>end(),\$item);\$cname->erase(pos) This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find(
$name->begin(), $name->end(),p); $name->erase(pos)
```

*Default = \$cname.remove(\$item)*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For

example, the following command calls the `clear()` operation for the container to remove all items:

```
$cname->clear()
```

*Default = \$cname.clear()*

## RemoveKey

The `RemoveKey` property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the `erase()` operation for the container, passing it the `$keyName`, which maps into a dictionary used to locate the item to be removed:

```
$cname-erase($keyName)
```

*Default =*

## Set

The `Set` property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, `$cname`, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

*Default =*

## Type

The `Type` property specifies the type of the container as a pointer to the relation.

*Default =*

# UnboundedUnordered

Defines properties for implementing relations whose multiplicity is unbounded (\*) and that should be accessed randomly.

## Add

The `Add` property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

```
Default = $cname.add($item)
```

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector:

```
new vector<$target*>
```

```
Default = new $CType()
```

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector<$target*> $cname()
```

```
Default = $Create
```

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

```
Default = new $CType()
```

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = LinkedList*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

*\$cname->find(\$item)*

*Default = \$cname.indexOf(\$item)*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

*\$cname->at(\$index)*

*Default = (\$RelationTargetType)(\$cname.get(\$index))*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the container end() operation to locate the last item in the collection:  
\$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:  
\$cname-operator[](\$keyName)

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default =*



## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

```
$cname()
```

*Default = new \$CType()*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL
```

*Default = \$cname = new \$CType()*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default = \$CreateStatic*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = \$IterType \$IterReset*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The `IterGetCurrent` property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for `IterTest` only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for `IterGetCurrent` returns a pointer to the current item in the collection.

*Default = (\$RelationTargetType)(\$iterator.next())*

## IterIncrement

The `IterIncrement` property specifies the code that increments the iterator. For example, the following command moves the `$iterator` ahead one item:

`$iterator++`

*Default = Empty string*

## IterIncrementForCleanUp

The `IterIncrementForCleanUp` property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

*Default = \$IterReset*

## IterIncrementForInit

The `IterIncrementForInit` property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterReset*

## IterInit

The `IterInit` property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the `relation.getRelation>()` generated operation.

*Default =*

## IterReset

The `IterReset` property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection:

`$iterator=$cname->begin()`

*Default = \$iterator = \$cname.listIterator(0)*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()`

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = \$iterator.hasNext()*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<>`, as defined in the STL.

```
vector<$target*>::const_iterator
```

You can change the iterator type to one of your own choice.

*Default = ListIterator*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: `int pos; for (pos = 0; pos $multiplicity; ++pos)` The default value for Java is as follows: `for (int pos = 0; pos $multiplicity; pos++)`

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$target*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname>erase(pos)` This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find(
$name->begin(), $name->end(),p); $name->erase(pos)
```

*Default = \$cname.remove(\$item)*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

```
$cname->clear()
```

*Default = \$cname.clear()*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

```
Default =
```

## Type

The Type property specifies the type of the container as a pointer to the relation.

```
Default =
```

## User

Defines properties for user-defined implementations of relations.

You can create your own implementations for relations by defining a new set of properties under the User metaclass. Once these are defined, you can give them permanent status by manually saving them in the factory.prp file under any other name, for example MyFaves. To complete their installation, you must add the new name as an enumerated value to the CG::Relation::Implementation property.

For example, you can change the definition of the Implementation property as follows:

```
Subject CG Metaclass Relation Property Implementation Enum "Default,Scalar,Fixed,BoundedOrdered,  
BoundedUnordered,UnboundedOrdered, UnboundedUnordered,Qualified,MyFaves, User" "Default" end  
end
```

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

```
Default = Empty string
```

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = Empty string*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector<$target*> $cname()
```

*Default = Empty string*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default = Empty string*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = Empty string*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

`$cname->find($item)`

*Default = Empty string*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = Empty string*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: `$me$cname`

The variable `$me` is replaced with the object context variable as specified by the `Me` property. The variable `$cname` is replaced with the name of the container, which is the role name for the relation.

*Default = Empty string*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The `ContainerTypes::RelationType::GetAt` property specifies a template for the body of the operation. For example, the following command generates code that calls the `container at()` operation to retrieve the item at the indexed position:

`$cname->at($index)`

*Default = Empty string*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the `container end()` operation to locate the last item in the collection:

`$cname-end()`

This property and `GetEndGenerate` were created to adhere to the standard library convention for "finding" where iteration should end. `GetEnd` is generated where `Get` is generated. The method name is defined by using the `CG::Relation::GetEnd` and `CG::Relation::GetEndGenerate` properties.

*Default = Empty string*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:  
\$cname-operator[](\$keyName)

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = Empty string*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

\$cname()



*Default = Empty string*

## **InitInCtorBody**

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL
```

*Default = Empty string*

## **InitStatic**

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default = Empty string*

## **IterCreate**

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = Empty string*

## **IterCreateByValue**

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = Empty string*

## **IterGetCurrent**

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = Empty string*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default = Empty string*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = Empty string*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = Empty string*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the iterator begin() operation to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = Empty string*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = Empty string*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end()

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = Empty string*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<>, as defined in the STL.

```
vector<$target*>::const_iterator
```

You can change the iterator type to one of your own choice.

*Default = Empty string*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = Empty string*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname>erase(pos)` This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find(
$cname->begin(), $cname->end(),p); $cname->erase(pos)
```

*Default = Empty string*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

```
$cname->clear()
```

*Default = Empty string*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

*Default* =

## **Type**

The Type property specifies the type of the container as a pointer to the relation.

*Default* =

# Java(1.5)Containers

---

Rational Rhapsody uses containers to implement to-many relations between objects. These include relationships of one object to many, or many objects to many.

The Java(1.5)Containers subject contains metaclasses that contain properties that control implementing relations.

Each property in this subject includes the default value for each container type and relation type. For easier readability, the placeholder `RelationType` in these values represents all the other relation types that are not explicitly detailed. For example, the relation type `User` might have the default value of an empty string, whereas all the other relation types have the value of an empty `MultiLine`.

## BoundedOrdered

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed sequentially.

### Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = \$cname.add(\$item)*

### Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = \$cname = new \$CType()*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector<$target*> $cname()
```

*Default = \$Create*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default = new \$CType()*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = ArrayList<\$RelationTargetType>*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default =*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

`$cname->at($index)`

*Default = \$cname.get(\$index)*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the container end() operation to locate the last item in the collection:  
`$cname-end()`

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:

`$cname-operator[]($keyName)`



*Default =*

## **IncludeDirective**

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## **IncludeFiles**

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default =*

## **Init**

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

`$cname()`

*Default = new \$CType()*

## **InitInCtorBody**

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

`pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL`

*Default = \$Create*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default = \$CreateStatic*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = \$IterType \$iterator = \$cname.listIterator()*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = \$cname.get(\$iterator.nextIndex())*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default = \$iterator.next()*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = \$IterIncrement*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterIncrement*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = \$iterator = \$cname.listIterator(0)*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end()

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With

OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = \$iterator.hasNext()*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<>, as defined in the STL.

```
vector<$target*>::const_iterator
```

You can change the iterator type to one of your own choice.

*Default = ListIterator<\$RelationTargetType>*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$target*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. vector<\$target\*>::iterator pos=find(\$cname>begin(),

`$cname>end(),$item);$cname>erase(pos)` This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair `p`, whose second element, `p.second`, is the item to be removed. The `find()` operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find(
$cname->begin(), $cname->end(),p); $cname->erase(pos)
```

*Default = `$cname.remove($item)`*

## RemoveAll

The `RemoveAll` property specifies the command used to remove all items from the container. For example, the following command calls the `clear()` operation for the container to remove all items:

```
$cname->clear()
```

*Default = `$cname.clear()`*

## RemoveKey

The `RemoveKey` property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the `erase()` operation for the container, passing it the `$keyName`, which maps into a dictionary used to locate the item to be removed:

```
$cname-erase($keyName)
```

*Default =*

## Set

The `Set` property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, `$cname`, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

*Default =*

## Type

The `Type` property specifies the type of the container as a pointer to the relation.

*Default =*

# BoundedUnordered

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed randomly.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = \$cname.add(\$item)*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = \$cname = new \$CType()*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`:

```
vector<$target*> $cname()
```

*Default = \$Create*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default = new \$CType()*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = ArrayList<\$RelationTargetType>*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default =*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

```
$cname->at($index)
```

```
Default = $cname.get($index)
```

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the container end() operation to locate the last item in the collection:

```
$cname-end()
```

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

```
Default =
```

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:

```
$cname-operator[]($keyName)
```

```
Default =
```

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

```
Default = strong
```



## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default =*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

`$cname()`

*Default = new \$CType()*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

`pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL`

*Default = \$Create*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default = \$CreateStatic*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the

container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = \$IterType \$iterator = \$cname.listIterator()*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = \$cname.get(\$iterator.nextIndex())*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default = \$iterator.next()*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = \$IterIncrement*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterIncrement*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = \$iterator = \$cname.listIterator(0)*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()`

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = \$iterator.hasNext()*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<>`, as defined in the STL.

```
vector<$target*>::const_iterator
```

You can change the iterator type to one of your own choice.

*Default = ListIterator<\$RelationTargetType>*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$target*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. vector<\$target\*>::iterator pos=find(\$cname>begin(), \$cname>end(),\$item);\$cname->erase(pos) This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find(
$cname->begin(), $cname->end(),p); $cname->erase(pos)
```

```
$cname.remove($item)
```

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For

example, the following command calls the `clear()` operation for the container to remove all items:

```
$cname->clear()
```

*Default = \$cname.clear()*

## RemoveKey

The `RemoveKey` property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the `erase()` operation for the container, passing it the `$keyName`, which maps into a dictionary used to locate the item to be removed:

```
$cname-erase($keyName)
```

*Default =*

## Set

The `Set` property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, `$cname`, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

*Default =*

## Type

The `Type` property specifies the type of the container as a pointer to the relation.

*Default =*

## Fixed

Defines properties for implementing embedded fixed relations.

## Add

The `Add` property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

```
Default = $cname.add(0, $item)
```

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector:

```
new vector<$target*>
```

```
Default = $cname = new $CType()
```

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector<$target*> $cname()
```

```
Default = $Create
```

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

```
Default = new $CType()
```

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = ArrayList<\$RelationTargetType>*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

*\$cname->find(\$item)*

*Default =*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default =*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

*\$cname->at(\$index)*

*Default = \$cname.get(\$index)*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the container end() operation to locate the last item in the collection:  
\$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:  
\$cname-operator[](\$keyName)

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default =*



## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

```
$cname()
```

*Default = new \$CType()*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL
```

*Default = \$Create*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default = \$CreateStatic*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = \$IterType \$iterator = \$cname.listIterator()*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The `IterGetCurrent` property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for `IterTest` only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for `IterGetCurrent` returns a pointer to the current item in the collection.

*Default = \$cname.get(\$iterator.nextIndex())*

## IterIncrement

The `IterIncrement` property specifies the code that increments the iterator. For example, the following command moves the `$iterator` ahead one item:

`$iterator++`

*Default = \$iterator.next()*

## IterIncrementForCleanUp

The `IterIncrementForCleanUp` property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

*Default = \$IterIncrement*

## IterIncrementForInit

The `IterIncrementForInit` property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterIncrement*

## IterInit

The `IterInit` property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## IterReset

The `IterReset` property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection:

`$iterator=$cname->begin()`

*Default = \$iterator = \$cname.listIterator(0)*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end()

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = \$iterator.hasNext()*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<>, as defined in the STL.

`vector<$target*>::const_iterator`

You can change the iterator type to one of your own choice.

*Default = ListIterator<\$RelationTargetType>*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$target*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname>erase(pos)` This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find(
$name->begin(), $name->end(),p); $name->erase(pos)
```

*Default = \$cname.remove(\$item)*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

```
$cname->clear()
```

*Default = \$cname.clear()*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

\$cname = \$item

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## General

Defines properties that enable you to set the directives and include files for the container.

### ContainerDirectives

The ContainerDirectives property specifies the preprocessor directives that are necessary when compiling code that uses a particular container library.

No additional directives are required when you use OMContainers.

*Default = Empty string*

### ContainerIncludes

The ContainerIncludes property specifies header files that must be included when you use a particular container library.

For example, when you use STLContainers, the following string causes one #include directive per container to be added to generated files (such as #include string): string, algorithm, vector, list, map, iterator Whether the #include directives are added to source or header files depends on the value of the IncludeDirective property.

*Default = java.util.\**

## Qualified

Defines properties for implementing qualified relations, which are accessed by way of a key.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = \$cname.put(\$keyName,\$item)*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = \$cname = new \$CType()*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`:

```
vector<$target*> $cname()
```

*Default = \$Create*

## CreateStatic

The CreateStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to `Reference`.

*Default = new \$CType()*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = HashMap<\$keyType, \$RelationTargetType>*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default =*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The `ContainerTypes::RelationType::GetAt` property specifies a template for the body of the operation. For example, the following command generates code that calls the `container at()` operation to retrieve the item at the indexed position:

```
$cname->at($index)
```

```
Default = ($RelationTargetType)($cname.get($index))
```

## GetEnd

The `GetEnd` property specifies the command used to retrieve the last item in the container. For example, the following command calls the `container end()` operation to locate the last item in the collection:

```
$cname-end()
```

This property and `GetEndGenerate` were created to adhere to the standard library convention for "finding" where iteration should end. `GetEnd` is generated where `Get` is generated. The method name is defined by using the `CG::Relation::GetEnd` and `CG::Relation::GetEndGenerate` properties.

```
Default =
```

## GetKey

The `GetKey` property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator `[]`, which has been overloaded according to the STL definition for maps:

```
$cname-operator[]($keyName)
```

```
Default = ($RelationTargetType)($cname.get($keyName))
```

## IncludeDirective

The `IncludeDirective` property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- `strong` - The `#include` directives are added to the header file.
- `weak` - The `#include` directives are added to the source file with forward declarations in the header file.

```
Default = strong
```

## IncludeFiles

The `IncludeFiles` property enables selective framework includes of templates based on a particular relation implementation.



If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default =*

## **Init**

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

`$cname()`

*Default = new \$CType()*

## **InitInCtorBody**

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

`pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL`

*Default = \$Create*

## **InitStatic**

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default = \$CreateStatic*

## **IterCreate**

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

`vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()`

*Default = \$IterType \$IterReset*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = \$cname.get(\$iterator.next())*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default = Empty string*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = \$IterCreate*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterCreate*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = \$iterator = \$cname.keySet().iterator()*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end()

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = \$iterator.hasNext()*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<>, as defined in the STL.

```
vector<$target*>::const_iterator
```

You can change the iterator type to one of your own choice.

*Default = Iterator*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$target*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. vector<\$target\*>::iterator pos=find(\$cname>begin(), \$cname>end(), \$item); \$cname->erase(pos) This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find(
$name->begin(), $name->end(),p); $name->erase(pos)
```

*Default = \$IterCreate; while(iter.hasNext()) { Object key = iter.next(); if (\$cname.get(key).equals(\$item))  
{ \$cname.remove(key); break; } }*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

```
$cname->clear()
```

*Default = \$cname.clear()*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed:

```
$cname-erase($keyName)
```

*Default = \$cname.remove(\$keyName)*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## Scalar

Defines properties for implementing scalar relations.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default =*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector:

```
new vector<$target*>
```

*Default =*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector<$target*> $cname()
```

*Default =*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default =*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = \$RelationTargetType*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following

command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default =*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

```
$cname->at($index)
```

*Default =*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the container end() operation to locate the last item in the collection:  
\$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by

using the `CG::Relation::GetEnd` and `CG::Relation::GetEndGenerate` properties.

*Default =*

## GetKey

The `GetKey` property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator `[]`, which has been overloaded according to the STL definition for maps:  
`$cname-operator[]($keyName)`

*Default =*

## IncludeDirective

The `IncludeDirective` property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- `strong` - The `#include` directives are added to the header file.
- `weak` - The `#include` directives are added to the source file with forward declarations in the header file.

*Default = strong*

## IncludeFiles

The `IncludeFiles` property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the `Containment` property is set to `Reference`, a forward declaration of the container is added to the class specification file, and the `#include` is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default =*

## Init

The `Init` property specifies the command used to initialize the container. For example, the following command calls the constructor for the container `$cname` to initialize it:



\$cname()

*Default =*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL
```

*Default =*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default =*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default =*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default =*

## **IterIncrement**

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default =*

## **IterIncrementForCleanUp**

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default =*

## **IterIncrementForInit**

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default =*

## **IterInit**

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

*Default =*

## **IterReset**

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default =*

## **IterReturnType**

The IterReturnType property specifies the return type for functions such as getters for the various

containers that Rational Rhapsody uses.

*Default = \$RelationTargetType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end()

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default =*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<>, as defined in the STL.

vector<\$target\*>::const\_iterator

You can change the iterator type to one of your own choice.

*Default =*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$target*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname>erase(pos)` This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find(
$name->begin(), $name->end(),p); $name->erase(pos)
```

*Default =*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

```
$cname->clear()
```

*Default =*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

`$cname = $item`

*Default = \$cname = \$item*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## StaticArray

Defines properties for implementing static arrays.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = \$Loop { if(\$cname[pos] == null) { \$cname[pos] = \$item; break; } }*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = \$cname = \$CreateStatic*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector<$target*> $cname()
```

*Default = \$Create*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default = new \$target[\$multiplicity]*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = \$RelationTargetType[] \$cname*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default =*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a

particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$name

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$name is replaced with the name of the container, which is the role name for the relation.

*Default = \$name*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

\$cname->at(\$index)

*Default = \$cname[\$index]*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the container end() operation to locate the last item in the collection:

\$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:

\$cname-operator[](\$keyName)

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## **IncludeFiles**

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default =*

## **Init**

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

`$cname()`

*Default = new \$target[\$multiplicity]*

## **InitInCtorBody**

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

`pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL`

*Default = \$Create; \$Loop { \$cname[pos] = null; }*

## **InitStatic**



The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default = \$CreateStatic*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = \$IterType \$iterator = 0;*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate;*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = \$cname[\$iterator]*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default = \$iterator++*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = \$IterIncrement*

## **IterIncrementForInit**

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterIncrement*

## **IterInit**

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## **IterReset**

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection:

`$iterator=$cname->begin()`

*Default = \$iterator = 0*

## **IterReturnType**

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## **IterTest**

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()`

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = \$iterator < \$multiplicity*

## **IterType**

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<>, as defined in the STL.

```
vector<$target*>::const_iterator
```

You can change the iterator type to one of your own choice.

*Default = int*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

```
for (int pos = 0; pos < $multiplicity; pos++)
```

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$target*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname>erase(pos)` This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find(
```

```
$cname->begin(), $cname->end(),p); $cname->erase(pos)
```

```
Default = $Loop { if($cname[pos] == $item) { $cname[pos] = null; break; } }
```

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

```
$cname->clear()
```

```
Default = Empty MultiLine
```

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed:

```
$cname-erase($keyName)
```

```
Default =
```

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

```
Default =
```

## SetAt

The SetAt property specifies how code is generated for the body of the mutator for a scalar container.

```
Default = $cname[$index] = $item
```

## Type

The Type property specifies the type of the container as a pointer to the relation.

```
Default =
```

# UnboundedOrdered

Defines properties for implementing relations whose multiplicity is unbounded (\*) and that should be accessed sequentially.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = \$cname.add(\$item)*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = \$cname = new \$CType()*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`:

```
vector<$target*> $cname()
```

*Default = \$Create*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default = new \$CType()*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = LinkedList<\$RelationTargetType>*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default =*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

```
$cname->at($index)
```

```
Default = $cname.get($index)
```

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the container end() operation to locate the last item in the collection:

```
$cname-end()
```

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

```
Default = $cname.listIterator($cname.lastIndexOf($cname.getLast()))
```

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:

```
$cname-operator[]($keyName)
```

```
Default =
```

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

```
Default = strong
```

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default =*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

`$cname()`

*Default = new \$CType()*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

`pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL`

*Default = \$Create*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default = \$CreateStatic*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the



container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = \$IterType \$iterator = \$cname.listIterator()*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = \$cname.get(\$iterator.nextIndex())*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default = \$iterator.next()*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = \$IterIncrement*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterIncrement*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = \$iterator = \$cname.listIterator(0)*

## IterReturntype

The IterReturntype property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()`

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = \$iterator.hasNext()*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<>`, as defined in the STL.

```
vector<$target*>::const_iterator
```

You can change the iterator type to one of your own choice.

*Default = ListIterator<\$RelationTargetType>*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$target*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname->erase(pos)` This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find(
$name->begin(), $name->end(),p); $name->erase(pos)
```

*Default = \$cname.remove(\$item)*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For

example, the following command calls the `clear()` operation for the container to remove all items:

```
$cname->clear()
```

*Default = \$cname.clear()*

## RemoveKey

The `RemoveKey` property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the `erase()` operation for the container, passing it the `$keyName`, which maps into a dictionary used to locate the item to be removed:

```
$cname-erase($keyName)
```

*Default =*

## Set

The `Set` property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, `$cname`, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

*Default =*

## Type

The `Type` property specifies the type of the container as a pointer to the relation.

*Default =*

# UnboundedUnordered

Defines properties for implementing relations whose multiplicity is unbounded (\*) and that should be accessed randomly.

## Add

The `Add` property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

```
Default = $cname.add($item)
```

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector:

```
new vector<$target*>
```

```
Default = new $CType()
```

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector<$target*> $cname()
```

```
Default = $Create
```

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

```
Default = new $CType()
```

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = LinkedList<\$RelationTargetType>*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

*\$cname->find(\$item)*

*Default = \$cname.indexOf(\$item)*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

*\$cname->at(\$index)*

*Default = \$cname.get(\$index)*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the container end() operation to locate the last item in the collection:  
\$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:  
\$cname-operator[](\$keyName)

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default =*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

```
$cname()
```

*Default = new \$CType()*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL
```

*Default = \$cname = new \$CType()*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default = \$CreateStatic*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = \$IterType \$iterator = \$cname.listIterator()*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent



The `IterGetCurrent` property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for `IterTest` only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for `IterGetCurrent` returns a pointer to the current item in the collection.

*Default = \$cname.get(\$iterator.nextIndex())*

## IterIncrement

The `IterIncrement` property specifies the code that increments the iterator. For example, the following command moves the `$iterator` ahead one item:

`$iterator++`

*Default = \$iterator.next()*

## IterIncrementForCleanUp

The `IterIncrementForCleanUp` property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

*Default = \$IterIncrement*

## IterIncrementForInit

The `IterIncrementForInit` property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterIncrement*

## IterInit

The `IterInit` property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## IterReset

The `IterReset` property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection:

`$iterator=$cname->begin()`

*Default = \$iterator = \$cname.listIterator(0)*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()`

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = \$iterator.hasNext()*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<>`, as defined in the STL.

`vector<$target*>::const_iterator`

You can change the iterator type to one of your own choice.

*Default = ListIterator<\$RelationTargetType>*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: `int pos; for (pos = 0; pos $multiplicity; ++pos)` The default value for Java is as follows: `for (int pos = 0; pos $multiplicity; pos++)`

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$target*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname>erase(pos)` This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find(
$name->begin(), $name->end(),p); $name->erase(pos)
```

*Default = \$cname.remove(\$item)*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

```
$cname->clear()
```

*Default = \$cname.clear()*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

```
Default =
```

## Type

The Type property specifies the type of the container as a pointer to the relation.

```
Default =
```

## User

Defines properties for user-defined implementations of relations.

You can create your own implementations for relations by defining a new set of properties under the User metaclass. Once these are defined, you can give them permanent status by manually saving them in the factory.prp file under any other name, for example MyFaves. To complete their installation, you must add the new name as an enumerated value to the CG::Relation::Implementation property.

```
Subject CG Metaclass Relation Property Implementation Enum "Default,Scalar,Fixed,BoundedOrdered,  
BoundedUnordered,UnboundedOrdered, UnboundedUnordered,Qualified,MyFaves, User" "Default" end  
end
```

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

```
Default = Empty string
```

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = Empty string*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector<$target*> $cname()
```

*Default = Empty string*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default = Empty string*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = Empty string*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default = Empty string*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = Empty string*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = Empty string*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

```
$cname->at($index)
```

*Default = Empty string*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the container end() operation to locate the last item in the collection:

```
$cname-end()
```

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default = Empty string*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:  
\$cname-operator[](\$keyName)

*Default =*

## **IncludeDirective**

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## **IncludeFiles**

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = Empty string*

## **Init**

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

\$cname()

*Default = Empty string*

## **InitInCtorBody**

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL
```

*Default = Empty string*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default = Empty string*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = Empty string*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = Empty string*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = Empty string*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following



command moves the \$iterator ahead one item:

```
$iterator++
```

*Default = Empty string*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = Empty string*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = Empty string*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = Empty string*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = Empty string*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end()

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = Empty string*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<>, as defined in the STL.

vector<\$target\*>::const\_iterator

You can change the iterator type to one of your own choice.

*Default = Empty string*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = Empty string*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(), $item); $cname->erase(pos)` This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find(
$cname->begin(), $cname->end(),p); $cname->erase(pos)
```

*Default = Empty string*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

```
$cname->clear()
```

*Default = Empty string*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

# JAVA\_CG

---

The JAVA\_CG subject contains metaclasses that contain properties that specify operating system environments.

## AnimInstrumentation

The AnimInstrumentation metaclass contains a property that controls the headers for Java files.

### Headers

The Headers property is a string that specifies additional `#import` statements that are needed for the instrumented code.

*Default =*

*com.ibm.rational.rhapsody.animation.\*,com.ibm.rational.rhapsody.animcom.\*,com.ibm.rational.rhapsody.animcom.animM*

## Argument

The Argument metaclass contains properties that control how arguments are generated in code.

### ClassWide

*The ClassWide property determines whether a class-wide modifier is generated for the argument. (Default = False)*

### DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element

descriptions.

- `$Name` - the element name
- `$FullName` - the full path of the element (P1::P2::C.a)
- `$Description` - the text provided on the Description tab of the Features window for the element
- `$Requirements` - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the `$Requirements` keyword in the `DescriptionTemplate` property to get such comments generated for specific model elements.
- `$Login` - the login name of the user who generated the file
- `$CodeGeneratedDate` - the date on which the code was generated
- `$CodeGeneratedTime` - the time at which the code was generated
- `$RhapsodyVersion` - the version of Rhapsody that was used to generate the file
- For operations: `$Signature` - the operation signature, `$Arguments` - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: `$Type` - the type of the attribute
- For arguments: `$Type` - the argument type, `$Direction` - the argument direction (In, Out, or InOut)
- For events: `$Arguments` - the text provided on the Description tab of the Features window for the event arguments
- For association ends: `$Target` - the other end of the association
- For requirements: `$ID` - the ID specified for the requirement, `$Specification` - the specification text for the requirement
- The values of tags can be included by using the syntax `$tag_name`. For example, if you use a tag called `author` for your classes, you can use the keyword `$author` in the value of `DescriptionTemplate` in order to have the value of the `author` tag included in the element description.
- The values of properties can be included by using the syntax `$property_name`. For example, you can use the keyword `$Animate` in the value of `DescriptionTemplate` in order to have the value of the `Animate` property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as `$Name`)
- Tag keywords
- Property keywords

If the value of a keyword is a MultiLine value, then each new line (except the first one) starts with the value of the `<lang>_CG::Configuration::DescriptionBeginLine` property, and each line ends with the value of the `<lang>_CG::Configuration::DescriptionEndLine` property.

*Default = @param \$Name[[ \$Description]]*

# Attribute

The Attribute metaclass contains properties that control attributes of code generation, such as whether to generate accessor operations.

## Accessor

The Accessor property is ignored by Rational Rhapsody.

## AccessorGenerate

The AccessorGenerate property specifies whether to generate accessor operations for attributes. The possible values are as follows:

Checked - A get() method is generated for the attribute. (Default)

Cleared - A get() method is not generated for the attribute.

Setting this property to Cleared is one way to optimize your code for size.

## AccessorVisibility

The AccessorVisibility property specifies the access level of the generated accessor for attributes. This property defines the access level of an accessor operation regardless of the visibility of the attribute. The possible values are as follows:

- fromAttribute - Use the access level for the attribute for the accessor.
- public - Set the accessor access level to public.
- private - Set the accessor access level to private.
- default - Set the accessor access level to default.

*Default = fromAttribute*

## AttributeInitializationFile

The AttributeInitializationFile property specifies how static const attributes are initialized. In Rational Rhapsody, you can initialize these attributes in the specification file or directly in the initialization file. This property is analogous to the VariableInitializationFile property for global const variables. The possible values are as follows:

- Default - The attribute is initialized in the specification file if the type declaration begins with const. Otherwise, the variable is initialized in the implementation file.
- Implementation - Initialize constant attributes in the implementation file.

- Specification - Initialize constant attributes in the specification file.

*(Default = Default)*

## ConstantVariableAsDefine

This property is a Boolean value that determines whether the variable, defined as constant in file or package, is generated by using a #define macro. Otherwise, it is generated by using the const qualifier.

*(Default = Cleared)*

## DeclarationPosition

The DeclarationPosition property controls the declaration order of attributes. The possible values are as follows:

- Default - Similar to the AfterClassRecord setting, with the following difference:
- For static attributes defined in a class with the Ada\_CG::Attribute::Visibility property set to Public, these attributes are generated after types whose Ada\_CG::Type::Visibility property is set to Public.
- You should not use this setting for new models. See the Rational Rhapsody Developer for Ada documentation for more information.
- BeforeClassRecord - Generate the attribute immediately before the class record.
- AfterClassRecord - Generate the attribute immediately after the class record.
- StartOfDeclaration - Generate the attribute immediately after the start of the section (private or public part of the specification, or package body).
- EndOfDeclaration - Generate the attribute immediately before the end of the section (private or public part of the specification, or package body).

*(Default = Default)*

## DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- \$Name - the element name



- `$FullName` - the full path of the element (P1::P2::C.a)
- `$Description` - the text provided on the Description tab of the Features window for the element
- `$Requirements` - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the `$Requirements` keyword in the `DescriptionTemplate` property to get such comments generated for specific model elements.
- `$Login` - the login name of the user who generated the file
- `$CodeGeneratedDate` - the date on which the code was generated
- `$CodeGeneratedTime` - the time at which the code was generated
- `$RhapsodyVersion` - the version of Rhapsody that was used to generate the file
- For operations: `$Signature` - the operation signature, `$Arguments` - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: `$Type` - the type of the attribute
- For arguments: `$Type` - the argument type, `$Direction` - the argument direction (In, Out, or InOut)
- For events: `$Arguments` - the text provided on the Description tab of the Features window for the event arguments
- For association ends: `$Target` - the other end of the association
- For requirements: `$ID` - the ID specified for the requirement, `$Specification` - the specification text for the requirement
- The values of tags can be included by using the syntax `$tag_name`. For example, if you use a tag called `author` for your classes, you can use the keyword `$author` in the value of `DescriptionTemplate` in order to have the value of the `author` tag included in the element description.
- The values of properties can be included by using the syntax `$property_name`. For example, you can use the keyword `$Animate` in the value of `DescriptionTemplate` in order to have the value of the `Animate` property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as `$Name`)
- Tag keywords
- Property keywords

If the value of a keyword is a `MultiLine` value, then each new line (except the first one) starts with the value of the `<lang>_CG::Configuration::DescriptionBeginLine` property, and each line ends with the value of the `<lang>_CG::Configuration::DescriptionEndLine` property.

*Default = [[ \* `$Description`]] [[ \* `@see $See`]] [[ \* `@since $Since`]]*

## ImplementationEpilog

The `ImplementationEpilog` property adds any code that you want to be added as verbatim text (to be ignored by Rational Rhapsody) to the end of the definition of a model element. For example, you could wrap a section of code with an `#ifdef-#endif` pair, add compiler-specific keywords, or add a `#pragma`

statement. For example, to specify that an operation is available only when the code is compiled with `_DEBUG`, set the following properties for the operation:

- Set `SpecificationProlog` to `#ifdef _DEBUG cr.`
- Set `SpecificationEpilog` to `#endif.`
- Set `ImplementationProlog` to `#ifdef _DEBUG cr.`
- Set `ImplementationEpilog` to `#endif.`

The following table lists whether leading and trailing linefeeds are generated.

Metaclass	Leading Linefeed	Added?	Class	Yes	Package	No
-----------	------------------	--------	-------	-----	---------	----

*(Default = Empty MultiLine)*

## ImplementationProlog

The `ImplementationProlog` property adds any code that you want to be added as verbatim text (to be ignored by Rational Rhapsody) to the beginning of the definition of a model element. For example, you could wrap a section of code with an `#ifdef-#endif` pair, add compiler-specific keywords, or add a `#pragma` statement. For example, to specify that an operation is available only when the code is compiled with `_DEBUG`, set the following properties for the operation:

- Set `SpecificationProlog` to `#ifdef _DEBUG cr.`
- Set `SpecificationEpilog` to `#endif.`
- Set `ImplementationProlog` to `#ifdef _DEBUG cr.`
- Set `ImplementationEpilog` to `#endif.`

The following table lists whether leading and trailing linefeeds are generated.

*Default = Empty MultiLine*

## InitializationStyle

The `InitializationStyle` property specifies the initialization style used for attributes. When you specify an initial value for an attribute, Rational Rhapsody initializes the attribute based on the value of this property.

In Rational Rhapsody Developer for Java, the possible values are as follows:

- `InClass` - Initialize the attribute in the class declaration. (Default)
- `InConstructor` - Initialize the attribute in each of the class constructors.

## Inline

The `Inline` property specifies how inline operations are generated. Which operations are affected by the `Inline` property depends on the metaclass:

- `Attribute` - Applies only to operations that handle attributes (such as accessors and mutators)

- Operation - Applies to all operations
- Relation - Applies only to operations that handle relations

*Inlining in Rational Rhapsody Developer for Java Because inlining has no meaning in Java, the Inline property is set to none. (Default = none)*

## **IsAliased**

*The IsAliased property is a Boolean value that specifies whether attributes are aliased. (Default = False)*

## **IsMutable**

*The IsMutable property is a Boolean value that allows you to specify that an attribute is a mutable attribute. (Default = False)*

## **IsTransient**

The IsTransient property allows you to specify that an attribute should be declared as transient in order to prevent it from being serialized.

*Default = Cleared*

## **IsVolatile**

The IsVolatile property allows you to specify that an attribute should be declared as volatile.

*Default = Cleared*

## **JavaAnnotation**

The JavaAnnotation property is used by the Rational Rhapsody code generator to insert Java annotations into generated code.

This property is used primarily for regenerating code that was reverse engineered. When you reverse engineer code that contains Java annotations, the value of the `JAVA_ReverseEngineering::ImplementationTrait::ImportJavaAnnotation` property determines how Rational Rhapsody handles the annotation code. If the value of this property is set to Verbatim, then Rational Rhapsody does not import annotations as model elements. Rather, the annotation code is stored as the value of the JavaAnnotation property. When code is later regenerated, it includes the code that was stored in this property.

*Default = Blank*

## **Kind**

The Kind property specifies the kind of operation that should be generated for an element. The kind of operations that can be generated is language-dependent (for example, virtual and abstract exist only in C++ and Java). In Java, Kind can be defined only for attributes and operations, but not for relations. This property affects class operations, in addition to accessors and mutators for relations and attributes. The possible values are as follows:

- common - Class operations and accessor/mutator are non-virtual.
- virtual - Class operations and accessor/mutator are virtual.
- abstract - Class operations and accessor/mutator are pure virtual.

*Default = common*

## MarkPrologEpilogInAnnotations

The MarkPrologEpilogInAnnotations property specifies whether to generate ignore annotations for the Specification, Implementation Prolog, and Epilog properties so they are ignored during roundtrip. When you insert code element declarations (variables, types, functions, and so on) in the Specification, Implementation Prolog, and Epilog properties, after a full roundtrip those elements are added to the model and are duplicated on the next code generation. When you use the MarkPrologEpilogInAnnotations property, you can have Rational Rhapsody automatically ignore the information specified in the Specification, Implementation Prolog, and Epilog properties instead of adding the ignore annotations manually. The possible values for the MarkPrologEpilogInAnnotations property are as follows:

- None - Rational Rhapsody does not generate any annotations. Any models created before Version 4.1 automatically have this property setting.
- Ignore - Rational Rhapsody generates the `/// ignore annotation before the code specified in the Specification, Implementation Prolog, and Epilog properties, and generates the /// annotation after the code specified in those properties.`
- Auto - If the code in the Specification, Implementation Prolog, and Epilog properties is one line (it does not contain any newline characters (`\n`)), no annotations are generated (the same behavior has the None setting). If there is more than one line, Rational Rhapsody generates the `/// ignore annotation before the code specified in the Specification, Implementation Prolog, and Epilog properties, and generates the /// annotation after the code specified in those properties (the same behavior as the Ignore setting).`

*During roundtrip, any ignore annotations in the comments of the element are not included in its description. Because the Specification, Implementation Prolog, and Epilog properties are generated between the annotation and its declaration for the element, you cannot rename those elements on roundtrip. If you change the name of an element, it is removed from the model and added with the new name. Some model information (for example, property settings) might be lost. (Default = Auto)*

## Mutator

The Mutator property is ignored by Rational Rhapsody.

## MutatorGenerate

The MutatorGenerate property specifies whether to generate mutators for attributes. The possible values are as follows:

- Smart - Mutators are not generated for attributes that have the Constant modifier.
- Always - Mutators are generated, regardless of the modifier.
- Never - Mutators are not generated.

*Default = Cleared*

## MutatorVisibility

The MutatorVisibility property specifies the access level of the generated mutator for attributes. This property defines the access level of a mutator operation regardless of the visibility of the attribute. The possible values are as follows:

- fromAttribute - Use the attribute access level for the mutator.
- public - Set the mutator access level to public.
- private - Set the mutator access level to private.
- protected - Set the mutator access level to protected.
- default - Set the mutator access level to default.

*Default = fromAttribute*

## RefactorRenameRegularExpression

When you use the Refactor:Rename feature, Rhapsody searches the user-defined code in the model for all occurrences of the name of the specified element, including places where the element name appears as part of the name of elements whose name is derived from the element that is being renamed.

To locate these elements, Rhapsody uses a set of regular expressions.

The regular expressions used for this search are defined by the property RefactorRenameRegularExpression.

Different values are used for this property, depending on what type of element you are renaming. For example, you will see that this property has a different value under Class than it does under Attribute. For some types of elements, Rhapsody uses a more generic expression, taken from the value of the property under ModelElement.

In general, the value of this property is a regular expression that consists of a number of "Or" components. For the renaming to work properly, each such component must contain three groupings, with the middle grouping using the variable \$keyword, which represents the name of the element that is being renamed. In some cases, the expression takes this information from the value of another property rather than using the \$keyword variable.

```
Default = ([^A-Za-z0-9_]/^)( $keyword)([^A-Za-z0-9_]/$) /( ^$<CG::Attribute::Mutator>[( $])
/([ ^A-Za-z0-9]$<CG::Attribute::Mutator>[( $]) /( ^$<CG::Attribute::Accessor>[( $])
/([ ^A-Za-z0-9]$<CG::Attribute::Accessor>[( $])
```

## ReferenceImplementationPattern

*The ReferenceImplementationPattern property specifies how the Reference option for attribute/typedefs (composite types) is mapped to code. See the Rational Rhapsody help for information about composite types. (Default = "\*\*")*

## Renames

The Renames property enables one element to rename another element of the same type. You can also rename an element by using a renames dependency. In the case of a conflict, the dependency has precedence. Note the following:

- For attributes, this property works only for static attributes in a class or for attributes in a package.
- For operations, this property contains the name of the operation this operation in renaming. The signatures of the two operations must match.

*(Default = empty string)*

## SpecificationEpilog

The SpecificationEpilog property allows you to add code to the end of the declaration of a model element.

*Default = Blank*

## SpecificationProlog

The SpecificationProlog property allows you to add code to the beginning of the declaration of a model element.

For example, you could add the @Deprecated annotation for an element by entering @Deprecated and a new line as the value of this property.

*Default = Blank*

## VariableInitializationFile

The VariableInitializationFile property specifies how global constant variables are initialized. You can initialize these variables in the specification file. You can use these variables as compile-time constants that can be used to define array sizes, for example. Rational Rhapsody automatically identifies constant variables with const. By modifying this property, you can choose the initialization file directly. The possible values are as follows:

- Default - The variable is initialized in the specification file if the type declaration begins with const. Otherwise, the variable is initialized in the implementation file.
- Implementation - Initialize global constant variables in the implementation file.
- Specification - Initialize global constant variables in the specification file.

*(Default = Default)*

## Visibility

The Visibility property specifies the visibility of that kind of model element. Code generation maps the visibility specified for an element to the same visibility in the generated language. The Visibility setting has the following applicability:

- Classes - Applies only to nested classes, which are defined inside other classes.
- Types - Applies only to types that are defined inside classes. It does not apply to global types, which are defined in packages.

The following table lists the visibility for the JAVA\_CG subject.

- Protected - The model element is protected.
- Private - The element is private.

*Default = fromAttribute*

## Class

The Class metaclass contains properties that affect the generated classes.

### AccessTypeName

*The AccessTypeName property specifies the name of the access type generated for the class record.  
(Default = empty string)*

### ActiveMessageQueueSize

The ActiveMessageQueueSize property specifies the size of the message queue allocated for active classes. The possible values are as follows:

- A string - Specifies the message queue size for an active class.
- An empty string (blank) - The value is set in an operating system-specific manner.

*Default = Empty string*

### ActiveStackSize

The ActiveStackSize property specifies the size of the stack allocated for active objects. The possible values are as follows:

- Any integer - Specifies that a stack of that size is allocated for active objects.
- An empty string (blank) - If not specified, the stack size is set in an operating system-specific manner, based on the value of the ActiveStackSize property for the framework.

*Default = Empty string*

## **ActiveThreadName**

The ActiveThreadName property specifies the name of the active thread. This facilitates debugging in complex environments in which many threads are constantly being created and deleted on-the-fly. This property is effective only in the pSoSystem (both PPC and X86) and VxWorks environments. In pSoSystem, the thread name is truncated to three characters. The animation thread name is not taken from the active thread name. The possible values are as follows:

- A string - Names the active thread.
- An empty string (blank) - The value is set in an operating system-specific manner, based on the value of the ActiveThreadName property for the framework.

*Default = Empty string*

## **ActiveThreadPriority**

The ActiveThreadPriority property specifies the priority of active class threads. The possible values are as follows:

- A string - Specifies thread priority of an active class.
- An empty string (blank) - The value is set in an operating system-specific manner.

*Default = Empty string*

## **AdditionalBaseClasses**

The AdditionalBaseClasses property adds inheritance from external classes to the model.

*Default = Empty string*

Java Specifics In Rational Rhapsody Developer for Java, an inheritance relation is assumed to mean implementing an interface rather than extending it. For example, if you set the AdditionalBaseClasses property to javax.swing.Jtree, the resulting code would be: public class MyClass implements javax.swing.Jtree In other words, MyClass is treated like an interface. In order to extend rather than implement the base class, you must add the string “extends” to the property. For example, if you set AdditionalBaseClasses to extends javax.swing.Jtree, the resulting code would be: public class MyClass extends javax.swing.JTree A third option would be to enter something like: extends javax.swing.Jtree implements Runnable In this case, the resulting code would be: public class MyClass extends javax.swing.Jtree implements Runnable

## **AdditionalNumberOfInstances**

The AdditionalNumberOfInstances property is a string that specifies the size of the local heap allocated for events when the current pool is full. Triggered operations use the event properties. This property provides support for static architectures found in hard real-time and safety-critical systems without memory management capabilities during run time. All events are dynamically allocated during



initialization. Once allocated, the event queue for the thread remains static in size. The possible values are as follows:

- An empty string (blank) - No additional memory is to be allocated when the initial memory pool is exhausted.
- n (a positive integer) - Specifies the size of the array allocated for additional instances.

*Default = Empty string*

## Animate

The Animate property specifies whether animation code is generated for an element. You can specify your own animation function by using the `CG::Attribute::AnimSerializeOperation` property. The semantics of the Animate property is always in favor of the owner settings:

- If a package Animate property is set to False, all the classes owned by the package are not animated, regardless of the class Animate settings.
- If a class Animate property is set to False, all the elements in the class (attributes, operations, relations, and so on) are not animated.
- If an operation Animate property is set to False, all the arguments are not animated.
- If the AnimateArguments property is set to False, all the arguments are not animated, regardless of the specific argument Animate property settings.

*(Default = Checked)*

## BaseNumberOfInstances

The BaseNumberOfInstances property is a string that specifies the size of the local heap memory pool allocated for either:

- Instances of the class (`CPP_CG::Class`)
- Instances of the event (`CPP_CG::Event`)
- This property provides support for static architectures found in hard real-time and safety-critical systems without memory management capabilities during run time. All instances of events are dynamically allocated during initialization. Once allocated, the event queue for the thread remains static in size.

Triggered operations use the properties defined for events. When the memory pool is exhausted, an additional amount, specified by the AdditionalNumberOfInstances property, is allocated. Memory pools for classes can be used only with the Flat statechart implementation scheme. The possible values are as follows:

- An empty string (blank) - Memory is always dynamically allocated.
- n (positive integer) - An array is allocated in this size for instances.

The related properties are as follows:

- AdditionalNumberOfInstances - Specifies the number of instances to allocate if the pool runs out.
- ProtectStaticMemoryPool - Specifies whether the pool should be protected (to support a multithreaded environment)

- EmptyMemoryPoolCallback - Specifies a user callback function to be called when the pool is empty. This property should be used instead of the AdditionalNumberOfInstance property for error handling.
- EmptyMemoryPoolMessage - When set to true, this property causes a message to be displayed if the pool runs out of memory in instrumented mode.

*Default = Empty string*

## ComplexityForInlining

The ComplexityForInlining property specifies the upper bound for the number of lines in user code that are allowed to be inlined. User code is the action part of transitions in statecharts. For example, when you use the value 3, all transitions with actions consisting of three lines or fewer of code are automatically inlined in the calling function. Inlining is replacing a function call in the generated code with the actual code statements that make up the body of the function. This optimizes the code execution at the expense of an increase in code size. For example, increasing the number of lines that can be inlined from 3 to 5 has shortened the code execution time in some cases up to 10%. This property applies only to the Flat implementation scheme for statecharts.

*Default = 0*

## DeclarationModifier

The DeclarationModifier property adds a string to the class or event declaration. The string is displayed between the class keyword and the class name in the generated code. For example, for a class A, the DeclarationModifier would appear as follows: `class DeclarationModifier> A {...}`; This property adds a modifier to the class declaration. For example, if you have a class myExportableClass that is exported from a DLL by using the MYDLL\_API macro, you can set the DeclarationModifier property to "MYDLL\_API." The generated code would then be as follows: `class MYDLL_API myExportableClass {...}`; This property supports two keywords: \$component and \$class.

*Default = Empty string*

## DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- \$Name - the element name

- `$FullName` - the full path of the element (P1::P2::C.a)
- `$Description` - the text provided on the Description tab of the Features window for the element
- `$Requirements` - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the `$Requirements` keyword in the `DescriptionTemplate` property to get such comments generated for specific model elements.
- `$Login` - the login name of the user who generated the file
- `$CodeGeneratedDate` - the date on which the code was generated
- `$CodeGeneratedTime` - the time at which the code was generated
- `$RhapsodyVersion` - the version of Rhapsody that was used to generate the file
- For operations: `$Signature` - the operation signature, `$Arguments` - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: `$Type` - the type of the attribute
- For arguments: `$Type` - the argument type, `$Direction` - the argument direction (In, Out, or InOut)
- For events: `$Arguments` - the text provided on the Description tab of the Features window for the event arguments
- For association ends: `$Target` - the other end of the association
- For requirements: `$ID` - the ID specified for the requirement, `$Specification` - the specification text for the requirement
- The values of tags can be included by using the syntax `$tag_name`. For example, if you use a tag called `author` for your classes, you can use the keyword `$author` in the value of `DescriptionTemplate` in order to have the value of the `author` tag included in the element description.
- The values of properties can be included by using the syntax `$property_name`. For example, you can use the keyword `$Animate` in the value of `DescriptionTemplate` in order to have the value of the `Animate` property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as `$Name`)
- Tag keywords
- Property keywords

If the value of a keyword is a `MultiLine` value, then each new line (except the first one) starts with the value of the `<lang>_CG::Configuration::DescriptionBeginLine` property, and each line ends with the value of the `<lang>_CG::Configuration::DescriptionEndLine` property.

*Default = [[ \* `$Description`]] [[ \* `@author $Author`]] [[ \* `@version $Version`]] [[ \* `@see $See`]] [[ \* `@since $Since`]]*

## Destructor

The `Destructor` property controls the generation of virtual destructors in C++. The property exists for C for historical reasons, with a single value of `auto`, but it has no effect on the generated C code. The possible

values are as follows:

- `auto` - A virtual destructor is generated for an object only if it has at least one virtual function.
- `virtual` - A virtual destructor is generated in all cases.
- `abstract` - A virtual destructor is generated as a pure virtual function.
- `common` - A nonvirtual destructor is generated.

(*Default = auto*)

## Embeddable

The `Embeddable` property is a Boolean property that specifies whether a class can be allocated by value (nested) inside another class or package. For example, if the `Embeddable` property is `True`, 20 instances of a class `A` can be allocated inside another class by using the following syntax: `A itsA[20]`; The possible values are as follows:

- `True` - The object can be allocated by value inside a composite object or package. The object declaration and definition are generated in the specification file of the composite.
- `False` - The object cannot be embedded inside another object. The object declaration and definition are generated in the implementation file of the composite.

The `Embeddable` property is used with the `EmbeddedScalar` and `EmbeddedFixed` properties to determine how to generate code for an embedded object. The `Embeddable` property must be set to `True` for either of those properties to take effect. It is also closely related to the `ImplementWithStaticArray` property, which also needs to be set in order to support by-value allocation. To generate C-like code in C++, set the `Embeddable` property to `True`. Relations can be generated by value only under the following circumstances:

- The `Embeddable` property of the nested class is set to `True`.
- The multiplicity of the relation is well-defined (not “\*”).
- The `ImplementWithStaticArray` property of the component relation is set to `FixedAndBounded`.

When the `Embeddable` property is `False`:

- The attributes of the object are encapsulated. Clients of the object are forced to use it only by way of its operations, because there is no direct access to its attributes.
- Dynamic allocation must be used. The compiler does not know how to statically allocate an object when its declaration is not visible.
- The nested object cannot be reactive. This is because of the reactive macros. There is a complex workaround for this issue.

(*Default = Checked*)

## EnableDynamicAllocation

The `EnableDynamicAllocation` property specifies whether to use dynamic memory allocation for objects. The possible values are as follows:

- `True` - Dynamic allocation of events is enabled. `Create()` and `Destroy()` operations are generated for the object or object type.

- False - Events are dynamically allocated during initialization, but not during run time. Create() and Destroy() operations are not generated for the object. This setting is recommended for static architectures that do not use dynamic memory management during run time.

*If you are managing your own memory pools, set this property to False and call CPPReactive\_gen() directly. The following example shows how to call RiCReactive\_gen() directly to send a static event to a reactive object A, when you use a member function of A genStaticEv2A(): void A\_genStaticEv2A(struct A\_t\* const me) { { /\*#[ operation genStaticEv2A() \*/ static struct ev \_ev; ev\_Init(\_ev); RiCEvent\_setDeleteAfterConsume(((RiCEvent\*)\_ev), RiCFALSE); (void) RiCReactive\_gen(me-ric\_reactive, ((RiCEvent\*)\_ev), RiCFALSE); /\*#]\*/ } } Alternatively, you can use internal memory pools by setting the BaseNumberOfInstances property, which results in the use of framework memory pools. If you use the framework memory pools, do not disable the Create() and Destroy() methods because these methods are used to manage the memory pool. When you disable the generation of the Create() and Destroy() methods, you can still inject events in animation by supplying an alternate function to get an event instance. To do this, set the AnimInstanceCreate property. (Default = Checked)*

## EnableUseFromCPP

The EnableUseFromCPP property specifies whether to wrap C operations with an appropriate extern C { } wrapper to prevent problems when code is compiled with a C++ compiler. Wrapping C code with extern C includes C code in a C++ application. Note that the structure definition for the object is not wrapped - only the functions are. For example, if the EnableUseFromCPP is set to True for an object, the following wrapper code is generated for its operations:

```
#ifdef __cplusplus extern "C" { #endif /* __cplusplus */ /* Operations */ #ifdef __cplusplus } #endif /* __cplusplus */
```

*(Default = False)*

## Final

*The Final property, when set to False, specifies that the generated record for the class is a tagged record. This property applies to Ada95. (Default = False)*

## GenerateAccessType

The GenerateAccessType property determines which access types are generated for the class. The possible values are as follows:

- None - Access types are not generated.
- Standard - An access type is generated.
- General - General access types are generated.

*(Default = General)*

## GenerateDestructor

The `GenerateDestructor` property specifies whether to generate a destructor for a class.

*Default = Cleared*

## GenerateRecordType

*The `GenerateRecordType` property determines whether the class record is generated. (Default = Checked)*

## HasUnknownDiscriminant

*The `HasUnknownDiscriminant` property determines whether an unknown discriminant `>` is generated for this class. (Default = False)*

## ImplIncludes

*The `ImplIncludes` property specifies the names (including full paths) of header files to be included at the top of implementation files generated for classes, objects or object types, or packages. Separate multiple file names by using commas, without spaces. (Default = empty string)>*

## ImplementationEpilog

The `ImplementationEpilog` property adds any code that you want to be added as verbatim text (to be ignored by Rational Rhapsody) to the end of the definition of a model element. For example, you could wrap a section of code with an `#ifdef-#endif` pair, add compiler-specific keywords, or add a `#pragma` statement. For example, to specify that an operation is available only when the code is compiled with `_DEBUG`, set the following properties for the operation:

- Set `SpecificationProlog` to `#ifdef _DEBUG cr.`
- Set `SpecificationEpilog` to `#endif.`
- Set `ImplementationProlog` to `#ifdef _DEBUG cr.`
- Set `ImplementationEpilog` to `#endif.`

The following table lists whether leading and trailing linefeeds are generated.

Metaclass	Leading	Linefeed	Added?	Class	Yes	Package	No
-----------	---------	----------	--------	-------	-----	---------	----

*(Default = Empty MultiLine)*

## ImplementationPragmas

*The `ImplementationPragmas` property specifies the user-defined pragmas to generate in the body. (Default = Empty MultiLine)*

## ImplementationPragmasInContextClause

*The ImplementationPragmasInContextClause property specifies the user-defined pragmas to generate in the context clause of the body. (Default = Empty MultiLine)*

## ImplementationProlog

The ImplementationProlog property adds any code that you want to be added as verbatim text (to be ignored by Rational Rhapsody) to the beginning of the definition of a model element. For example, you could wrap a section of code with an #ifdef-#endif pair, add compiler-specific keywords, or add a #pragma statement. For example, to specify that an operation is available only when the code is compiled with \_DEBUG, set the following properties for the operation:

- Set SpecificationProlog to #ifdef \_DEBUG cr.
- Set SpecificationEpilog to #endif.
- Set ImplementationProlog to #ifdef \_DEBUG cr.
- Set ImplementationEpilog to #endif.

The following table lists whether leading and trailing linefeeds are generated.

*(Default = Empty MultiLine)*

## In

The In property specifies how code is generated when the type is used with an argument that has the modifier "In". When a class is used with the "In" modifier, the default is "final \$type" in J.

## InitCleanUpRelations

*The InitCleanUpRelations property specifies whether to generate initRelations() and cleanUpRelations() operations for sets of related global instances. This property applies only to composites and global relations. (Default = Checked)*

## InitializationCode

The InitializationCode property adds the specified initialization code in the body of the class. A non-abstract class can have initialization code that is executed during elaboration of the associated package. (Empty MultiLine)

## InOut

The InOut property specifies how code is generated when the type is used with an argument that has the modifier "InOut". When a class is used with the "InOut" modifier, the default is "\$type" in J.

## InstanceDeclaration

The InstanceDeclaration property specifies how instances are declared in code. The default value for C is

as follows: `struct $name$suffix`

In the generated code, the variable `$name` is replaced with the object (or object type) name. The variable `$suffix` is replaced with the type suffix “`_t`,” if the object is of implicit type. The default value for C++ is as follows: `$name$suffix`

## IsCompletedOperation

The `IsCompletedOperation` specifies whether `state_IS_COMPLETED` operations are generated as functions or macros (by using `#define`). The possible values are as follows:

- Plain - `state_IS_COMPLETED` operations are generated as functions (pre-V4.2 behavior). This is the default value.
- Inline - `state_IS_COMPLETED` operations are generated by using `#define` macros, if the body contains only a return statement.

*(Default = Plain)*

## IsInOperation

The `IsInOperation` specifies how `state_IN` methods are generated.

## IsLimited

*The `IsLimited` property determines whether the class or record type is generated as limited. (Default = False)*

## IsNested

*The `IsNested` property specifies whether to generate the class or package as nested. (Default = False)*

## IsPrivate

*The `IsPrivate` property specifies whether to generate the class or package as private. (Default = False)*

## IsReactiveInterface

The `IsReactiveInterface` property modifies the way reactive classes are generated. It has the following effects:

- Virtual inheritance from `OMReactive`
- Prevents instrumentation
- Prevents the thread argument and the initialization code (setting the active context) in the class constructor



- Creates a pure-virtual destructor (by default)

This property affects only classes that declare themselves as interfaces by having a stereotype with a name that contains the word “interface” (case-insensitive). In previous versions of Rational Rhapsody, a class could inherit from a single reactive class only, regardless of whether it was an interface or implementation class. Beginning with Version 4.0.1 MR2, a class can inherit (implement) several reactive interfaces. In Rational Rhapsody Developer for C++, you must explicitly designate reactive interfaces because the code generator applies special translation rules involving multiple inheritance from the Rational Rhapsody framework. You can designate a reactive interface in two ways:

- Set the `CPP_CG::Class::IsReactiveInterface` property to true.
- Use the predefined stereotype `Reactive_interface`. This stereotype uses stereotype-based code generation in order to automatically apply the correct property value.

Alternatively, you can define another stereotype (such as `PortSpec`) that sets `IsReactiveInterface` to true and use that stereotype. A class is considered reactive if it meets all the following conditions:

- The `CPP_CG::Framework::ReactiveBase` property is not empty.
- The `CPP_CG::Framework::ReactiveBaseUsage` property is set to true.
- One or more of the following conditions are true:
  - The class has a statechart or activity diagram.
  - The class is a composite class.
  - The class has event receptions or triggered operations.

*(Default = Checked)*

Rational Rhapsody Developer for Java Note the following:

- A class is considered a reactive instance when it has an interface (for example, the `Interface` stereotype is applied) and it has event receptions or triggered operations.
- A reactive interface is implemented as an interface that extends `RiJStateConcept`.
- A class that implements a reactive interface is implemented like any other reactive class with the following exceptions:
  - The class implements the reactive interface instead of `RiJStateConcept`.
  - If the reactive interface has triggered operations, the triggered operations must be redefined in the concrete class.

## JavaAnnotation

The `JavaAnnotation` property is used by the Rational Rhapsody code generator to insert Java annotations into generated code.

This property is used primarily for regenerating code that was reverse engineered. When you reverse engineer code that contains Java annotations, the value of the `JAVA_ReverseEngineering::ImplementationTrait::ImportJavaAnnotation` property determines how Rational Rhapsody handles the annotation code. If the value of this property is set to `Verbatim`, then Rational Rhapsody does not import annotations as model elements. Rather, the annotation code is stored as the value of the `JavaAnnotation` property. When code is later regenerated, it includes the code that was stored in this property.

*Default = Blank*

## MarkPrologEpilogInAnnotations

The MarkPrologEpilogInAnnotations property specifies whether to generate ignore annotations for the Specification, Implementation Prolog, and Epilog properties so they are ignored during roundtrip. When you insert code element declarations (variables, types, functions, and so on) in the Specification, Implementation Prolog, and Epilog properties, after a full roundtrip those elements are added to the model and are duplicated on the next code generation. When you use the MarkPrologEpilogInAnnotations property, you can have Rational Rhapsody automatically ignore the information specified in the Specification, Implementation Prolog, and Epilog properties instead of adding the ignore annotations manually. The possible values for the MarkPrologEpilogInAnnotations property are as follows:

- None - Rational Rhapsody does not generate any annotations. Any models created before Version 4.1 automatically have this property setting.
- Ignore - Rational Rhapsody generates the `///  
ignore` annotation before the code specified in the Specification, Implementation Prolog, and Epilog properties, and generates the `///  
]` annotation after the code specified in those properties.
- Auto - If the code in the Specification, Implementation Prolog, and Epilog properties is one line (it does not contain any newline characters (`\n`)), no annotations are generated (the same behavior has the None setting). If there is more than one line, Rational Rhapsody generates the `///  
ignore` annotation before the code specified in the Specification, Implementation Prolog, and Epilog properties, and generates the `///  
]` annotation after the code specified in those properties (the same behavior as the Ignore setting).

*During roundtrip, any ignore annotations in the comments of the element are not included in its description. Because the Specification, Implementation Prolog, and Epilog properties are generated between the annotation and its declaration for the element, you cannot rename those elements on roundtrip. If you change the name of an element, it is removed from the model and added with the new name. Some model information (for example, property settings) might be lost. (Default = Auto)*

## MaximumPendingEvents

The MaximumPendingEvents property specifies the maximum number of events that can be simultaneously pending in the event queue of the active class. The possible values are as follows:

- -1 - Memory is dynamically allocated.
- Positive integer - Specifies the maximum number of events.

*Default = -1*

## NestingVisibility

*The NestingVisibility property specifies the visibility of the generated specification of the nested class or package. (Default = Public)*

## ObjectTypeAsSingleton

The `ObjectTypeAsSingleton` property generates singleton code for object-types and actors. This functionality saves a singleton-type (actor) in its own repository unit, and manage that unit by using a configuration management tool. Set this property for a single object-type or higher. An object-type is generated as a singleton when all of the following conditions are met:

- The object-type has the «Singleton» stereotype.
- There is one and only one object of the object-type and the object multiplicity is 1.
- The `ObjectTypeAsSingleton` property is set to `True`.

*Note that when you expose a singleton object (for example, by creating a singleton object-type), Rational Rhapsody also modifies the code that is generated for the singleton. (Default = False)*

## OptimizeStatechartsWithoutEventsMemoryAllocation

*The `OptimizeStatechartsWithoutEventsMemoryAllocation` property determines whether the generated code uses dynamic memory allocation for statecharts that use only triggered operations. (Default = False)*

## Out

The `Out` property specifies how code is generated when the type is used with an argument that has the modifier "Out". The following table lists how classes are mapped as code when used with the `Out` modifier.

When a class is used with the "Out" modifier, the default is "\$type" in J.

## ReactiveThreadSettingPolicy

The `ReactiveThreadSettingPolicy` property specifies how threads are set for reactive classes. The possible values are as follows:

- `Default` - During code generation, Rational Rhapsody adds a thread argument to the constructor.
- `MainThread` - Rational Rhapsody does not add an argument; the thread is set to the main thread.
- `UserDefined` - Rational Rhapsody does not add an argument; you must set the value for the thread yourself.

*Default = Default*

## RecordTypeName

*The `RecordTypeName` property specifies the name of the class record type. If this is not set, Rational Rhapsody uses `class_name>_t`. (Default = empty string)*

## RefactorRenameRegularExpression

When you use the `Refactor:Rename` feature, Rhapsody searches the user-defined code in the model for all occurrences of the name of the specified element, including places where the element name appears as

part of the name of elements whose name is derived from the element that is being renamed.

To locate these elements, Rhapsody uses a set of regular expressions.

The regular expressions used for this search are defined by the property `RefactorRenameRegularExpression`.

Different values are used for this property, depending on what type of element you are renaming. For example, you will see that this property has a different value under `Class` than it does under `Attribute`. For some types of elements, Rhapsody uses a more generic expression, taken from the value of the property under `ModelElement`.

In general, the value of this property is a regular expression that consists of a number of "Or" components. For the renaming to work properly, each such component must contain three groupings, with the middle grouping using the variable `$keyword`, which represents the name of the element that is being renamed. In some cases, the expression takes this information from the value of another property rather than using the `$keyword` variable.

```
Default = (^its/[A-Za-z0-9_]its /^$<CG::Relation::Add>Its /^A-Za-z0-9_]$/<CG::Relation::Add>Its  
/^$<CG::Relation::Clear>Its /^A-Za-z0-9_]$/<CG::Relation::Clear>Its  
/^$<CG::Relation::CreateComponent>Its /^A-Za-z0-9_]$/<CG::Relation::CreateComponent>Its  
/^$<CG::Relation::DeleteComponent>Its /^A-Za-z0-9_]$/<CG::Relation::DeleteComponent>Its  
/^$<CG::Relation::Find>Its /^A-Za-z0-9_]$/<CG::Relation::Find>Its /^$<CG::Relation::Get>Its  
/^A-Za-z0-9_]$/<CG::Relation::Get>Its /^$<CG::Relation::Remove>Its  
/^A-Za-z0-9_]$/<CG::Relation::Remove>Its /^$<CG::Relation::Set>Its  
/^A-Za-z0-9_]$/<CG::Relation::Set>Its /^$<CG::Relation::GetKey>Its  
/^A-Za-z0-9_]$/<CG::Relation::GetKey>Its /^$<CG::Relation::GetAt>Its  
/^A-Za-z0-9_]$/<CG::Relation::GetAt>Its /^$<CG::Relation::RemoveKey>Its  
/^A-Za-z0-9_]$/<CG::Relation::RemoveKey>Its ) ($keyword:c)(^A-Za-z0-9_]_[1-9]+^A-Za-z0-9_]$/  
/(^A-Za-z0-9_]^(^$keyword)(^A-Za-z0-9_]$/)
```

## RelativeEventDataRecordTypeComponentsNaming

*The `RelativeEventDataRecordTypeComponentsNaming` property enables relative naming of event data record type components that represent events and triggered operation parameters. If this is `True`, no events or triggered operations share argument names because they would generate record components with the same name (which would not compile). (Default = `False`)*

## Renames

The `Renames` property enables one element to rename another element of the same type. You can also rename an element by using a renames dependency. In the case of a conflict, the dependency has precedence. Note the following:

- For attributes, this property works only for static attributes in a class or for attributes in a package.
- For operations, this property contains the name of the operation this operation is renaming. The signatures of the two operations must match.

(Default = empty string)

## ReturnType

The ReturnType property specifies how code is generated when the type is used as a return type. When a class is used with the "ReturnType" modifier, the default is "\$type" in J.

## SingletonExposeThis

*The SingletonExposeThis property, when set to False, specifies that all non-static methods are considered as static methods and does not have a this parameter passed in. (Default = False)*

## SpecificationEpilog

The SpecificationEpilog property allows you to add code to the end of the declaration of a model element.

*Default = Blank*

## SpecificationPragmas

*The SpecificationPragmas property specifies the user-defined pragmas to generate in the specification. (Default = Empty MultiLine)*

## SpecificationPragmasInContextClause

*The SpecificationPragmasInContextClause property specifies the user-defined pragmas to generate in the context clause of the specification. (Default = Empty MultiLine)*

## SpecificationProlog

The SpecificationProlog property allows you to add code to the beginning of the declaration of a model element.

For example, you could add the @Deprecated annotation for an element by entering @Deprecated and a new line as the value of this property.

*Default = Blank*

## SpecIncludes

The SpecIncludes property specifies the names (including full paths) of header files to be included at the top of specification files generated for classes (C++ and Java), objects or object types (C), and packages. Separate multiple file names by using commas, without spaces.

*Default = Empty string*

## Static

The Static property allows you to specify that an inner class should be declared as static. This allows you to instantiate the class outside the context of an object of the outer class.

Note that if you set the value of this property to True for a top-level class (a non-inner class), it does not affect the declaration generated for that class.

*Default = Cleared*

## TaskBody

*The TaskBody property defines an alternate task body for Ada Task and Ada Task Type classes. (Default = empty string)*

## TriggerArgument

The TriggerArgument property specifies how the type should be passed in when used as an argument for events\triggered operations. By default, classes that are used as an argument are passed by way of a pointer. There are 4 other properties that effect how types are passed into and returned by regular operations: "in", "out", "in/out" and "return." See also:

- In
- InOut
- Out

*Default = \$type*

## Visibility

The Visibility property specifies the visibility of the model element. Code generation maps the visibility specified for an element to the same visibility in the generated language. The possible values are as follows:

- Public - The model element is public.
- Protected - The model element is protected.
- Private - The element is private.

See “Visibility” for more information.

*Default = Public*

## Component

The Component metaclass contains properties that affect the Java component.

## InitializationScheme

*Default = ByPackage*

## Configuration

The Configuration metaclass contains properties that affect the configuration.

## ClassStateDeclaration

The ClassStateDeclaration property supports C compilers that cannot handle enum declarations inside struct declaration. The possible values are as follows:

- InClassDeclaration - Generate the reactive statechart enum declaration in the class declaration (as in Rational Rhapsody 3.0.1).
- BeforeClassDeclaration - Generate the reactive class statechart enum declaration before the declaration of the class.

*(Default = InClassDeclaration)*

## CodeGeneratorTool

The CodeGeneratorTool property specifies which code generation tool to use for the given configuration. The possible values are as follows:

- External - Use the registered, external code generator.
- Internal - Use the Rational Rhapsody internal code generator.

The default value is Internal.

## ContainerSet

The ContainerSet property specifies the container set used to implement relations.

The possible Java values are as follows: Java(1.1)Containers Java(1.2)Containers Java(1.5)Containers (Default)

## DefaultActiveGeneration

The DefaultActiveGeneration property specifies whether the default active class is created, as well as the classes for which it acts as the active context. The possible values are as follows:

- **Disable** - The default active singleton is not created.
- **ReactiveWithoutContext** - The default active singleton is created if there are reactive classes that consume events and do not have an active context explicitly specified. The default active singleton can handle only these classes.
- **All** - The default active singleton is generated if there is at least one event-consuming reactive class and the active singleton can handle all reactive classes that consume events - even those reactive classes that specify another active class as their active context.

*(Default = ReactiveWithoutContext)*

## DefaultImplementationDirectory

The DefaultImplementationDirectory property specifies the relative path to the default directory for generated implementation files. The value of this property is added after the configuration path. Consider the following case:

- File C.cpp is an implementation of class C mapped to a folder Foo.
- The active configuration (cfg) is under component cmp.
- DefaultImplementationDirectory is set to “src”

Rational Rhapsody generates C.cpp to root>\cmp\cfg\src\Foo. Note the following limitations:

- This feature is not supported in COM- or CORBA-related components (C++ only).
- The predefined OSE environments (OsePPCDiab and OseSfk) are not supported due to makefile flexibility issues.
- This feature is not supported by the INTEGRITY adapter build file generator.

*(Default = empty string)*

## DefaultSpecificationDirectory

The DefaultSpecificationDirectory property specifies the relative path to the default directory for generated specification files. The value of this property is added after the configuration path. Consider the following case:

- File B.h is a specification of class B that is not mapped to any file.
- The active configuration (cfg) is under component cmp.
- DefaultSpecificationDirectory is set to “inc”

Rational Rhapsody generates B.h to root>\cmp\cfg\inc. Note the following limitations:

- This feature is not supported in COM- or CORBA-related components (C++ only).
- The predefined OSE environments (OsePPCDiab and OseSfk) are not supported due to makefile flexibility issues.
- This feature is not supported by the INTEGRITY adapter build file generator.

*(Default = empty string)*



## DependencyRuleScheme

The DependencyRuleScheme property specifies how dependency rules should be generated in the makefile. The possible values are as follows:

- Basic - Generates only the local implementation and specification files in the dependency rule in the makefile.
- ByScope - In addition to generating the same files as the Basic option, generates the specification files of related elements (dependencies, associations, generalizations, and so on) that are in the scope of the active component.
- This option corresponds to the Rational Rhapsody 5.0.1 behavior.
- Extended - In addition to generating the same files as the ByScope option, generates the specification files of related external elements (specified by using the CG::Class::UseAsExternal and CG::Package::UseAsExternal properties) and elements that are not in the scope of the active component.

(Default = ByScope)

## DescriptionBeginLine

This property specifies the prefix for the beginning of comment lines in the generated code. This functionality uses a documentation system (such as Doxygen), which looks for a certain prefix to produce the documentation. This property affects only the code that is generated for descriptions of model elements; other auto-generated comments are not affected. The following table lists the default value for each language.

Language	Edition	Default Value	
C	"/"	C++	""

When you set this property, you should check the value of the lang\_CG::DiffDelimiter property - if the same prefix is used, Rational Rhapsody does not update the generated code when the description is modified. If both DescriptionBeginLine and DiffDelimiter use the same prefix, modify the values of the following properties under C\_CG::File:

DiffDelimiter	ImplementationHeader	SpecificationHeader
---------------	----------------------	---------------------

## DescriptionEndLine

This property specifies the prefix for the end of comment lines in the generated code. This functionality uses a documentation system (such as Doxygen), which looks for a certain prefix to produce the documentation. This property affects only the code that is generated for descriptions of model elements; other auto-generated comments are not affected. The following table lists the default value for each language.

Language	Edition	Default Value	
C	"/*"	C++	"/**/"

## EmptyArgumentListName

The `EmptyArgumentListName` specifies the string generated for the argument list when an operation has no arguments. For example, if you set this value to “void”, for an operation foo that has no arguments, Rational Rhapsody generates the following code:

```
int foo (void){ ...}
```

*(Default = empty string)*

## Environment

This property determines the target environment for a configuration. Generated code is targeted for that environment. See the Release Notes for the environments supported by Rational Rhapsody "out-of-the-box." "Out-of-the-box" support means that Rational Rhapsody includes a set of preconfigured code generation properties for the environment and precompiled versions of the relevant OXF libraries. The precompiled OXF libraries have been fully tested.

You can also add new environments, for example if you want to generate code for another RTOS. This involves retargeting the OS wrapper files in the Rational Rhapsody framework and creating a new set of code generation properties for the target environment.

*Default = JDK*

## ExternalGenerationTimeout

*The ExternalGenerationTimeout property specifies how long, in seconds, Rational Rhapsody waits for the each class in the configuration scope to complete so you can once again make changes to the model. This property applies to both the full-featured external generator and makefile generator. For example, if you set this property to 2 and you have 10 classes, Rational Rhapsody sets a timeout of 20. If the external code generator does not complete generation in this timeframe, Rational Rhapsody displays a message in the output window saying that the generator is not responding, and you are allowed to make changes to the model. If you set this property to 0, Rational Rhapsody does not time out the generation session, and waits for the code generator to complete its task - even if it takes forever. Rhapsody waits for a notification from the full-featured external code generator, or for the process termination of a makefile generator. (Default = 0)*

## ExternalGeneratorFileMappingRules

The `ExternalGeneratorFileMappingRules` property specifies whether the external code generator uses the same file mapping and naming scheme ( mapping rules ) as Rational Rhapsody. If the mapping rules are different , the external generator must implement handlers to the `GetFileName`, `GetMainFileName`, and `GetMakefileName` events that Rational Rhapsody runs to get a requested file name and path. The possible values are as follows:

- `AsRhapsody` - The external generator uses the same mapping rules as Rational Rhapsody.
- `DefinedByGenerator` - The external generator has its own mapping rules.

The default value is `AsRhapsody`.

## GeneratorExtraPropertyFiles

The GeneratorExtraPropertyFiles property opens the default Text Editor allowing the user to edit the \$OMROOT\CodeGenerator\GenerationRules\LangC\ReC\_CG.ini file.

## GeneratorRulesSet

The GeneratorRulesSet property specifies your own rules set.

*Default = Empty MultiLine*

## GeneratorScenarioName

The GeneratorScenarioName property specifies the scenario name for the rule, if you write your own set of code generation rules.

*Default = Empty string*

## GenericEventHandling

The GenericEventHandling property is a Boolean value that determines whether to generate generic event-handling code. This property supports large-scale collaboration, where you might not be aware of which classes consume a base event of your part in the event hierarchy, and might not have access to parts of the model that use base events.

The framework base event class includes a new, virtual method that checks the event ID against the specified ID, thereby supplying a generic mechanism for events without super events.

For Java, the specific method is as follows: Boolean isTypeOf(long id) {return lId == id;}

Each generated event that has a super event overrides the method to check the ID against its own ID, then calls its base event directly to continue the check. An event without a base event returns Cleared if the ID does not equal its own. When you set the GenericEventHandling property to Cleared, event consumption code is generated as in version 3.0.1. Setting this property affects only the way events are consumed - the override on the isTypeOf() method is still generated, to allow handling of events in components that use the generic event handling. To support complete generic event handling, you should regenerate the code for all events and reactive classes.

*Default = Checked*

## ImplementationEpilog

The ImplementationEpilog property adds any code that you want to be added as verbatim text (to be ignored by Rational Rhapsody) to the end of the definition of a model element. For example, you could wrap a section of code with an #ifdef-#endif pair, add compiler-specific keywords, or add a #pragma statement. For example, to specify that an operation is available only when the code is compiled with

`_DEBUG`, set the following properties for the operation:

- Set `SpecificationProlog` to `#ifdef _DEBUG cr.`
- Set `SpecificationEpilog` to `#endif.`
- Set `ImplementationProlog` to `#ifdef _DEBUG cr.`
- Set `ImplementationEpilog` to `#endif.`

The following table lists whether leading and trailing linefeeds are generated.

Metaclass	Leading Linefeed Added?	Class	Yes	Package	No
-----------	-------------------------	-------	-----	---------	----

*(Default = Empty MultiLine)*

## ImplementationProlog

The `ImplementationProlog` property adds any code that you want to be added as verbatim text (to be ignored by Rational Rhapsody) to the beginning of the definition of a model element. For example, you could wrap a section of code with an `#ifdef-#endif` pair, add compiler-specific keywords, or add a `#pragma` statement. For example, to specify that an operation is available only when the code is compiled with `_DEBUG`, set the following properties for the operation:

- Set `SpecificationProlog` to `#ifdef _DEBUG cr.`
- Set `SpecificationEpilog` to `#endif.`
- Set `ImplementationProlog` to `#ifdef _DEBUG cr.`
- Set `ImplementationEpilog` to `#endif.`

The following table lists whether leading and trailing linefeeds are generated.

*(Default = Empty MultiLine)*

## InitializeEmbeddableObjectsByValue

*The `InitializeEmbeddableObjectsByValue` property specifies whether embeddable classes and object types selected in the configuration initial instances list should be allocated by value in the `main()` routine.*  
*(Default = False)*

## JarFileGenerate

Boolean property that determines whether or not a JAR file is generated as part of the build process. The value of this property is controlled by the "Generate JAR File" option on the Settings tab of the Features window for configurations.

*Default = Cleared*

## JarFileGeneratorCommand

Specifies the jar command that should be carried out if the JarFileGenerate property has been set to True.

## LocalVariablesDeclaration

*The LocalVariablesDeclaration property specifies variables that you want to appear in the declaration of the entrypoint or operation. (Default = Empty MultiLine)*

## MainFunctionArgList

This property provides a list of the main function arguments.

*Default = String[] args*

## MarkPrologEpilogInAnnotations

The MarkPrologEpilogInAnnotations property specifies whether to generate ignore annotations for the Specification, Implementation Prolog, and Epilog properties so they are ignored during roundtrip. When you insert code element declarations (variables, types, functions, and so on) in the Specification, Implementation Prolog, and Epilog properties, after a full roundtrip those elements are added to the model and are duplicated on the next code generation. When you use the MarkPrologEpilogInAnnotations property, you can have Rational Rhapsody automatically ignore the information specified in the Specification, Implementation Prolog, and Epilog properties instead of adding the ignore annotations manually. The possible values for the MarkPrologEpilogInAnnotations property are as follows:

- None - Rational Rhapsody does not generate any annotations. Any models created before Version 4.1 automatically have this property setting.
- Ignore - Rational Rhapsody generates the `/// ignore annotation before the code specified in the Specification, Implementation Prolog, and Epilog properties, and generates the /// annotation after the code specified in those properties.`
- Auto - If the code in the Specification, Implementation Prolog, and Epilog properties is one line (it does not contain any newline characters `\n`), no annotations are generated (the same behavior has the None setting). If there is more than one line, Rational Rhapsody generates the `/// ignore annotation before the code specified in the Specification, Implementation Prolog, and Epilog properties, and generates the /// annotation after the code specified in those properties (the same behavior as the Ignore setting).`

*During roundtrip, any ignore annotations in the comments of the element are not included in its description. Because the Specification, Implementation Prolog, and Epilog properties are generated between the annotation and its declaration for the element, you cannot rename those elements on roundtrip. If you change the name of an element, it is removed from the model and added with the new name. Some model information (for example, property settings) might be lost. (Default = Auto)*

## MarkStatechartDescriptionAsComment

Prior to release 8.2, comment symbols were not generated correctly for descriptions of statechart elements (states, transitions, entry/exit actions) and associated requirements, resulting in compilation errors. This issue was corrected in 8.2.

To preserve the previous code generation behavior for pre-8.2 models, the property `JAVA_CG::Configuration::MarkStatechartDescriptionAsComment` was added to the backward compatibility settings for Java with a value of `False`.

## ShowCgSimplifiedModelPackage

The first step of the code generation process consists of the building of a simplified model based on the Rational Rhapsody model.

By default, the simplified model is not displayed in Rational Rhapsody. To have the simplified model displayed in the browser, set the `ShowCgSimplifiedModelPackage` property property to `True`. Once you have done so, the next time you generate code, the simplified model is added automatically at the top of the project tree in the browser.

*Default = Cleared*

## SourceListFile

The `SourceListFile` property specifies the name of the file containing a list of .java source files to be compiled with javac. The batch file used by the Build command (jdkmake.bat) can use the following call, rather than including a long list of source files: `javac -g @files.lst` This same command is generated from the following line in the `MakeFileContent` property for Java: `javac -g @$SourceListFile` If the `SourceListFile` property is empty, `$SourceListFile` is replaced with a string containing all source file names, separated by spaces (for example, "A.java B.java"). This means that if the `MakeFileContent` default value is not changed, you get: `javac -g @A.java B.java ...` If you do not want to use the file containing the list of sources, you must also change the `MakeFileContent` property to replace "`javac -g @$SourceListFile`" with "`javac -g $SourceListFile`".

*Default = files.lst*

## SpecificationEpilog

The `SpecificationEpilog` property allows you to add code to the end of the declaration of a model element.

*Default = Blank*

## SpecificationProlog

The `SpecificationProlog` property allows you to add code to the beginning of the declaration of a model element.

For example, you could add the `@Deprecated` annotation for an element by entering `@Deprecated` and a new line as the value of this property.

*Default = Blank*

# Dependency

The Dependency metaclass controls the dependency for a package that defines a namespace.

## CreateUseStatement

*The CreateUseStatement property determines whether a use statement is added to the code after the with statement. The supplier of the dependency must be a class or type. (Default = False)*

## GenerateOriginComment

When set to True, generates a comment before #include statements that indicate which element "caused" the #include.

## GeneratePragmaElaborate

*The GeneratePragmaElaborate property determines whether to generate an elaborate pragma for the supplier class in the client class or package. (Default = False)*

## GeneratePragmaElaborateAll

*The GeneratePragmaElaborateAll property determines whether to generate a pre-elaborate pragma for the supplier class in the client class or package. (Default = False)*

## GenerateWithClause

*The GenerateWithClause property determines whether with clauses are generated for Usage dependencies. For example, you can generate a with clause for a package, P1, in the specification of another package, P2, by using a dependency, D1, and generate a use clause for P1 in the body of P2. In addition, this functionality is useful for modeling inherited annotations across classes and packages. (Default = Checked)*

## ImplementationEpilog

The ImplementationEpilog property adds any code that you want to be added as verbatim text (to be ignored by Rational Rhapsody) to the end of the definition of a model element. For example, you could wrap a section of code with an #ifdef-#endif pair, add compiler-specific keywords, or add a #pragma statement. For example, to specify that an operation is available only when the code is compiled with \_DEBUG, set the following properties for the operation:

- Set SpecificationProlog to #ifdef \_DEBUG cr.
- Set SpecificationEpilog to #endif.

- Set ImplementationProlog to `#ifdef _DEBUG` cr.
- Set ImplementationEpilog to `#endif`.

The following table lists whether leading and trailing linefeeds are generated.

Metaclass Leading Linefeed Added? Class Yes Package No

*(Default = Empty MultiLine)*

## ImplementationProlog

The ImplementationProlog property adds any code that you want to be added as verbatim text (to be ignored by Rational Rhapsody) to the beginning of the definition of a model element. For example, you could wrap a section of code with an `#ifdef-#endif` pair, add compiler-specific keywords, or add a `#pragma` statement. For example, to specify that an operation is available only when the code is compiled with `_DEBUG`, set the following properties for the operation:

- Set SpecificationProlog to `#ifdef _DEBUG` cr.
- Set SpecificationEpilog to `#endif`.
- Set ImplementationProlog to `#ifdef _DEBUG` cr.
- Set ImplementationEpilog to `#endif`.

The following table lists whether leading and trailing linefeeds are generated.

*(Default = Empty MultiLine)*

## IncludeStyle

The IncludeStyle property controls the style of `#include` statements. When you use this property, you can control the style of a specific dependency, or the entire configuration/component/project. To set the style for include files that are synthesized based on associations between model elements (for example, setting the type of some attribute to a class), add a «Usage» dependency between the elements and set this property to the appropriate value. The possible values are as follows:

- Default - Use angle brackets for include statements for external elements, and quotes for include statements for other elements.
- Quotes - Enclose include files in quotation marks. For example: `#include "A.h"`
- When a compiler encounters an include file in quotes, it searches for the file in both the current directory and the directories specified in the include path. Note that the specific algorithm used is compiler-dependent.
- AngledBrackets - Enclose include files in angle brackets. For example: `#include A.h`
- When a compiler encounters an include file in angle brackets, it searches for the file only in the directories specified in the include path.
- If you set the property to AngledBrackets at the configuration level, you must also change the `CG::File::IncludeScheme` property to `RelativeToConfiguration` to ensure successful compilation.

*(Default = Default)*



## MarkPrologEpilogInAnnotations

The MarkPrologEpilogInAnnotations property specifies whether to generate ignore annotations for the Specification, Implementation Prolog, and Epilog properties so they are ignored during roundtrip. When you insert code element declarations (variables, types, functions, and so on) in the Specification, Implementation Prolog, and Epilog properties, after a full roundtrip those elements are added to the model and are duplicated on the next code generation. When you use the MarkPrologEpilogInAnnotations property, you can have Rational Rhapsody automatically ignore the information specified in the Specification, Implementation Prolog, and Epilog properties instead of adding the ignore annotations manually. The possible values for the MarkPrologEpilogInAnnotations property are as follows:

- None - Rational Rhapsody does not generate any annotations. Any models created before Version 4.1 automatically have this property setting.
- Ignore - Rational Rhapsody generates the `///  
ignore` annotation before the code specified in the Specification, Implementation Prolog, and Epilog properties, and generates the `///  
end` annotation after the code specified in those properties.
- Auto - If the code in the Specification, Implementation Prolog, and Epilog properties is one line (it does not contain any newline characters (`\n`)), no annotations are generated (the same behavior has the None setting). If there is more than one line, Rational Rhapsody generates the `///  
ignore` annotation before the code specified in the Specification, Implementation Prolog, and Epilog properties, and generates the `///  
end` annotation after the code specified in those properties (the same behavior as the Ignore setting).

*During roundtrip, any ignore annotations in the comments of the element are not included in its description. Because the Specification, Implementation Prolog, and Epilog properties are generated between the annotation and its declaration for the element, you cannot rename those elements on roundtrip. If you change the name of an element, it is removed from the model and added with the new name. Some model information (for example, property settings) might be lost. (Default = Auto)*

## SpecificationEpilog

The SpecificationEpilog property allows you to add code to the end of the declaration of a model element.

*Default = Blank*

## SpecificationProlog

The SpecificationProlog property allows you to add code to the beginning of the declaration of a model element.

For example, you could add the `@Deprecated` annotation for an element by entering `@Deprecated` and a new line as the value of this property.

*Default = Blank*

## Static

The Static property allows you to specify that a dependency should be generated as a static import.

When you apply the StaticImport stereotype to a dependency, the value of this property is set to True.

*Default = Cleared*

## UseNamespace

*The UseNamespace property allows you to model namespace usage. When you set a dependency to a package that defines a namespace and set this property to True, Rational Rhapsody generates a “using namespace” statement to the package namespace. (Default = False)*

## Event

The Event metaclass contains properties that control events.

## AnimInstanceCreate

*The AnimInstanceCreate property affects event creation. If you set the C\_CG::Event::NoDynamicAllocAnimCreate property to False, Rational Rhapsody does not generate the event creation method, effectively disabling the ability to inject the event in animation. To enable the injection of the event, you can specify a different method to obtain an instance of the event by setting this property to the name of the method to use. (Default = empty string)*

## DeclarationModifier

*The DeclarationModifier property adds a string to the class or event declaration. The string is displayed between the class keyword and the class name in the generated code. For example, for a class A, the DeclarationModifier would appear as follows: class DeclarationModifier> A {...}; This property adds a modifier to the class declaration. For example, if you have a class myExportableClass that is exported from a DLL by using the MYDLL\_API macro, you can set the DeclarationModifier property to “MYDLL\_API.” The generated code would then be as follows: class MYDLL\_API myExportableClass {...}; This property supports two keywords: \$component and \$class. (Default = empty string)*

## DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text

generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- `$Name` - the element name
- `$FullName` - the full path of the element (`P1::P2::C.a`)
- `$Description` - the text provided on the Description tab of the Features window for the element
- `$Requirements` - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the `$Requirements` keyword in the `DescriptionTemplate` property to get such comments generated for specific model elements.
- `$Login` - the login name of the user who generated the file
- `$CodeGeneratedDate` - the date on which the code was generated
- `$CodeGeneratedTime` - the time at which the code was generated
- `$RhapsodyVersion` - the version of Rhapsody that was used to generate the file
- For operations: `$Signature` - the operation signature, `$Arguments` - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: `$Type` - the type of the attribute
- For arguments: `$Type` - the argument type, `$Direction` - the argument direction (In, Out, or InOut)
- For events: `$Arguments` - the text provided on the Description tab of the Features window for the event arguments
- For association ends: `$Target` - the other end of the association
- For requirements: `$ID` - the ID specified for the requirement, `$Specification` - the specification text for the requirement
- The values of tags can be included by using the syntax `$tag_name`. For example, if you use a tag called `author` for your classes, you can use the keyword `$author` in the value of `DescriptionTemplate` in order to have the value of the `author` tag included in the element description.
- The values of properties can be included by using the syntax `$property_name`. For example, you can use the keyword `$Animate` in the value of `DescriptionTemplate` in order to have the value of the `Animate` property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as `$Name`)
- Tag keywords
- Property keywords

If the value of a keyword is a `MultiLine` value, then each new line (except the first one) starts with the value of the `<lang>_CG::Configuration::DescriptionBeginLine` property, and each line ends with the value of the `<lang>_CG::Configuration::DescriptionEndLine` property.

```
Default = [[ * $Description]] [[ * @author $Author]] [[ * @version $Version]] [[ * @see $See]] [[ *  
@since $Since]]
```

## In

The In property determines the exact syntax used when an event is used as an "in" parameter for an operation.

*Default = final \$type*

## InOut

The InOut property determines the exact syntax used when an event is used as an "in/out" parameter for an operation.

*Default = \$type*

## Out

The Out property determines the exact syntax used when an event is used as an "out" parameter for an operation.

*Default = \$type*

## ReturnType

The ReturnType property determines the exact syntax used when an event is used as the return type of an operation.

*Default = \$type*

# File

The File metaclass contains properties that control the generated code files.

## DiffDelimiter

When code is generated, new files are generated only where a before vs. after comparison indicates that the code has changed. When dealing with comments in the code, there are cases where you may decide that a change is not significant enough to justify regeneration of the file. For example, if you record the author of a file as a comment in the file, you may decide that the file should not be regenerated if the only change is the name of the author. The property DiffDelimiter can be used to mark such insignificant changes. When you use the string specified for the DiffDelimiter property somewhere in your code, the text to the right of the delimiter will be ignored when the code comparison is done prior to the regeneration of files.

For example, the default value of the property `CPP_CG::File::SpecificationHeader` includes the following text:

```
//! Generated Date: $CodeGeneratedDate
```

So if the only change in the code is the code generation date, the file will not be regenerated.

Note that the delimiter can be used at the beginning of a line (in which case the entire line will be ignored in the comparison) or in the middle of a line (in which case only the text to the right of the delimiter will be ignored).

If you do not want to use this feature in your model, change the value of the property to blank.

*Default =* `//!`

## Footer

The Footer property specifies a multiline footer that is added to the end of generated Java files.

*Default =*

```
"/***** File Path:
$FullCodeGeneratedFileName *****/"
```

Footer format strings can contain any of the following keywords:

- `$ProjectName` - The project name.
- `$ComponentName` - The component name.
- `$ConfigurationName` - The configuration name.
- `$ModelElementName` - The name of the element mapped to the file. If there is more than one, this is the name of the first element.
- `$FullModelElementName` - The name of the element mapped to the file, including the full path. If there is more than one, this is the name of the first element.
- `$CodeGeneratedDate` - The generation date.
- `$CodeGeneratedTime` - The generation time.
- `$RhapsodyVersion` - The version of Rational Rhapsody that generated the file.
- `$Login` - The user who generated the file.
- `$CodeGeneratedFileName` - The name of the generated file.
- `$FullCodeGeneratedFileName` - The full file name.
- `$Tag` - The value of the tag for the specified element
- `$Property` - The value of the element property with the specified name.

To avoid redundant compilation, Rational Rhapsody avoids unnecessary changes to specific lines prefixed with a special string, defined by the `CPP_CG::File::DiffDelimiter` property. The default `DiffDelimiter` value is `"/!"`.

## Generate

The Generate property is used to specify whether code should be generated for a Class or File element. For Java code generation, this is a Boolean property. For C and C++, there are also property values that can be used to generate only the specification file or only the implementation file.

The possible values are:

- True - for C and C++ both specification and implementation files are generated
- False - no files are generated
- Specification (C, C++) - only the specification file is generated
- Implementation (C, C++) - only the implementation file is generated

*Default = True*

## Header

The Header property specifies a multiline header that is added to the top of all generated Java files.

*Default =*

```
/***** Rhapsody : $RhapsodyVersion  
Login : $Login Component : $ComponentName Configuration : $ConfigurationName Model Element :  
$FullModelElementName //! Generated Date : $CodeGeneratedDate File Path :  
$FullCodeGeneratedFileName *****/
```

Header format strings can contain any of the following keywords:

- \$ProjectName - The project name.
- \$ComponentName - The component name.
- \$ConfigurationName - The configuration name.
- \$ModelElementName - The name of the element mapped to the file. If there is more than one, this is the name of the first element.
- \$FullModelElementName - The name of the element mapped to the file, including the full path. If there is more than one, this is the name of the first element.
- \$CodeGeneratedDate - The generation date.
- \$CodeGeneratedTime - The generation time.
- \$RhapsodyVersion - The version of Rational Rhapsody that generated the file.
- \$Login - The user who generated the file.
- \$CodeGeneratedFileName - The name of the generated file.
- \$FullCodeGeneratedFileName - The full file name.
- \$Tag - The value of the tag for the specified element.
- \$Property - The value of the element property with the specified name.

To avoid redundant compilation, Rational Rhapsody avoids unnecessary changes to specific lines prefixed

with a special string, defined by the `CPP_CG::File::DiffDelimiter` property. The default `DiffDelimiter` value is `"//!"`.

## ImplementationEpilog

The `ImplementationEpilog` property adds any code that you want to be added as verbatim text (to be ignored by Rational Rhapsody) to the end of the definition of a model element. For example, you could wrap a section of code with an `#ifdef-#endif` pair, add compiler-specific keywords, or add a `#pragma` statement. For example, to specify that an operation is available only when the code is compiled with `_DEBUG`, set the following properties for the operation:

- Set `SpecificationProlog` to `#ifdef _DEBUG cr.`
- Set `SpecificationEpilog` to `#endif.`
- Set `ImplementationProlog` to `#ifdef _DEBUG cr.`
- Set `ImplementationEpilog` to `#endif.`

The following table lists whether leading and trailing linefeeds are generated for the `ImplementationEpilog` metaclass.

Leading Linefeed Added? Class Yes Package No

*(Default = Empty MultiLine)*

## ImplementationFooter

The `ImplementationFooter` property specifies the multiline footer to be generated at the end of implementation files. The default footer template for Ada is an empty `MultiLine`; the default for C and C++ is as follows:

```
/****** File Path:
$FullCodeGeneratedFileName *****/
```

Footer format strings can contain any of the following keywords:

- `$ProjectName` - The project name.
- `$ComponentName` - The component name.
- `$ConfigurationName` - The configuration name.
- `$ModelElementName` - The name of the element mapped to the file. If there is more than one, this is the name of the first element.
- `$FullModelElementName` - The name of the element mapped to the file, including the full path. If there is more than one, this is the name of the first element.
- `$CodeGeneratedDate` - The generation date.
- `$CodeGeneratedTime` - The generation time.
- `$RhapsodyVersion` - The version of Rational Rhapsody that generated the file.
- `$Login` - The user who generated the file.
- `$CodeGeneratedFileName` - The name of the generated file.

- `$FullCodeGeneratedFileName` - The full file name.
- `$Tag` - The value of the tag for the specified element.
- `$Property` - The value of the element property with the specified name.

To avoid redundant compilation, Rational Rhapsody avoids unnecessary changes to specific lines prefixed with a special string, defined by the `CPP_CG::File::DiffDelimiter` property. The default `DiffDelimiter` value is `“//!”`. The keywords are resolved in the following order:

- Predefined keywords
- Property keywords
- Tag keywords

Note the following:

- Keyword names can be written in parentheses. For example: `$(Name)`
- If the value of a keyword is a `MultiLine`, each new line (except the first one) starts with the value of the `lang_CG::Configuration::DescriptionBeginLine` property; each line ends with the value of the `CPP_CG::Configuration::DescriptionEndLine` property.

## ImplementationHeader

The `ImplementationHeader` property specifies the multiline header that is generated at the beginning of implementation files. The default header template for Ada is an empty `MultiLine`; the default for C and C++ is as follows:

```

/***** Rhapsody: $RhapsodyVersion
Login: $Login Component: $ComponentName Configuration: $ConfigurationName Model Element:
$FullModelElementName //! Generated Date: $CodeGeneratedDate File Path:
$FullCodeGeneratedFileName *****/

```

Header format strings can contain any of the following keywords:

- `$ProjectName` - The project name.
- `$ComponentName` - The component name.
- `$ConfigurationName` - The configuration name.
- `$ModelElementName` - The name of the element mapped to the file. If there is more than one, this is the name of the first element.
- `$FullModelElementName` - The name of the element mapped to the file, including the full path. If there is more than one, this is the name of the first element.
- `$CodeGeneratedDate` - The generation date.
- `$CodeGeneratedTime` - The generation time.
- `$RhapsodyVersion` - The version of Rational Rhapsody that generated the file.
- `$Login` - The user who generated the file.
- `$CodeGeneratedFileName` - The name of the generated file.
- `$FullCodeGeneratedFileName` - The full file name.
- `$Tag` - The value of the tag for the specified element.
- `$Property` - The value of the element property with the specified name.



To avoid redundant compilation, Rational Rhapsody avoids unnecessary changes to specific lines prefixed with a special string, defined by the `CPP_CG::File::DiffDelimiter` property. The default `DiffDelimiter` value is `“//!”`. The keywords are resolved in the following order:

- Predefined keywords
- Property keywords
- Tag keywords

Note the following:

- Keyword names can be written in parentheses. For example: `$(Name)`
- If the value of a keyword is a `MultiLine`, each new line (except the first one) starts with the value of the `CPP_CG::Configuration::DescriptionBeginLine` property; each line ends with the value of the `CPP_CG::Configuration::DescriptionEndLine` property.

## ImplementationProlog

The `ImplementationProlog` property adds any code that you want to be added as verbatim text (to be ignored by Rational Rhapsody) to the beginning of the definition of a model element. For example, you could wrap a section of code with an `#ifdef-#endif` pair, add compiler-specific keywords, or add a `#pragma` statement. For example, to specify that an operation is available only when the code is compiled with `_DEBUG`, set the following properties for the operation:

- Set `SpecificationProlog` to `#ifdef _DEBUG cr.`
- Set `SpecificationEpilog` to `#endif.`
- Set `ImplementationProlog` to `#ifdef _DEBUG cr.`
- Set `ImplementationEpilog` to `#endif.`

The following table lists whether leading and trailing linefeeds are generated for the metaclasses.

Metaclass	Trailing Linefeed Added?	Class	No	Package	Yes
-----------	--------------------------	-------	----	---------	-----

(Default = Empty MultiLine)

## MarkPrologEpilogInAnnotations

The `MarkPrologEpilogInAnnotations` property specifies whether to generate ignore annotations for the `Specification`, `Implementation Prolog`, and `Epilog` properties so they are ignored during roundtrip. When you insert code element declarations (variables, types, functions, and so on) in the `Specification`, `Implementation Prolog`, and `Epilog` properties, after a full roundtrip those elements are added to the model and are duplicated on the next code generation. When you use the `MarkPrologEpilogInAnnotations` property, you can have Rational Rhapsody automatically ignore the information specified in the `Specification`, `Implementation Prolog`, and `Epilog` properties instead of adding the ignore annotations manually. The possible values for the `MarkPrologEpilogInAnnotations` property are as follows:

- None - Rational Rhapsody does not generate any annotations. Any models created before Version 4.1 automatically have this property setting.
- Ignore - Rational Rhapsody generates the `///[ ignore annotation before the code specified in the Specification, Implementation Prolog, and Epilog properties, and generates the ///] annotation after the code specified in those properties.`

- Auto - If the code in the Specification, Implementation Prolog, and Epilog properties is one line (it does not contain any newline characters (\n)), no annotations are generated (the same behavior has the None setting). If there is more than one line, Rational Rhapsody generates the `/// ignore annotation before the code specified in the Specification, Implementation Prolog, and Epilog properties, and generates the /// annotation after the code specified in those properties (the same behavior as the Ignore setting).`

*During roundtrip, any ignore annotations in the comments of the element are not included in its description. Because the Specification, Implementation Prolog, and Epilog properties are generated between the annotation and its declaration for the element, you cannot rename those elements on roundtrip. If you change the name of an element, it is removed from the model and added with the new name. Some model information (for example, property settings) might be lost. (Default = Auto)*

## SpecificationEpilog

The SpecificationEpilog property allows you to add code to the end of the declaration of a model element.

*Default = Blank*

## SpecificationFooter

The SpecificationFooter property specifies the multiline footer to be generated at the end of specification files. The default footer template for Ada is an empty MultiLine; the default for C and C++ is as follows:

```

/***** File Path:
$FullCodeGeneratedFileName *****/

```

Footer format strings can contain any of the following keywords:

- \$ProjectName - The project name.
- \$ComponentName - The component name.
- \$ConfigurationName - The configuration name.
- \$ModelElementName - The name of the element mapped to the file. If there is more than one, this is the name of the first element.
- \$FullModelElementName - The name of the element mapped to the file, including the full path. If there is more than one, this is the name of the first element.
- \$CodeGeneratedDate - The generation date.
- \$CodeGeneratedTime - The generation time.
- \$RhapsodyVersion - The version of Rational Rhapsody that generated the file.
- \$Login - The user who generated the file.
- \$CodeGeneratedFileName - The name of the generated file.
- \$FullCodeGeneratedFileName - The full file name.
- \$Tag - The value of the tag for the specified element.
- \$Property - The value of the element property with the specified name.

To avoid redundant compilation, Rational Rhapsody avoids unnecessary changes to specific lines prefixed with a special string, defined by the `CPP_CG::File::DiffDelimiter` property. The default DiffDelimiter

value is “//!”. The keywords are resolved in the following order:

- Predefined keywords (such as \$Name)
- Property keywords
- Tag keywords

Note the following:

- Keyword names can be written in parentheses. For example: \$(Name)
- If the value of a keyword is a MultiLine, each new line (except the first one) starts with the value of the lang\_CG::Configuration::DescriptionBeginLine property; each line ends with the value of the CPP\_CG::Configuration::DescriptionEndLine property.

## SpecificationHeader

The SpecificationHeader property specifies the multiline header to be generated at the beginning of specification files.

Header format strings can contain any of the following keywords:

- \$ProjectName - The project name.
- \$ComponentName - The component name.
- \$ConfigurationName - The configuration name.
- \$ModelElementName - The name of the element mapped to the file. If there is more than one, this is the name of the first element.
- \$FullModelElementName - The name of the element mapped to the file, including the full path. If there is more than one, this is the name of the first element.
- \$CodeGeneratedDate - The generation date.
- \$CodeGeneratedTime - The generation time.
- \$RhapsodyVersion - The version of Rational Rhapsody that generated the file.
- \$Login - The user who generated the file.
- \$CodeGeneratedFileName - The name of the generated file.
- \$FullCodeGeneratedFileName - The full file name.
- \$Tag - The value of the tag for the specified element.
- \$Property - The value of the element property with the specified name.

To avoid redundant compilation, Rational Rhapsody avoids unnecessary changes to specific lines prefixed with a special string, defined by the CPP\_CG::File::DiffDelimiter property. The default DiffDelimiter value is “//!”. The keywords are resolved in the following order:

- Predefined keywords (such as \$Name)
- Property keywords
- Tag keywords

Note the following:

- Keyword names can be written in parentheses. For example: \$(Name)

- If the value of a keyword is a MultiLine, each new line (except the first one) starts with the value of the lang\_CG::Configuration::DescriptionBeginLine property; each line ends with the value of the CPP\_CG::Configuration::DescriptionEndLine property.

## SpecificationProlog

The SpecificationProlog property adds code to the beginning of the declaration of a model element (such as a configuration or class). For example, to create an abstract class in Java, you can set the SpecificationProlog property for the class to “abstract.” You must include the space after the word “abstract.” If the visibility for the class is set to default, the following class declaration is generated in the .java file: abstract class classname {...} The SpecificationProlog property allows you to add compiler-specific keywords, add a #pragma statement, or wrap a section of code with an #ifdef-#endif pair. For example, to specify that an operation is available only when the code is compiled with \_DEBUG, set the following properties for the operation:

- Set SpecificationProlog to #ifdef \_DEBUG cr.
- Set SpecificationEpilog to #endif.
- Set ImplementationProlog to #ifdef \_DEBUG cr.
- Set ImplementationEpilog to #endif.

The following table lists whether leading and trailing linefeeds are generated.

Metaclass	Leading Linefeed Added?	Trailing Linefeed Added?	Class	Yes	No	Package	Yes	Yes
-----------	-------------------------	--------------------------	-------	-----	----	---------	-----	-----

(Default = Empty MultiLine)

## Framework

The Framework metaclass contains properties that affect the Rational Rhapsody framework.

### ActivateFrameworkDefaultEventLoop

*The ActivateFrameworkDefaultEventLoop property specifies the framework call that initializes the framework main event loop. (Default = OXF::start(\$Fork);) The value of \$Fork is calculated from the CG::Configuration::StartFrameworkInMainThread property for regular applications and from the CORBA::Configuration::StartFrameworkInMainThread property for CORBA servers. This property can be set at the configuration level or higher.*

### ActiveBase

The ActiveBase property specifies the superclass from which to specialize all threads, if the ActiveBaseUsage property is set to Checked.

*Default = RiJThread*

## ActiveBaseIncludeFiles

This property specifies a comma separated list of packages or classes to be imported from the framework to support active classes.

*Default = com.ibm.rational.rhapsody.oxf.\**

## ActiveBaseUsage

The ActiveBaseUsage property specifies whether to use the superclass specified by the ActiveBase property as the superclass for all threads.

*Default = Checked*

## ActiveDestructorGuard

*The ActiveDestructorGuard property specifies the macro that starts protection for an active user object destructor. (Default = START\_DTOR\_THREAD\_GUARDED\_SECTION)*

## ActiveExecuteOperationName

The ActiveExecuteOperationName property sets the user object virtual table for an active object and passes it to a task in the task initialization function (RiCTask\_init()). Follow these steps:

- Create a method with the following signature: struct RiCReactive \* operation name> (RiCTask \* const)
- Set the operation name in the ActiveExecuteOperationName property.
- Start the execution of the active object task by calling the RICTASK\_START() macro on the object.

*The virtual function table member name is stored in the ActiveVtblName property. (Default = empty string)*

## ActiveGuardInitialization

*The ActiveGuardInitialization property specifies the call that makes the active object event dispatching guarded. (Default = SetToGuardThread)*

## ActiveIncludeFiles

The ActiveIncludeFiles property specifies the base class for threads when you use selective framework includes. If a class is active and this property is defined, the file specified by the property is included in the class specification file. The default value for C++ is as follows: oxf/omthread.h The default value for C is as follows: oxf/RiCTask.h

## ActiveInit

The ActiveInit property specifies the format of the declaration generated for the initializer for an active class. The default value for Ada is an empty string.

*Default = m\_thread = new \$base("\$class");*

## ActiveMessageQueueSize

The ActiveMessageQueueSize property specifies the size of the message queue allocated for active objects, if the ActiveMessageQueueSize property for classes is left blank.

*Default = Empty string*

## ActiveStackSize

The ActiveStackSize property specifies the size of the stack allocated for active objects, if the ActiveStackSize property for classes is left blank.

*Default = Empty string*

## ActiveThreadName

The ActiveThreadName property specifies the name of threads, if the ActiveThreadName property for classes is left blank.

*Default = ""*

## ActiveThreadPriority

The ActiveThreadPriority priority specifies the priority of threads, if the ActiveThreadPriority property for classes is left blank.

*Default = Empty string*

## ActiveVtblName

*The ActiveVtblName property stores the name of the virtual function table that is associated with a task (the RiCTask member of the structure). (Default = \$ObjectName\_activeVtbl)*

## BooleanType

*The BooleanType property specifies the Boolean type used by the framework. (Default = bool)*

## CurrentEventId

*The CurrentEventId property specifies the call or macro used to obtain the ID of the currently consumed event. (Default = OM\_CURRENT\_EVENT\_ID)*

## DefaultProvidedInterfaceName

The DefaultProvidedInterfaceName property specifies the interface that must be implemented by the "in" part of a rapid port. See the Rational Rhapsody help for more information on rapid ports.

*Default = DefaultProvidedInterface*

## DefaultReactivePortBase

The DefaultReactivePortBase property stores the base class for the generic rapid port (or default reactive port). This base class relays all events. See the Rational Rhapsody help for more information on rapid ports.

*Default = RiJDefaultReactivePort*

## DefaultReactivePortIncludeFiles

The DefaultReactivePortIncludeFiles property specifies the include files that are referenced in the generated file that implements the class with the rapid ports. See the Rational Rhapsody help for more information on rapid ports.

*Default = oxf/OMDefaultReactivePort.h*

## DefaultRequiredInterfaceName

The DefaultRequiredInterfaceName property specifies the interface that must be implemented by the "out" part of a rapid port. See the Rational Rhapsody help for more information on rapid ports.

*Default = DefaultRequiredInterface*

## EnableDirectReactiveDeletion

*The EnableDirectReactiveDeletion property specifies the call to the framework that supports direct deletion of reactive instances (by using the delete operator) instead of graceful framework termination (by using the reactive destroy() method). When you use destroy(), the object waits in a zombie mode until all the events that are designated to it are removed from the active context queue, and then self-destructs. In this scheme, there is no need to traverse the queue of the active context to cancel pending events, and there is no need to make the reactive destructor guarded to ensure safe deletion. A reactive object can be either in a graceful termination or forced deletion (by using the delete operator) state: you cannot use graceful deletion on an object that allows forced deletion, and vice versa. You can set a single reactive*

object in a forced deletion state, or set the entire system (all reactive instances) in a forced deletion state (as is done for compatibility with earlier versions). Graceful termination should not be used when a reactive part (of a composite class) runs in a context of an active object that is not part of, and different from, the composite active context. If you are using a Rational Rhapsody library component as part of an application where the main is not generated by Rational Rhapsody (for example, GUI applications), the framework initializes itself in full compatibility mode on the call to `OXF::init()`. If you want to remove part or all of the compatibility features, call `OXF::initialize()` instead of `OXF::init()` (the operation takes the same arguments) and add independent activation calls that are compatible with earlier versions, prior to the `initialize()` call. Note that the `CPP_CG::Framework::UseDirectReactiveDeletion` property must be set to `True` for this property to take effect. When it is set to `True`, the code specified in the `EnableDirectReactiveDeletion` is generated in the main prior to the call to `OXF::init()`. (Default = `OXF::supportExplicitReactiveDeletion();`)

## EventBase

The `EventBase` property specifies the base class for all events, if the `EventBaseUsage` property is set to `Checked`.

*Default = `RiJEvent`*

## EventBaseUsage

The `EventBaseUsage` property specifies whether to use the event superclass specified by the `EventBase` property as the parent of all events.

*Default = `Checked`*

## EventGenerationPattern

The `EventGenerationPattern` property supplies some of the information needed to generate code for `Send Action` elements in C, C++, and Java.

For each language, code generation for this element is determined by the following properties:

- `C_CG::Framework::EventGenerationPattern` - general format
- `CG::Framework::EventToPortGenerationPattern` - used when sending even to a port

*Default = `$target$(goArr)gen(new $event)`*

Note: Rational Rhapsody does not support roundtripping for `Send Action` elements.

## EventIDType

When Rational Rhapsody generates code for an event, it creates an ID number for the event. The `EventIDType` property allows you to specify the type that should be used for this number if you do not want to use the type that Rational Rhapsody generates by default. In C and C++, the value of this property affects code generation only if the `EventIdAsDefine` property is set to `False`.



*Default = short*

## EventIncludeFiles

The EventIncludeFiles property specifies the base class for events when you use selective framework includes. If events are defined in a package, the file specified by this property is included in the package specification file to enable the use of events and timeouts in the package.

*Default = com.ibm.rational.rhapsody.oxf.RiJEvent*

## EventSetParamsStatement

The EventSetParamsStatement property specifies a template for the body of the setParams() method, provided by the Rational Rhapsody framework for Java, to set the parameters of an event. For example, for an event of type evOn(), the default template would generate the following code in the body of the setParams() method: evOn params = (evOn) event;

*Default = \$eventType params = (\$eventType) event;*

## FrameworkInitialization

The FrameworkInitialization property specifies the framework initialization code that is called by the main. The default value is as follows: OXF::initialize\$(Argc)\$ (Argv)\$ (AnimationPortNumber) \$(RemoteHost)\$ (TimerResolution)\$ (TimerMaxTimeouts) \$(TimeModel)))

## HeaderFile

The HeaderFile property specifies the framework header files to be included in objects that are within the scope of a particular configuration.

To optimize your code for size, leave the HeaderFile property blank. In this way, you can explicitly include the framework only when needed.

*Default = Blank*

## IncludeHeaderFile

The IncludeHeaderFile property specifies whether to include the framework header files specified by the CG::Framework::HeaderFile property in the project.

*Default = Checked*

## InnerReactiveClassName

The InnerReactiveClassName property specifies the name of a reactive class that serves as a bridge

between a reactive class in your model and the framework. The implementation scheme of reactive classes is different in Java than in C++. Java does not allow inheritance from the reactive framework classes because that would mean that you would not be able to inherit from an additional base class that might not be reactive. The chosen alternative is to delegate an inner class instance that inherits from `RiJStateReactive`. Delegation is the implementation of an interface that forwards relevant messages to the inner class instance.

*Default = Reactive*

## InnerReactiveInstanceName

The `InnerReactiveInstanceName` property specifies the name of a reactive instance that serves as a bridge between a reactive class in your model and the framework. The implementation scheme of reactive classes is different in Java than in C++. Java does not allow inheritance from the reactive framework classes because that would mean that you would not be able to inherit from an additional base class that might not be reactive. The chosen alternative is to delegate an inner class instance that inherits from `RiJStateReactive`. Delegation is the implementation of an interface that forwards relevant messages to the inner class instance.

*Default = reactive*

## InstrumentVtblName

*The `InstrumentVtblName` property specifies the name of the virtual function table that is associated with animation objects. Each animated object has its own virtual function table (Vtbl). This table creates your own framework, with its own virtual instrumentation functions, and connect it to Rational Rhapsody. (Default = `$ObjectName_instrumentVtbl`)*

## IsCompletedCall

*The `IsCompletedCall` property specifies the call or macro that determines whether the state reached a final state so it can be exited on a null transition. The property supports the `$State` keyword so you can use state-based calls. The keyword is resolved to the state implementation (code) name. (Default = `IS_COMPLETED($State)`)*

## IsInCall

*The `IsInCall` property specifies the query that determines whether the state is in the current active configuration. The property supports the `$State` keyword so you can use state-based calls. The keyword is resolved to the state implementation (code) name. (Default = `IS_IN($State)`)*

## MakeFileName

The `MakeFileName` property specifies a new name for the makefile. To use this property, add the following line to the .prp file:

Property MakeFileName String "MyFileName"

In this syntax, MyFileName specifies the name of the makefile.

## NullTransitionId

*The NullTransitionId property specifies the ID reserved for null transition consumption. (Default = OMEventNullId)*

## OperationGuard

*The OperationGuard property specifies the macro that guards an operation. (Default = GUARD\_OPERATION)*

## ProtectedBase

The ProtectedBase property specifies the base class for protected objects, if the ProtectedBaseUsage property is set to Checked.

*Default = Empty string*

## ProtectedBaseUsage

The ProtectedBaseUsage property specifies whether to use the class specified by the ProtectedBase property as the base class for protected objects.

*Default = Cleared*

## ProtectedClassDeclaration

*The ProtectedClassDeclaration property affects how protected classes are implemented. Beginning with Rational Rhapsody 4.0, instead of inheriting from OMProtected, the class embeds an aggregate OMProtected. The aggregate member and helper methods are defined in the macro OMDECLARE\_GUARDED (defined in omprotected.h). (Default = OMDECLARE\_GUARDED)*

## ProtectedIncludeFiles

The ProtectedIncludeFiles property specifies the base class for protected classes when you use selective framework includes. The default value for C is as follows: oxf/RiCProtected.h The default value for C++ is as follows: oxf/omprotected.h

## ProtectedInit

The ProtectedInit property specifies the declaration generated for the initializer for guarded objects. The default value for Ada is an empty string. The default value for C is as follows: \$base\_init(\$member)

## ReactiveBase

The ReactiveBase property specifies the base class for all reactive classes, if the ReactiveBaseUsage property is set to Checked.

*Default = RiJStateReactive*

## ReactiveBaseUsage

The ReactiveBaseUsage property specifies whether to use the class specified by the ReactiveBase property as the base class for all reactive objects.

*Default = Checked*

## ReactiveConsumeEventOperationName

The ReactiveConsumeEventOperationName property sets the user object virtual table for a reactive object. Follow these steps:

- Create a method with the following signature: void operation name>(RiCReactive \* const, RiCEvent\*)
- Set the operation name in the ReactiveConsumeEventOperationName property.

*Rational Rhapsody Developer for Ada ignores all the values for the properties under the Framework metaclass except for this one. (Default = empty string)*

## ReactiveCtorActiveArgDefaultValue

*The ReactiveCtorActiveArgDefaultValue property specifies the default value of the active context argument in a reactive constructor. (Default = 0)*

## ReactiveCtorActiveArgName

*The ReactiveCtorActiveArgDefaultValue property specifies the name of the active context argument in a reactive constructor. (Default = activeContext)*

## ReactiveCtorActiveArgType

*The ReactiveCtorActiveArgDefaultValue property specifies the type of the active context argument in a reactive constructor. (Default = IOxfActive\*)*

## ReactiveDestructorGuard

*The ReactiveDestructorGuard property specifies the macro that starts protection of a section of code used for destruction of a reactive instance. This prevents a “race” (between the deletion and event dispatching)*

when deleting an active instance. (Default = `START_DTOR_REACTIVE_GUARDED_SECTION`)

## ReactiveEnableAccessEventData

The `ReactiveEnableAccessEventData` property specifies the code to be used to enable access to the specific event data in a transition (typically by assigning a local variable of the appropriate type). The property supports the `$Event` keyword so you can specify the event type. (Default = `OMSETPARAMS($Event);`)

## ReactiveGuardInitialization

The `ReactiveDestructorGuard` property specifies the framework call that makes the event consumption of a specific reactive class guarded. (Default = `setToGuardReactive`)

## ReactiveHandleEventNotConsumed

The `ReactiveHandleEventNotConsumed` property registers a method to handle unconsumed events in a reactive class. Specify the method name as the value for this property. (Default = empty string)

## ReactiveHandleTONotConsumed

The `ReactiveHandleTONotConsumed` property registers a method to handle unconsumed trigger operations in a reactive class. Specify the method name as the value for this property. (Default = empty string)

## ReactiveIncludeFiles

The `ReactiveIncludeFiles` property specifies the base classes for reactive classes when you use selective framework includes. If a class is reactive and this property is defined, the file specified by the property is included in the class specification file. For reactive classes, the header files specified by the following properties are also included:

- `EventIncludeFiles` - For the event base class
- `ActiveIncludeFiles` - If the class is guarded or instrumented

The default value for C is as follows: `oxf/RiCReactive.h`

## ReactiveInit

The `ReactiveInit` property specifies the declaration for the initializer generated for reactive objects. The default pattern for C is as follows: `$base_init($member, (void*)$mePtr, $task, $VtblName);` The `$base` variable is replaced with the name of the reactive object during code generation. The string `"_init"` is appended to the object name in the name of the operation. For example, if the reactive object is named `A`, the initializer generated for `A` is named `A_init()`. The `$member` variable is replaced with the name of the reactive member (equivalent to the base class) of the object during code generation. The `$mePtr` variable is replaced with the name of the user object (the value of the `Me` property). The member and `mePtr` objects

are not equivalent if the user object is active. The `$VtblName` variable is replaced with the name of the virtual function table for an object, specified by the `ReactiveVtblName` property. The default value for Ada is an empty string. The default for C is as follows: `$base_init($member, (void*)$mePtr, $task, $VtblName);`

*Default = reactive = new Reactive(\$task);*

## ReactiveInterface

The `ReactiveInterface` property specifies the name of the interface class that forwards messages to an inner class instance of a reactive class in order to implement its reactive behavior.

*Default = RiJStateConcept*

## ReactiveSetEventHandlingGuard

*The `ReactiveSetEventHandlingGuard` property controls the code that is generated within the constructor of a reactive class. When you use this property with guarded triggered operations, it enables guarding of the event handling (in order to provide mutual exclusion between the event and TO handling). (Default = `setEventGuard(getGuard());`)*

## ReactiveSetTask

The `ReactiveSetTask` property specifies the string that tells a reactive object whether it is an active or a sequential instance. The default value for Ada is an empty string. The default value for C is as follows: `RiCReactive_setActive($member, $isActive);` The default value for C++ is as follows: `setThread($task, $isActive);`

## ReactiveVtblName

*The `ReactiveVtblName` property specifies the name of the virtual function table (Vtbl) associated with a reactive object. Each reactive object has its own Vtbl, which creates your own framework and connect it to Rational Rhapsody. (Default = `$ObjectName_reactiveVtbl`)*

## SetManagedTimeoutCanceling

*The `SetManagedTimeoutCanceling` property is a property for compatibility with earlier versions that specifies whether the framework uses the pre-Rational Rhapsody 6.0 scheme of timeout creation and cancellation (where `OMTimerManager` is responsible for cancellation of timeouts) or the Rational Rhapsody 6.0 scheme. In Rational Rhapsody 6.0, the framework moves the responsibility for a timeout cancellation from the timer manager to the timeout client (the reactive object). This change reduces the timer manager responsibilities and the overhead in timeout management (thus improving timeout scheduling performance). The change also includes changes in the generated code (the user reactive objects hold pointers to the waiting timeouts in order to enable canceling). If you are using a Rational Rhapsody library component as part of an application where the main is not generated by Rational Rhapsody (for example, GUI applications), the framework initializes itself in full compatibility mode on the call to `OXF::init()`. If you want to remove part or all of the compatibility features, call*

*OXF::initialize()* instead of *OXF::init()* (the operation takes the same arguments) and add independent activation calls that are compatible with previous versions, prior to the *initialize()* call. (Default = *OXF::setManagedTimeoutCanceling(true);*)

## SetRhp5CompatibilityAPI

*The SetRhp5CompatibilityAPI property specifies the call that configures models created before Rational Rhapsody 6.0 so they use the 5. x version of the framework instead of the new one. See UseRhp5CompatibilityAPI for more information on Version 5. x compatibility mode. (Default = OXF::setRhp5CompatibleAPI(true);)*

## StaticMemoryIncludeFiles

*The StaticMemoryIncludeFiles property specifies the files to be included in the package specification file if static memory management is enabled and you are using selective framework includes. (Default = oxf/MemAlloc.h)*

## StaticMemoryPoolDeclaration

The StaticMemoryPoolDeclaration property specifies the declaration of the memory pool for timeouts. The default value is as follows:

```
DECLARE_MEMORY_ALLOCATOR($Class, $BaseNumberOfInstances)
```

## StaticMemoryPoolImplementation

The StaticMemoryPoolImplementation property specifies the generated code in the implementation file for a memory pool implementation (see the BaseNumberOfInstances property). The default value is as follows:

```
IMPLEMENT_MEMORY_ALLOCATOR($Class, $BaseNumberOfInstances,  
$AdditionalNumberOfInstances, $ProtectStaticMemoryPool)
```

## TestEventTypeCall

*The TestEventTypeCall property specifies the test used in event consumption code to check if the currently consumed event is of a given type. (Default = IS\_EVENT\_TYPE\_OF(\$Id))*

## TimeoutId

*The TimeoutId property specifies the ID reserved for timeout events. (Default = OMTimeoutEventId)*

## TimerMaxTimeouts

The `TimerMaxTimeouts` property specifies the maximum number of timeouts allowed simultaneously in the system, if the `TimerMaxTimeouts` property for the configuration is not overridden. In the framework, the default number of timers is 100.

*Default = Empty string*

## TimerResolution

The `TimerResolution` property specifies the length of time that must pass until the timer should check for matured timeouts. In the framework, the default number of timers is 100.

*Default = Empty string*

## UseDirectReactiveDeletion

*The `UseDirectReactiveDeletion` property determines whether direct deletion of reactive instances (by using the delete operator) is used instead of graceful framework termination (by using the reactive `destroy()` method). When this property is set to `True`, the code specified in the `EnableDirectReactiveDeletion` is generated in the main prior to the call to `OXF::init()`. See `EnableDirectReactiveDeletion` and the upgrade history on the support site for more information on this functionality. (Default = `False`)*

## UseManagedTimeoutCanceling

*The `UseManagedTimeoutCanceling` property specifies whether the framework uses the pre-Rational Rhapsody 6.0 scheme of timeout creation and cancellation (so `OMTimerManager` is responsible for cancellation of timeouts). In Rational Rhapsody 6.0, the framework moves the responsibility for a timeout cancellation from the timer manager to the timeout client (the reactive object). This change reduces the timer manager responsibilities and the overhead in timeout management (thus improving timeout scheduling performance). The change also includes changes in the generated code (the user reactive objects hold pointers to the waiting timeouts in order to enable canceling). When loading a pre-6.0 model, Rational Rhapsody sets the project `CPP_CG::Framework::UseManagedTimeoutCanceling` to `True` to set the system-compatibility mode. See the upgrade history on the support site for more information. (Default = `False`)*

## UseRhp5CompatibilityAPI

*The `UseRhp5CompatibilityAPI` property specifies whether to use the virtual functions of the core implementation classes that existed in the pre-Rational Rhapsody 6.0 framework. The Rational Rhapsody 6.0 framework introduces a set of interfaces for the core behavioral framework. The interfaces define a concise API for the framework and enable you to replace the actual implementation of these interfaces while maintaining the framework behavior. As a result of the interfaces' introduction, the framework behavioral classes (`OMReactive`, `OMThread`, and `OMEvent`) use a new set of virtual operations to implement the interfaces and provide the behavioral infrastructure. To support existing customizations of these classes (made by inheriting and overriding the virtual operations), the framework can work in a mode where the pre-6.0 API virtual operations are called. When loading a pre-6.0 model, Rational Rhapsody sets the project property `CPP_CG::Framework::UseRhp5CompatibilityAPI` to `True` to set the system-compatibility mode. If this is set to `True`, the pre-6.0 API is called by the framework instead of the interface-based API. Without this flag, user customizations compile but are not called. See the upgrade*



*history on the support site for more information on the Version 5. x compatibility mode. (Default = False)*

## Generalization

The Generalization metaclass contains a property used to support generalization. See the Rational Rhapsody help for more information on generalization.

## Animate

The Animate property specifies whether animation code is generated for an element. You can specify your own animation function by using the CG::Attribute::AnimSerializeOperation property. The semantics of the Animate property is always in favor of the owner settings:

- If a package Animate property is set to False, all the classes owned by the package are not animated, regardless of the class Animate settings.
- If a class Animate property is set to False, all the elements in the class (attributes, operations, relations, and so on) are not animated.
- If an operation Animate property is set to False, all the arguments are not animated.
- If the AnimateArguments property is set to False, all the arguments are not animated, regardless of the specific argument Animate property settings.

*(Default = Checked)*

## JDK

The JDK metaclass contains properties that manipulate the operating system environment.

## AdaptorSearchPath

The AdaptorSearchPath property specifies the path to the operating system configuration file. This path is added to the generated makefile search path. This property reflects the change in Version 4.1 where the RTOS-specific code was removed from the framework code and placed in separate files, and a new adapter builder was created. This new scheme makes it easier to add a custom adapter because you do not need to modify the framework files. To upgrade a custom adapter to the new scheme, you must do the following:

- Create the relevant operating system configuration file.
- Add the file directory to the search path in the framework makefiles.
- Add the AdaptorSearchPath property to the adapter environment properties, with the value set to the path to the operating system configuration file.

*Default =*

## AdditionalReservedWords

The AdditionalReservedWords property is a string that specifies additional reserved keywords that Rational Rhapsody does not allow you to use. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody. The property value is checked at run time when you name or rename an element, based on the active configuration environment setting. Note that this property affects the algorithm only when the active configuration is of the selected environment.

*Default = Empty string*

## BriefErrorMessages

*The BriefErrorMessages property determines whether a /brief option is generated on SPARK Examiner calls. (Default = Checked)*

## BSP

*The BSP property specifies the board support package (BSP) for the system. If you need to change the value of the CPU, you can simply reset the value in this property instead of changing the value in the MakeFileContent property. (Default = "PENTIUM")*

## BSP\_Libraries

The BSP\_Libraries property specifies the default BSP libraries to link to. The default value is as follows:

```
"%RAVENROOT%/bsp/raven/standard_model" "%RAVENROOT%/bsp/system/simulator"  
"%RAVENROOT%/lib/extensions"
```

## BuildCommandSet

The BuildCommandSet property generates a set of commands in the makefile to build a debug or release version of the configuration. To change this property, use the Configuration window in the browser - do not change it by using the Properties window or by modifying the site.prp file. Note that this property also affects the names of the framework libraries used in the link. The possible values are as follows:

- Debug - Generate the debug command that is set in the makefile.
- DebugNoExp - Generate the debug command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).
- Release - Generate the release command that is set in the makefile.
- ReleaseNoExp - Generate the release command that is set in the makefile without the exceptions flag (:cx\_option=exceptions).

*Default = Debug*

## BuildInIDE

The BuildInIDE property is a Boolean value that allows you to specify the program that should perform the build - Rational Rhapsody or the IDE with which it is being used. If the value of the property is set to True, then Rational Rhapsody calls the IDE build command when its own build command is started.

This property corresponds to the "Build configuration in IDE" option on the IDE tab of the features window for configurations.

*Default = True*

## CodeTestSettings

*The CodeTestSettings property specifies the compiler command settings. This property supports integration with Applied Microsystems Corporation CodeTest. (Default = CXX=\$AMC\_HOME)\bin\ctcxx)*

## COM

*The COM property specifies whether the current component is a COM component. By default, this property is set to True for all COM components (stereotypes COM DLL, COM EXE, and COM TLB). If you set this property in the generated makefile for the component, the linker option /SUBSYSTEM is set to :windows. (Default = False)*

## CompileCommand

*The CompileCommand property is a string that specifies a different compile command. (Default = empty string)*

## CompileSwitches

The CompileSwitches property specifies the compiler switches. This property replaces the CPPCompileSwitches property.

*Default = Empty MultiLine*

## ConvertHostToIP

*The ConvertHostToIP property specifies whether to convert the host name to an IP number. This is necessary because pSOSystem does not include a name service. Default = Checked*

## DEFExtension

*The DEFExtension property is a string that specifies the extension for DLL definition files. In general, this*

*is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody. (Default = .def)*

## DllExtension

*The DllExtension property is a string that specifies the extension for DLL files. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody. (Default = .dll)*

## DuplicateLibsListInMakeFile

*The DuplicateLibsListInMakeFile property is a Boolean value that specifies whether Rational Rhapsody should duplicate the libraries list in the generated makefile link command. This property supports linkers that are sensitive to library order in the link command. (Default = Checked)*

## EnableDebugIntegrationWithIDE

When you use Rational Rhapsody in conjunction an IDE such as Eclipse, you can use the EnableDebugIntegrationWithIDE property to specify whether or not the IDE debugger is to be used in conjunction with the Rational Rhapsody animation feature.

If the value of the property is set to Checked, the IDE debugger is used.

## EntryPoint

The EntryPoint property specifies the name of the main program for a given environment.

*Default = main*

See also the definition of the EntryPointDeclarationModifier property for more information.

## EntryPointDeclarationModifier

The EntryPointDeclarationModifier property specifies a modifier for the entry point declaration. This property allows generation of the main() function in the specified syntax. To modify the main() signature implemented in the OSE adapter, do the following:

- Add the EntryPointDeclarationModifier property to your environment properties and set it to the main return value and name. For example: "int main"
- Set the EntryPoint property to the main arguments. For example: "int a, long b, char\*\*"
- Generate the code.

You get the following main() declaration:

```
int main(int a, long b, char** c) { ... }
```

(Default = OS\_PROCESS)

## EnvironmentVarName

*The EnvironmentVarName property specifies the name of the global variable that you must define in order to use the Embedded C++ compiler. It is used by the MultiMakefileGenerator. The value replaces the \$EnvironmentVarName value> keyword inside the value for the BLDAditionalOptions property. (Default = INTEGRITY\_ROOT)*

## ErrorMessageTokensFormat

The ErrorMessageTokensFormat, working with the ParseErrorMessage property, specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler. ErrorMessageTokensFormat defines the number and location of tokens within the regular expression defined by the ParseErrorMessage property. ErrorMessageTokens has three parameters, each with an integer value:

- TotalNumberOfTokens - The number of tokens in the regular expression
- FileTokenPosition - The position of the file name token in the expression
- LineTokenPosition - The position of the line number token in the expression

*Default = ToTalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2*

## ESTLCompliance

The ESTLCompliance property is a Boolean value that determines whether you are using the Embedded C++ (ESTL) environment and conform to its requirements. In instrumentation mode, the Rational Rhapsody code generator usually creates an OMAAnimatedUser Class> friend class for each user-defined class. This class inherits from AOMInstance, if its User Class> does not inherit from another class in the model. This inheritance is virtual and is needed for multiple inheritance support. Because ESTL does not support multiple inheritance (as far as virtual inheritance), the Rational Rhapsody Developer for C++ code generator does not create “virtual” inheritance if ESTLCompliance is set to True. To support ESTL compliance, Rational Rhapsody includes a new check to recognize the following elements of ESTL-noncompliance:

- Multiple inheritance, caused by the user model (several superclasses)
- Multiple inheritance, caused by Rational Rhapsody (an active reactive class is generated with two base classes: OMReactive and OMThread)
- Multiple inheritance, caused by a combination of the following factors:
  - An active class containing a superclass
  - A reactive class containing a superclass
  - Virtual inheritance, declared by the user in the features of the superclass

*In these cases, Rational Rhapsody displays the following warning message for each problematic class: "ESTL does not support multiple/virtual inheritance" Note that this check runs only when the ESTLCompliance property is set to True. (Default = Checked)*

## ExeExtension

The ExeExtension property specifies the extension that is appended to compiled executable components for a given environment.

*Default = .class*

## FileDependencies

The FileDependencies property specifies which framework specification files and implementation files should be included in model elements. The file inclusions are generated in the makefile. The default value for GNAT and OsePPCDiab/OseSfk is an empty string. The file dependency string for most of the C and C++ environments is as follows: \$OMSpecIncludeInElements \$OMImpIncludeInElements

## FrameworkAnimJars

This property specifies a list of JAR files that make up the animated framework. These JAR files are included in the CLASSPATH when executing an animated application.

## FrameworkNoneJars

This property specifies a list of JAR files that make up the non-animated framework. These JAR files are included in the CLASSPATH when executing a non-animated application.

## GeneratedAllDependencyRule

*The GeneratedAllDependencyRule property specifies whether to automatically generate the "all:" rule as part of the expansion of the \$OMContextMacros keyword in the makefile. If this is False, you can define the makefile macros manually. (Default = False)*

## GetConnectedRuntimeLibraries

The GetConnectedRuntimeLibraries property specifies the list of libraries that need to be linked with Web-enabled projects (when the Web Instrumentation check box is enabled). During code generation, these libraries are added to the generated makefile. Note that if you select Release Build Set (in the Environment Settings group on the Settings page), these libraries are automatically added with the R postfix (the Rational Rhapsody convention for framework libraries).

*Default = \$RhapJarsDir\webComponents.jar*

## HasIDEInterface

The HasIDEInterface property is a Boolean value that specifies whether IDE support is enabled. If IDE support is enabled (True), the IDEInterfaceDLL property points to an IDE adapter that provides

connection to the IDE. If the property is set to False, IDE support is disabled and IDE services are not attempted. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody. The default value for QNXNeutrinoCW is False; for the other environments, the default value is True.

## ImpExtension

The ImpExtension property specifies the extension that Rational Rhapsody appends to generated implementation files for a given language and environment. The default values for Java is None

## InvokeExecutable

The InvokeExecutable property specifies the command used to run an executable file.

*Default = "\$OMROOT\etc\jdkrun.bat" "\$makefile" Main\$ComponentName*

## InvokeMake

The InvokeMake property specifies the command that is started for Build. The command syntax and batch file are target-dependent. The command syntax is as follows:

"rhapsody\_dir>\share/etc/Executer.exe""dir; dir | more"

The InvokeMake property might include the value of any other property in the relevant environment. To use this feature, simply include the name of the property preceded by \$. As shown in the example below (from the VxWorks RTP environment), the value of the InvokeMake property includes the value of the BSP property.

"\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\vx6make.bat\" \$makefile \$maketarget 6.2 \$BSP gnu"

*Default = "\$OMROOT/etc/Executer.exe" "\"\$OMROOT\etc\jdkmake.bat\" \$makefile \$maketarget"*

## InvokeMakeGenerator

The InvokeMakeGenerator specifies the path to the executable program for an external makefile generator. This external generator is started each time you request a makefile generation. If the specified path is incorrect, Rational Rhapsody generates an error message. If you are using a full-featured external code generator, this property setting is ignored. The default values are as follows:

Java - None

## IsFileNameShort

The IsFileNameShort property specifies whether to truncate generated file names to 8.3 format. If this is Checked:

- The file name is not truncated.
- If the FileName property is not blank, its value overrides any automatic file name synthesis.
- If the file name is longer than eight characters, the Checker reports this prior to code generation.

*Default = Cleared*

## LibExtension

The LibExtension property specifies the extension that is appended to compiled library components for a given environment.

*Default = Empty string*

## MainIncludes

*The MainIncludes property is a string that specifies the files that need to be included in the main program generated for an application. (Default = ose.h)*

## MakeExtension

The MakeExtension property specifies the extension that Rational Rhapsody appends to makefiles.

*Default = .bat*

## MakeFileContent

The MakeFileContent property specifies how the makefile is generated for a configuration. The makefile can be of any length. The InvokeMake property references this makefile. A makefile has the following sections:

- Target type
- Compilation flags
- Commands definitions
- Generated macros
- Predefined macros
- Generated dependencies
- Linking instructions

The following sections describe the contents of the makefile in detail.

**Target Type** The target type section of the makefile contains the macros needed to build either a Debug or Release version of a configuration.

```
echo off set RHAP_JARS_DIR=$OMRoot\LangJava\lib set
SOURCEPATH=$ConfigSources$ComponentSources%SOURCEPATH% set
```



```

CLASSPATH=$ConfigClasspath$ComponentClasspath%CLASSPATH%;;%RHAP_JARS_DIR%\oxf.jar;%RHAP_JARS_
set PATH=$ConfigPath$ComponentPath%RHAP_JARS_DIR%;%PATH%; set
INSTRUMENTATION=$INSTRUMENTATION set BUILDSET=$BuildSet if
%INSTRUMENTATION%==Animation goto anim :noanim set
CLASSPATH=%CLASSPATH%;%RHAP_JARS_DIR%\oxfInstMock.jar goto setEnv_end :anim set
CLASSPATH=%CLASSPATH%;%RHAP_JARS_DIR%\oxfInst.jar :setEnv_end if "%1" == "" goto
compile if "%1" == "build" goto compile if "%1" == "clean" goto clean if "%1" == "rebuild" goto clean if
"%1" == "run" goto run :clean echo cleaning class files $ClassClean if "%1" == "clean" goto end :compile
if %BUILDSET%==Debug goto compile_debug echo compiling JAVA source files javac
$ConfigCompilerSwitches @$SourceListFile goto end :compile_debug echo compiling JAVA source files
javac -g $ConfigCompilerSwitches @$SourceListFile goto end :run java %2 :end

```

Java Users To generate Java JAR files, run the jar command from the makefile, use the MakeFileContent property. You can specify the manifest file as an external file with a text element in it. You can add additional files to the model for completeness. There is no specialized support for RMI in Rational Rhapsody. Call the JDK and run the relevant tools manually, or by way of the generated makefile (change the MakeFileContent property).

## NullValue

The NullValue property specifies an alternative expression for NULL in the generated code.

*Default = NULL*

## ObjCleanCommand

The ObjCleanCommand property specifies the environment-specific command used to clean the object files generated by a previous build.

*Default = if exist \$OMFileObjPath del \$OMFileObjPath*

## ObjectName

*The ObjectName property specifies an alternative name for the compiled object file in the generated makefile. (Default = empty string)*

## ObjExtension

The ObjExtension property specifies the extension appended to compiled object files for a given environment.

*Default = .class*

## OMCPU

*The OMCPU property is resolved in the MakeFileContent property as the CPU type. The QNXNeutrinoCW environment uses the custom keywords feature to enable you to select the CPU without*

*modifying the makefile template. (Default = x86)*

## **OMCPU\_SUFFIX**

*The OMCPU\_SUFFIX property is resolved in the MakeFileContent property as the CPU extension (which is required for PPC targets). The QNXNeutrinoCW environment uses the custom keywords feature to enable you to select the CPU without modifying the makefile template. (Default = (\$NO\_CPU\_EXT))*

## **OpenHTMLReports**

*The OpenHTMLReports property specifies whether to open the HTML reports when the examination is complete. (Default = Checked)*

## **OSFileSystemCaseSensitive**

The OSFileSystemCaseSensitive property specifies whether the OS file system for a given environment is case sensitive.

Default = Cleared

## **ParseErrorDescript**

The ParseErrorDescript property is used to define a regular expression that represents the format of build error messages. The property is used to extract the "description" part of the message so that it can be displayed in the Description column on the Build tab of the Output window.

*Default = ([^:]+):([0-9]+):(.\*)*

## **ParseErrorMessage**

The ParseErrorMessage property defines a regular expression that matches the format of error messages. The regular expression can contain two or more tokens. For example: (lineNumber), (fileName), and (Error|Fatal|Warning) Along with the ErrorMessageTokensFormat property, ParseErrorMessage specifies the expected format of error messages for a given environment. These two properties retrieve the file name and line number of errors reported by the compiler.

*Default = ([^:]+):([0-9]+):*

## **ParseSeverityError**

The ParseSeverityError property is used to define a regular expression that represents the format of compilation messages with severity "error." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = ([^:]+):([0-9]+):*

## ParseSeverityWarning

The ParseSeverityWarning property is used to define a regular expression that represents the format of compilation messages with severity "warning." This property is used to determine the type of icon that should be displayed alongside the message on the Build tab of the Output window.

*Default = (.\*) (Note:/warning:) (.\*)*

## PathDelimiter

The PathDelimiter property specifies an alternative path separator for code generation.

*Default = /*

## ProcessToKillAtStopExec

The ProcessToKillAtStopExec property stops the running process of the Java application when you select Code > Stop Execution in the Rational Rhapsody GUI.

*Default = Java*

## QuoteOMROOT

The QuoteOMROOT property specifies whether to enclose the value of the OMROOT path variable in double quotation marks in the generated makefile.

*Default = Checked*

## RCExtension

*The RCExtension property is a string that specifies the extension for resource files. In general, this is an environment property that can be contained in any of the environment metaclasses supported by Rational Rhapsody. (Default = .rc)*

## RemoteHost

The RemoteHost property specifies the name of the host machine when you run an application remotely. In such configurations, the host is the machine running the product, whereas the target is the machine running the application. To run remotely, the UseRemoteHost property must be set to True. If UseRemoteHost is True and RemoteHost is blank, the current host name is used for the remote host. You can use this as a workaround if you have problems running animated applications on Windows 95. The RemoteHost property can be left blank if both the application and Rational Rhapsody are running on the same machine.

*Default = Empty string*

## RhapJarsDir

This property indicates the folder containing the compiled JAR files of the Java framework.

## SpecExtension

The SpecExtension property determines the extension that Rational Rhapsody appends to generated specification (header) files for a given language and environment.

*Default = .java*

## SpecFilesInDependencyRules

The SpecFilesInDependencyRules property specifies whether to include specification files in makefile dependency rules. The OSE makefile does not support specification files in the Dependency line. Therefore, the default for OSE is False. When this property is False, no .h files are added to the Dependency line of the makefile. The default value for GNAT is True; for OSE, the default value is False.

## SubSystem

The SubSystem property is a string that defines the type of the program for the Microsoft linker. The possible values are as follows:

- **CONSOLE** - Used for a Win32 character-mode application
- **WINDOWS** - Used for an application that does not require a console
- **NATIVE** - Applies device drivers for Windows NT
- **POSIX** - Creates an application that runs with the POSIX subsystem in Windows NT

*(Default = /SUBSYSTEM:console)*

## TargetConfigurationFileName

*The TargetConfigurationFileName property specifies the name of the target configuration file to be passed as an argument to the SPARK Examiner. (Default = empty string)*

## UnixLineTerminationStyle

The UnixLineTerminationStyle property specifies whether generated files use the UNIX end-of-line style. If this property is set to Cleared, the end-of-line style depends on the host type (for example, DOS style on Windows machines, and UNIX style on Solaris machines).

*Default = Cleared*

## UnixPathNameForOMROOT

The UnixPathNameForOMROOT property specifies whether the makefile must include UNIX-style path names. The pRISM compilers do not tolerate DOS-style path conventions. If you do not set this property correctly, there might be many compilation problems.

*Default = Cleared*

## UpdateBuildSettingsInIDE

The UpdateBuildSettingsInIDE property is used when you use Rational Rhapsody in conjunction with an IDE such as Eclipse. If the value of the property is set to True, then Rational Rhapsody updates the build settings in the IDE after any changes are made to the build settings (such as make-related properties). The update is performed after code generation.

*Default = False*

## UseActorsCode

*The UseActorsCode property specifies whether code is generated for actors. The value of the property should be synchronized with the configuration Generate Code For Actors checkmark (located in the configuration Initialization tab). (Default = False)*

## UseNewBuildOutputWindow

The UseNewBuildOutputWindow property determines which tab is brought to the front of the Output window after the completion of a build action. If set to True, the Build tab is shown. Otherwise, the Log tab is shown.

This property can be set individually for different environments.

If you would like to have the Log tab shown for all environments, you can set the value of the CG::General::ShowLogViewAfterBuild property to True.

*Default = Checked*

## UseNonZeroStdInputHandle

The UseNonZeroStdInputHandle property is a Boolean value that specifies whether to use a non-zero standard input handle. For INTEGRITY, OBJECTADA, RAVEN\_PPC, and SPARK the default value is False; for the other environments, the default value is True.

## UseRemoteHost

The UseRemoteHost property specifies whether you intend to run an application remotely over a network

by default in a given environment.

*Default = Cleared*

## ModelElement

Contains properties related to multiple types of model elements.

### RefactorRenameRegularExpression

When you use the Refactor:Rename feature, Rhapsody searches the user-defined code in the model for all occurrences of the name of the specified element, including places where the element name appears as part of the name of elements whose name is derived from the element that is being renamed.

To locate these elements, Rhapsody uses a set of regular expressions.

The regular expressions used for this search are defined by the property RefactorRenameRegularExpression.

Different values are used for this property, depending on what type of element you are renaming. For example, you will see that this property has a different value under Class than it does under Attribute. For some types of elements, Rhapsody uses a more generic expression, taken from the value of the property under ModelElement.

In general, the value of this property is a regular expression that consists of a number of "Or" components. For the renaming to work properly, each such component must contain three groupings, with the middle grouping using the variable \$keyword, which represents the name of the element that is being renamed. In some cases, the expression takes this information from the value of another property rather than using the \$keyword variable.

*Default = ([^A-Za-z0-9\_]/^)([extract\_itex]keyword)([/extract\_itex][^A-Za-z0-9\_]/\$)*

## Operation

The Operation metaclass contains properties that control operations.

### ActivityReferenceToAttributes

*The ActivityReferenceToAttributes property specifies whether Rational Rhapsody should generate references in the functor object, thereby giving you direct access to the attributes of the class that owns the modeled operation (without the need for this\_). See the section on activity diagrams in the Rational Rhapsody help for information about modeled operations and functor classes. (Default = Checked)*

## AnimAllowInvocation

The AnimAllowInvocation property specifies whether primitive and triggered operations can be called during instrumentation. If an operation is called during animation, its return value is displayed in the output window; if it is traced, the return value is displayed in the console. The possible values are as follows:

- All - Enable all operation calls, regardless of visibility.
- None - Do not enable operation calls.
- Public - Enable calls to public operations only.
- Protected - Enable calls to protected operations only.

(Default = None)

## DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- \$Name - the element name
- \$FullName - the full path of the element (P1::P2::C.a)
- \$Description - the text provided on the Description tab of the Features window for the element
- \$Requirements - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the \$Requirements keyword in the DescriptionTemplate property to get such comments generated for specific model elements.
- \$Login - the login name of the user who generated the file
- \$CodeGeneratedDate - the date on which the code was generated
- \$CodeGeneratedTime - the time at which the code was generated
- \$RhapsodyVersion - the version of Rhapsody that was used to generate the file
- For operations: \$Signature - the operation signature, \$Arguments - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: \$Type - the type of the attribute

- For arguments: \$Type - the argument type, \$Direction - the argument direction (In, Out, or InOut)
- For events: \$Arguments - the text provided on the Description tab of the Features window for the event arguments
- For association ends: \$Target - the other end of the association
- For requirements: \$ID - the ID specified for the requirement, \$Specification - the specification text for the requirement
- The values of tags can be included by using the syntax \$tag\_name. For example, if you use a tag called author for your classes, you can use the keyword \$author in the value of DescriptionTemplate in order to have the value of the author tag included in the element description.
- The values of properties can be included by using the syntax \$property\_name. For example, you can use the keyword \$Animate in the value of DescriptionTemplate in order to have the value of the Animate property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as \$Name)
- Tag keywords
- Property keywords

If the value of a keyword is a MultiLine value, then each new line (except the first one) starts with the value of the <lang>\_CG::Configuration::DescriptionBeginLine property, and each line ends with the value of the <lang>\_CG::Configuration::DescriptionEndLine property.

*Default = [[ \* \$Description]] [[ \* \$Arguments]] [[ \* @return \$Return]] [[ \* @throws \$Throws]] [[ \* @see \$See]] [[ \* @since \$Since]] [[ \* @deprecated \$Deprecated]]*

## EntryCondition

*The EntryCondition property specifies the task guard. (Default = empty string)*

## GenerateImplementation

*The GenerateImplementation property specifies whether to generate the body for the operation. To generate Import pragmas in Rational Rhapsody Developer for Ada, set this property to False and add the "pragma..." declaration in the Ada\_CG::Operation::SpecificationEpilog property. (Default = Checked)*

## ImplementActivityDiagram

*The ImplementActivityDiagram property enables or disables code generation for activity diagrams. (Default = False)*

## ImplementationEpilog

The ImplementationEpilog property adds any code that you want to be added as verbatim text (to be



ignored by Rational Rhapsody) to the end of the definition of a model element. For example, you could wrap a section of code with an `#ifdef-#endif` pair, add compiler-specific keywords, or add a `#pragma` statement. For example, to specify that an operation is available only when the code is compiled with `_DEBUG`, set the following properties for the operation:

- Set SpecificationProlog to `#ifdef _DEBUG cr.`
- Set SpecificationEpilog to `#endif.`
- Set ImplementationProlog to `#ifdef _DEBUG cr.`
- Set ImplementationEpilog to `#endif.`

The following table lists whether leading and trailing linefeeds are generated.

*(Default = Empty MultiLine)*

## ImplementationName

The ImplementationName property gives an operation one model name and generate it with another name. It is introduced as a workaround that generates const and non-const operations with the same name. For example:

- Create a class A.
- Add a non-const operation `f()`.
- Add a const operation `f_const()`.
- Set the `CPP_CG::Operation::ImplementationName` property for `f_const()` to “f.”
- Generate the code.

*The resulting code is as follows: class A { ... void f(); /\* the non const f \*/ ... void f() const; /\* actually f\_const() \*/ ... }; The creation of two operations with the same signature, differing only in whether it is a const, is a common practice in C++, especially for STL users. (Default = empty string)*

## ImplementationProlog

The ImplementationProlog property adds any code that you want to be added as verbatim text (to be ignored by Rational Rhapsody) to the beginning of the definition of a model element. For example, you could wrap a section of code with an `#ifdef-#endif` pair, add compiler-specific keywords, or add a `#pragma` statement. For example, to specify that an operation is available only when the code is compiled with `_DEBUG`, set the following properties for the operation:

- Set SpecificationProlog to `#ifdef _DEBUG cr.`
- Set SpecificationEpilog to `#endif.`
- Set ImplementationProlog to `#ifdef _DEBUG cr.`
- Set ImplementationEpilog to `#endif.`

The following table lists whether leading and trailing linefeeds are generated.

*(Default = Empty MultiLine)*

## Inline

The Inline property specifies how inline operations are generated. Which operations are affected by the Inline property depends on the metaclass:

- Attribute - Applies only to operations that handle attributes (such as accessors and mutators)
- Operation - Applies to all operations
- Relation - Applies only to operations that handle relations

Inlining in Rational Rhapsody Developer for Java Because inlining has no meaning in Java, the Inline property is set to none.

*Default = none*

## IsEntry

*The IsEntry property indicates whether the operation is a task entry or a regular operation in AdaTask and AdaTaskType classes. (Default = False)*

## IsExplicit

*The IsExplicit property is a Boolean value that allows you to specify that a constructor is an explicit constructor. (Default = False)*

## IsNative

The IsNative property specifies whether the Java modifier "native" should be added to an operation in the source file. The body of such operations, if specified, is ignored by the code generator.

*Default = Cleared*

## JavaAnnotation

The JavaAnnotation property is used by the Rational Rhapsody code generator to insert Java annotations into generated code.

This property is used primarily for regenerating code that was reverse engineered. When you reverse engineer code that contains Java annotations, the value of the `JAVA_ReverseEngineering::ImplementationTrait::ImportJavaAnnotation` property determines how Rational Rhapsody handles the annotation code. If the value of this property is set to Verbatim, then Rational Rhapsody does not import annotations as model elements. Rather, the annotation code is stored as the value of the JavaAnnotation property. When code is later regenerated, it includes the code that was stored in this property.

*Default = Blank*

## Kind

The Kind property specifies the kind of operation that should be generated for an element. The kind of operations that can be generated is language-dependent (for example, virtual and abstract exist only in C++ and Java). In Java, Kind can be defined only for attributes and operations, but not for relations. This property affects class operations, in addition to accessors and mutators for relations and attributes. The possible values are as follows:

- common - Class operations and accessor/mutator are non-virtual.
- virtual - Class operations and accessor/mutator are virtual. This type is valid for C++ and Java only.
- abstract - Class operations and accessor/mutator are pure virtual. This type is valid for C++ and Java only.

*Default = common*

## LocalVariablesDeclaration

*The LocalVariablesDeclaration property specifies variables that you want to appear in the declaration of the entrypoint or operation. (Default = empty string)*

## MarkPrologEpilogInAnnotations

The MarkPrologEpilogInAnnotations property specifies whether to generate ignore annotations for the Specification, Implementation Prolog, and Epilog properties so they are ignored during roundtrip. When you insert code element declarations (variables, types, functions, and so on) in the Specification, Implementation Prolog, and Epilog properties, after a full roundtrip those elements are added to the model and are duplicated on the next code generation. When you use the MarkPrologEpilogInAnnotations property, you can have Rational Rhapsody automatically ignore the information specified in the Specification, Implementation Prolog, and Epilog properties instead of adding the ignore annotations manually. The possible values for the MarkPrologEpilogInAnnotations property are as follows:

- None - Rational Rhapsody does not generate any annotations. Any models created before Version 4.1 automatically have this property setting.
- Ignore - Rational Rhapsody generates the `/// ignore annotation before the code specified in the Specification, Implementation Prolog, and Epilog properties, and generates the /// annotation after the code specified in those properties.`
- Auto - If the code in the Specification, Implementation Prolog, and Epilog properties is one line (it does not contain any newline characters (`\n`)), no annotations are generated (the same behavior has the None setting). If there is more than one line, Rational Rhapsody generates the `/// ignore annotation before the code specified in the Specification, Implementation Prolog, and Epilog properties, and generates the /// annotation after the code specified in those properties (the same behavior as the Ignore setting).`

*During roundtrip, any ignore annotations in the comments of the element are not included in its description. Because the Specification, Implementation Prolog, and Epilog properties are generated between the annotation and its declaration for the element, you cannot rename those elements on roundtrip. If you change the name of an element, it is removed from the model and added with the new name. Some model information (for example, property settings) might be lost. (Default = Auto)*

## Me

*The Me property specifies the name of the first argument to operations generated in C. (Default = me)*

## MeDeclType

The MeDeclType property is a string that specifies the type of the first argument to operations generated in C, as a pointer to an object or object type. The default value is as follows: \$ObjectName\* const The variable \$ObjectName is replaced with the name of the object or object type.

## PrivateQualifier

*The PrivateQualifier property specifies the qualifier that is printed at the beginning of a private operation declaration or definition. You can set this property to an empty string to prevent the generation of the static qualifier in the private function declaration or definition. (Default = static)*

## ProtectedName

The ProtectedName property specifies the pattern used to generate names of private operations in C. The default value is as follows: \$opName The \$opName variable specifies the name of the operation. For example, the generated name of a private operation go() of an object A is generated as: go()

## PublicName

The PublicName property specifies the pattern used to generate names of public operations in C. The default value is as follows: \$ObjectName\_\$opName The \$ObjectName variable specifies the name of the object; the \$opName variable specifies the name of the operation. For example, the generated name of a public operation go() of an object A is generated as: A\_go()

## PublicQualifier

*The PublicQualifier property specifies the qualifier that is printed at the beginning of a public operation declaration or definition. Note that the Static checkmark in the operation window UI is disabled in Rational Rhapsody Developer for C because the checkmark is associated with class-wide semantics that are not supported by Rational Rhapsody Developer for C. When loading models from previous versions, the Static check box is cleared; if the operation is public, the C\_CG::Operation::PublicQualifier property value is set to Static in order to generate the same code. (Default = empty string)*

## Renames

The Renames property enables one element to rename another element of the same type. You can also rename an element by using a renames dependency. In the case of a conflict, the dependency has precedence. Note the following:

- For attributes, this property works only for static attributes in a class or for attributes in a package.

- For operations, this property contains the name of the operation this operation in renaming. The signatures of the two operations must match.

*(Default = empty string)*

## RenamesKind

*The RenamesKind property specifies whether the renaming of the operation designated in the Ada\_CG::Operation::Renames property is “as specification” or “as body.” (Default = Specification)*

## ReturnTypeByAccess

*The ReturnTypeByAccess property determines whether the return type is generated as an access type or a regular type. Note that this property is applicable only to classes for which an access type is generated. (Default = False)*

## SpecificationEpilog

The SpecificationEpilog property allows you to add code to the end of the declaration of a model element.

*Default = Blank*

## SpecificationProlog

The SpecificationProlog property allows you to add code to the beginning of the declaration of a model element.

For example, you could add the @Deprecated annotation for an element by entering @Deprecated and a new line as the value of this property.

*Default = Blank*

## TaskDefaultScheme

The TaskDefaultScheme property sets the task default entry scheme. The possible values are as follows:

- Conditional
- Timed
- None

*(Default = None)*

## TaskDefaultSchemeDelayStatement

*The TaskDefaultScheme property sets the task default entry statement for timed entry schemes. (Default =*

*Empty MultiLine)*

## ThisByAccess

*The ThisByAccess property specifies whether to pass the this parameter as an access mode parameter for a non-static operation. (Default = False)*

## ThisName

*The ThisName property specifies the name of the this parameter, which specifies the instance. (Default = this)*

## ThrowExceptions

The ThrowExceptions property specifies the exceptions that an operation can throw. Separate multiple exceptions with commas.

*Default = Empty string*

## VirtualMethodGenerationScheme

The VirtualMethodGenerationScheme property enables compatibility with previous versions for methods of interface and abstract classes. The possible values are as follows:

- Default - The class type is class-wide, but the this parameters are not.
- ClassWideOperations - The class type is not class-wide, but the this parameters are.

*(Default = Default)*

# Package

The Package metaclass contains properties that affect packages.

## Animate

The Animate property specifies whether animation code is generated for an element. You can specify your own animation function by using the CG::Attribute::AnimSerializeOperation property. The semantics of the Animate property is always in favor of the owner settings:

- If a package Animate property is set to False, all the classes owned by the package are not animated, regardless of the class Animate settings.
- If a class Animate property is set to False, all the elements in the class (attributes, operations, relations, and so on) are not animated.

- If an operation Animate property is set to False, all the arguments are not animated.
- If the AnimateArguments property is set to False, all the arguments are not animated, regardless of the specific argument Animate property settings.

*(Default = Checked)*

## ContributesToNamespace

*The ContributesToNamespace property specifies whether the packages contained in this package is declared as children packages of this package. Regardless of the setting, a directory is created for the current package to hold its contained elements. (Default = Checked)*

## DefineNameSpace

The DefineNameSpace property specifies whether a package defines a namespace. A namespace is a declarative region that attaches an additional identifier to any names declared inside it.

*Default = Checked*

## DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- `$Name` - the element name
- `$FullName` - the full path of the element (P1::P2::C.a)
- `$Description` - the text provided on the Description tab of the Features window for the element
- `$Requirements` - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the `$Requirements` keyword in the DescriptionTemplate property to get such comments generated for specific model elements.
- `$Login` - the login name of the user who generated the file
- `$CodeGeneratedDate` - the date on which the code was generated
- `$CodeGeneratedTime` - the time at which the code was generated

- `$RhapsodyVersion` - the version of Rhapsody that was used to generate the file
- For operations: `$Signature` - the operation signature, `$Arguments` - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: `$Type` - the type of the attribute
- For arguments: `$Type` - the argument type, `$Direction` - the argument direction (In, Out, or InOut)
- For events: `$Arguments` - the text provided on the Description tab of the Features window for the event arguments
- For association ends: `$Target` - the other end of the association
- For requirements: `$ID` - the ID specified for the requirement, `$Specification` - the specification text for the requirement
- The values of tags can be included by using the syntax `$tag_name`. For example, if you use a tag called `author` for your classes, you can use the keyword `$author` in the value of `DescriptionTemplate` in order to have the value of the `author` tag included in the element description.
- The values of properties can be included by using the syntax `$property_name`. For example, you can use the keyword `$Animate` in the value of `DescriptionTemplate` in order to have the value of the `Animate` property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as `$Name`)
- Tag keywords
- Property keywords

If the value of a keyword is a `MultiLine` value, then each new line (except the first one) starts with the value of the `<lang>_CG::Configuration::DescriptionBeginLine` property, and each line ends with the value of the `<lang>_CG::Configuration::DescriptionEndLine` property.

*Default = [[ \* `$Description`]] [[ \* `@author $Author`]] [[ \* `@version $Version`]] [[ \* `@see $See`]] [[ \* `@since $Since`]]*

## EventsBaseID

All events are assigned an ID number that is used when code is generated.

If you would like the numbering of events in a package to start at a number different than the default start number used by Rational Rhapsody, you can use the `EventsBaseID` property to specify your own start number.

*Default = 16*

## GenerateDirectory

The `GenerateDirectory` property is used to specify that the code files for classes and files in a package should be generated in a separate directory that has the same name as the package.



If the property is set to False, the code files will be generated in a single directory that contains the generated files for all packages for which this property is set to False.

Note that if this property is set to False and your model contains classes with the same name in different packages, the generated files for these classes will overwrite the previous file with the same name. This will leave you with only one generated file even though the model contains a number of classes with that name.

*Default = True*

## ImplIncludes

*The ImplIncludes property specifies the names (including full paths) of header files to be included at the top of implementation files generated for classes, objects or object types, or packages. Separate multiple file names by using commas, without spaces. (Default = empty string)*

## ImplementationEpilog

The ImplementationEpilog property adds any code that you want to be added as verbatim text (to be ignored by Rational Rhapsody) to the end of the definition of a model element. For example, you could wrap a section of code with an #ifdef-#endif pair, add compiler-specific keywords, or add a #pragma statement. For example, to specify that an operation is available only when the code is compiled with \_DEBUG, set the following properties for the operation:

- Set SpecificationProlog to #ifdef \_DEBUG cr.
- Set SpecificationEpilog to #endif.
- Set ImplementationProlog to #ifdef \_DEBUG cr.
- Set ImplementationEpilog to #endif.

The following table lists whether leading and trailing linefeeds are generated.

*(Default = Empty MultiLine)*

## ImplementationPragmasInContextClause

*The ImplementationPragmasInContextClause property specifies the user-defined pragmas to generate in the context clause of the body. (Default = Empty MultiLine)*

## ImplementationProlog

The ImplementationProlog property adds any code that you want to be added as verbatim text (to be ignored by Rational Rhapsody) to the beginning of the definition of a model element. For example, you could wrap a section of code with an #ifdef-#endif pair, add compiler-specific keywords, or add a #pragma statement. For example, to specify that an operation is available only when the code is compiled with \_DEBUG, set the following properties for the operation:

- Set SpecificationProlog to #ifdef \_DEBUG cr.

- Set SpecificationEpilog to #endif.
- Set ImplementationProlog to #ifdef \_DEBUG cr.
- Set ImplementationEpilog to #endif.

The following table lists whether leading and trailing linefeeds are generated.

*(Default = Empty MultiLine)*

## InitializationCode

The InitializationCode property specifies the user-defined initialization code to add to the package body.  
(Empty MultiLine)

## IsNested

*The IsNested property specifies whether to generate the class or package as nested. (Default = False)*

## IsPrivate

*The IsPrivate property specifies whether to generate the class or package as private. (Default = False)*

## MarkPrologEpilogInAnnotations

The MarkPrologEpilogInAnnotations property specifies whether to generate ignore annotations for the Specification, Implementation Prolog, and Epilog properties so they are ignored during roundtrip. When you insert code element declarations (variables, types, functions, and so on) in the Specification, Implementation Prolog, and Epilog properties, after a full roundtrip those elements are added to the model and are duplicated on the next code generation. When you use the MarkPrologEpilogInAnnotations property, you can have Rational Rhapsody automatically ignore the information specified in the Specification, Implementation Prolog, and Epilog properties instead of adding the ignore annotations manually. The possible values for the MarkPrologEpilogInAnnotations property are as follows:

- None - Rational Rhapsody does not generate any annotations. Any models created before Version 4.1 automatically have this property setting.
- Ignore - Rational Rhapsody generates the `///  
ignore` annotation before the code specified in the Specification, Implementation Prolog, and Epilog properties, and generates the `///  
]` annotation after the code specified in those properties.
- Auto - If the code in the Specification, Implementation Prolog, and Epilog properties is one line (it does not contain any newline characters (`\n`)), no annotations are generated (the same behavior has the None setting). If there is more than one line, Rational Rhapsody generates the `///  
ignore` annotation before the code specified in the Specification, Implementation Prolog, and Epilog properties, and generates the `///  
]` annotation after the code specified in those properties (the same behavior as the Ignore setting).

*During roundtrip, any ignore annotations in the comments of the element are not included in its description. Because the Specification, Implementation Prolog, and Epilog properties are generated between the annotation and its declaration for the element, you cannot rename those elements on roundtrip. If you change the name of an element, it is removed from the model and added with the new*

*name. Some model information (for example, property settings) might be lost. (Default = Auto)*

## **NestingVisibility**

*The NestingVisibility property specifies the visibility of the generated specification of the nested class or package. (Default = Public)*

## **PackageClassNamePolicy**

The PackageClassNamePolicy property specifies the naming policy for classes generated by Rational Rhapsody. Rhapsody generates a class for each package in the Rational Rhapsody Developer for Java model. The possible values are as follows:

- Default - Use the default naming style (the package class name is the same as the package name).
- WithSuffix - Add a suffix to the class name. The suffix is "\_pkgClass".

*Default = WithSuffix*

## **PackageEventIdRange**

The PackageEventIdRange property specifies the maximum number of events allowed in a package. This property is set on the component level.

*Default = 200*

## **Renames**

The Renames property enables one element to rename another element of the same type. You can also rename an element by using a renames dependency. In the case of a conflict, the dependency has precedence. Note the following:

- For attributes, this property works only for static attributes in a class or for attributes in a package.
- For operations, this property contains the name of the operation this operation in renaming. The signatures of the two operations must match.

*(Default = empty string)*

## **SpecificationEpilog**

The SpecificationEpilog property allows you to add code to the end of the declaration of a model element.

*Default = Blank*

## **SpecificationPragmas**

*The SpecificationPragmas property specifies the user-defined pragmas to generate in the specification. (Default = Empty MultiLine)*

## SpecificationPragmasInContextClause

*The SpecificationPragmasInContextClause property specifies the user-defined pragmas to generate in the context clause of the specification. (Default = Empty MultiLine)*

## SpecificationProlog

The SpecificationProlog property allows you to add code to the beginning of the declaration of a model element.

For example, you could add the @Deprecated annotation for an element by entering @Deprecated and a new line as the value of this property.

*Default = Blank*

## SpecIncludes

The SpecIncludes property specifies the names (including full paths) of header files to be included at the top of specification files generated for classes (C++ and Java), objects or object types (C), and packages. Separate multiple file names by using commas, without spaces.

*Default = Empty string*

## Port

The Port metaclass controls whether code is generated for ports.

## DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- `$Name` - the element name
- `$FullName` - the full path of the element (P1::P2::C.a)
- `$Description` - the text provided on the Description tab of the Features window for the element
- `$Requirements` - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the `$Requirements` keyword in the `DescriptionTemplate` property to get such comments generated for specific model elements.
- `$Login` - the login name of the user who generated the file
- `$CodeGeneratedDate` - the date on which the code was generated
- `$CodeGeneratedTime` - the time at which the code was generated
- `$RhapsodyVersion` - the version of Rhapsody that was used to generate the file
- For operations: `$Signature` - the operation signature, `$Arguments` - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: `$Type` - the type of the attribute
- For arguments: `$Type` - the argument type, `$Direction` - the argument direction (In, Out, or InOut)
- For events: `$Arguments` - the text provided on the Description tab of the Features window for the event arguments
- For association ends: `$Target` - the other end of the association
- For requirements: `$ID` - the ID specified for the requirement, `$Specification` - the specification text for the requirement
- The values of tags can be included by using the syntax `$tag_name`. For example, if you use a tag called `author` for your classes, you can use the keyword `$author` in the value of `DescriptionTemplate` in order to have the value of the `author` tag included in the element description.
- The values of properties can be included by using the syntax `$property_name`. For example, you can use the keyword `$Animate` in the value of `DescriptionTemplate` in order to have the value of the `Animate` property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as `$Name`)
- Tag keywords
- Property keywords

If the value of a keyword is a MultiLine value, then each new line (except the first one) starts with the value of the `<lang>_CG::Configuration::DescriptionBeginLine` property, and each line ends with the value of the `<lang>_CG::Configuration::DescriptionEndLine` property.

*Default = [[ \* `$Description`]] [[ \* `@see $See`]] [[ \* `@since $Since`]]*

## Generate

The Generate property specifies whether to generate code for a particular type of element.

*Default = Checked*

## OptimizeCode

Code generation for ports was optimized in version 7.5.3 of Rhapsody, relative to the code generated in previous versions. A new property named OptimizeCode was added with a default value of True. In the Java backward compatibility profile for 7.5.3., the value for this property is set to False so that the old code generation mechanism will be used for ports in older models.

## Relation

The Relation metaclass contains properties that affect relations.

### Add

*The Add property specifies the command used to add an item to a container. (Default = Add\_\$target:c)*

### AddGenerate

*The AddGenerate property specifies whether to generate an Add() operation for relations. (Default = Checked)*

### Animate

The Animate property specifies whether animation code is generated for an element. You can specify your own animation function by using the CG::Attribute::AnimSerializeOperation property. The semantics of the Animate property is always in favor of the owner settings:

- If a package Animate property is set to False, all the classes owned by the package are not animated, regardless of the class Animate settings.
- If a class Animate property is set to False, all the elements in the class (attributes, operations, relations, and so on) are not animated.
- If an operation Animate property is set to False, all the arguments are not animated.
- If the AnimateArguments property is set to False, all the arguments are not animated, regardless of the specific argument Animate property settings.

*(Default = Checked)*

### Clear

*The Clear property specifies the name of an operation that removes all items from a relation. (Default = Clear\_\$target:c)*

## **ClearGenerate**

*The ClearGenerate property specifies whether to generate a Clear() operation for relations. (Default = Checked)*

## **CreateComponent**

*The CreateComponent property specifies the name of an operation that creates a new component in a composite class. (Default = New\_\$target:c)*

## **CreateComponentGenerate**

*The CreateComponentGenerate property specifies whether to generate a CreateComponent operation for composite objects. Setting this property to False is one way to optimize your code for size. (Default = Checked)*

## **DataMemberVisibility**

The DataMemberVisibility property specifies the visibility of the relation data member. For example, if the relation is implemented as a pointer, this property determines whether the pointer data member is declared as public, private, or protected.

*Default = Protected*

## **DeleteComponent**

*The DeleteComponent property specifies the name of an operation that deletes a component from a composite class. (Default = Delete\_\$target:c)*

## **DeleteComponentGenerate**

*The DeleteComponentGenerate property specifies whether to generate a DeleteComponent() operation for composite objects. (Default = Checked)*

## **DescriptionTemplate**

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, `// Description: $Description`

In addition to providing your own text, you can use the following keywords to customize element descriptions.

- `$Name` - the element name
- `$FullName` - the full path of the element (`P1::P2::C.a`)
- `$Description` - the text provided on the Description tab of the Features window for the element
- `$Requirements` - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the `$Requirements` keyword in the `DescriptionTemplate` property to get such comments generated for specific model elements.
- `$Login` - the login name of the user who generated the file
- `$CodeGeneratedDate` - the date on which the code was generated
- `$CodeGeneratedTime` - the time at which the code was generated
- `$RhapsodyVersion` - the version of Rhapsody that was used to generate the file
- For operations: `$Signature` - the operation signature, `$Arguments` - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: `$Type` - the type of the attribute
- For arguments: `$Type` - the argument type, `$Direction` - the argument direction (In, Out, or InOut)
- For events: `$Arguments` - the text provided on the Description tab of the Features window for the event arguments
- For association ends: `$Target` - the other end of the association
- For requirements: `$ID` - the ID specified for the requirement, `$Specification` - the specification text for the requirement
- The values of tags can be included by using the syntax `$tag_name`. For example, if you use a tag called `author` for your classes, you can use the keyword `$author` in the value of `DescriptionTemplate` in order to have the value of the `author` tag included in the element description.
- The values of properties can be included by using the syntax `$property_name`. For example, you can use the keyword `$Animate` in the value of `DescriptionTemplate` in order to have the value of the `Animate` property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as `$Name`)
- Tag keywords
- Property keywords

If the value of a keyword is a `MultiLine` value, then each new line (except the first one) starts with the



value of the <lang>\_CG::Configuration::DescriptionBeginLine property, and each line ends with the value of the <lang>\_CG::Configuration::DescriptionEndLine property.

*Default = [[ \* \$Description]] [[ \* @see \$See]] [[ \* @since \$Since]]*

## Find

*The Find property specifies the name of an operation that locates an item among relational objects. (Default = Find\_\$target:c)*

## FindGenerate

*The FindGenerate property specifies whether to generate a Find() operation for relations. (Default = False)*

## Get

*The Get property specifies the name of an operation that retrieves the relation currently pointed to by the iterator. (Default = Get\_\$target:c)*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index. The ContainerTypes>::Relationtype::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the at() operation for the container to retrieve the item at the indexed position:  
\$cname-at(\$index)

*Default = get\$cname:cAt*

## GetAtGenerate

The GetAtGenerate property specifies whether to generate a getAt() operation for relations. The possible values are as follows:

- Checked - Generate a getAt() operation for relations.
- Cleared - Do not generate a getAt() operation for relations. Setting the GetAtGenerate property to False is one way to optimize your code for size.

*Default = Cleared*

## GetEnd

*The GetEnd property specifies the name of an operation that points the iterator to the last item in a collection. (Default = Get\_\$target:cEnd)*

## GetEndGenerate

The *GetEndGenerate* property specifies whether to generate a *GetEnd()* operation for relations. (Default = *Checked*)

## GetGenerate

The *GetGenerate* property specifies whether to generate accessor operations for relations. (Default = *Checked*)

## GetKey

The *GetKey* property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key. For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps: *\$cname-operator[](\$keyName)*

(Default = *get\$cname:c*)

## GetKeyGenerate

The *GetKeyGenerate* property specifies whether to generate *getKey()* operations for relations. Setting this property to *Cleared* is one way to optimize your code for size.

Default = *Checked*

## ImplementationEpilog

The *ImplementationEpilog* property adds any code that you want to be added as verbatim text (to be ignored by Rational Rhapsody) to the end of the definition of a model element. For example, you could wrap a section of code with an *#ifdef-#endif* pair, add compiler-specific keywords, or add a *#pragma* statement. For example, to specify that an operation is available only when the code is compiled with *\_DEBUG*, set the following properties for the operation:

- Set *SpecificationProlog* to *#ifdef \_DEBUG cr.*
- Set *SpecificationEpilog* to *#endif.*
- Set *ImplementationProlog* to *#ifdef \_DEBUG cr.*
- Set *ImplementationEpilog* to *#endif.*

The following table lists whether leading and trailing linefeeds are generated.

(Default = *Empty MultiLine*)

## ImplementationProlog

The ImplementationProlog property adds any code that you want to be added as verbatim text (to be ignored by Rational Rhapsody) to the beginning of the definition of a model element. For example, you could wrap a section of code with an #ifdef-#endif pair, add compiler-specific keywords, or add a #pragma statement. For example, to specify that an operation is available only when the code is compiled with \_DEBUG, set the following properties for the operation:

- Set SpecificationProlog to #ifdef \_DEBUG cr.
- Set SpecificationEpilog to #endif.
- Set ImplementationProlog to #ifdef \_DEBUG cr.
- Set ImplementationEpilog to #endif.

The following table lists whether leading and trailing linefeeds are generated.

*(Default = Empty MultiLine)*

## ImplementWithStaticArray

The ImplementWithStaticArray property specifies whether to implement relations as static arrays. The possible values are as follows:

- Default - Rational Rhapsody provides the appropriate implementation for all fixed and bounded relations.
- FixedAndBounded - All fixed and bounded relations are generated into static arrays.

To generate C-like code in C++ or Java, modify the value of the ImplementWithStaticArray property to FixedAndBounded.

*Default = Default*

## InitializationStyle

The InitializationStyle property determines how relations are initialized in the generated code. The possible values are:

- InClass - relation is initialized when it is declared
- InConstructor - relation is initialized in the constructor

This property can be set at the level of individual relations and higher.

*Default = InClass*

Sample code:

When the value is set to InClass, the generated code will resemble the following:

```
public class car {  
  
protected ArrayList<wheel> itsWheels = new ArrayList<wheel>(); /// link itsWheels
```

```
// Constructors

///  

public car() {  

}
```

When the value is set to InConstructor, the generated code will resemble the following:

```
///  

public class car {  

    protected ArrayList<wheel> itsWheels; ///  

// Constructors  

///  

public car() {  

    itsWheels = new ArrayList<wheel>();  

}
```

## InitializeComposition

The InitializeComposition property controls how a composition relation is initialized. The possible values are as follows:

- InInitializer
- InRecordType
- None

*(Default = InInitializer)*

## InitialValue

Use the InitialValue property to set an initial value; for example, if you want to specify an initial value for a static association.

See also the <lang>\_CG::Relation:Static property.

*Default = Empty string*

## Inline

The Inline property specifies how inline operations are generated. Which operations are affected by the Inline property depends on the metaclass:

- Attribute - Applies only to operations that handle attributes (such as accessors and mutators)
- Operation - Applies to all operations
- Relation - Applies only to operations that handle relations

*Inlining in Rational Rhapsody Developer for Java Because inlining has no meaning in Java, the Inline property is set to none. (Default = none)*

## IsAliased

*The IsAliased property is a Boolean value that specifies whether attributes are aliased. (Default = False)*

## JavaAnnotation

The JavaAnnotation property is used by the Rational Rhapsody code generator to insert Java annotations into generated code.

This property is used primarily for regenerating code that was reverse engineered. When you reverse engineer code that contains Java annotations, the value of the `JAVA_ReverseEngineering::ImplementationTrait::ImportJavaAnnotation` property determines how Rational Rhapsody handles the annotation code. If the value of this property is set to Verbatim, then Rational Rhapsody does not import annotations as model elements. Rather, the annotation code is stored as the value of the JavaAnnotation property. When code is later regenerated, it does include the code that was stored in this property.

*Default = Blank*

## Kind

The Kind property specifies the kind of operation that should be generated for an element. The kind of operations that can be generated is language-dependent (for example, virtual and abstract exist only in C++ and Java). In Java, Kind can be defined only for attributes and operations, but not for relations. This property affects class operations, in addition to accessors and mutators for relations and attributes. The possible values are as follows:

- common - Class operations and accessor/mutator are non-virtual.
- virtual - Class operations and accessor/mutator are virtual. This type is valid for C++ and Java only.
- abstract - Class operations and accessor/mutator are pure virtual. This type is valid for C++ and Java only.

*(Default = common)*

## ObjectInitialization

The ObjectInitialization property specifies what kind of initialization occurs for the initial instances of a

configuration. The possible values are as follows:

- Full - Instances are initialized and their behavior is started.
- Creation - Instances are initialized but their behavior is not started.
- None- Instances are not initialized and their behavior is not started.

*(Default = Full)*

## Remove

*The Remove property specifies the name of an operation that removes an item from a relation. (Default = Remove\_\$target:c)*

## RemoveGenerate

*The RemoveGenerate property specifies whether to generate a Remove() operation for relations. (Default = Checked)*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: \$cname-erase(\$keyName)

*Default = remove\$cname:c*

## RemoveKeyGenerate

The RemoveKeyGenerate property specifies whether to generate a removeKey() operation for qualified relations. Setting this property to False is one way to optimize your code for size.

*Default = Checked*

## RemoveKeyHelpersGenerate

The RemoveKeyHelpersGenerate property controls the generation of the relation helper methods (for example, \_removeItsX() and \_\_removeItsX()). The possible values are as follows:

- True - Generate the helpers whenever code generation analysis determines that the methods are needed.
- False - Never generate the helpers.
- FromModifier - Generate the helpers based on the value of the CPP\_CG::Relation::RemoveKey property.

*Default = True*

## SafeInitScalar

The SafeInitScalar property specifies whether to initialize scalar relations as null pointers.

*Default = Cleared*

## Set

*The Set property specifies the name of the mutator generated for scalar relations. (Default = Set\_\$target:c)*

## SetGenerate

*The SetGenerate property specifies whether to generate mutators for relations. (Default = Checked)*

## SpecificationEpilog

The SpecificationEpilog property allows you to add code to the end of the declaration of a model element.

*Default = Blank*

## SpecificationProlog

The SpecificationProlog property adds code to the beginning of the declaration of a model element (such as a configuration or class). For example, to create an abstract class in Java, you can set the SpecificationProlog property for the class to “abstract.” You must include the space after the word “abstract.” If the visibility for the class is set to default, the following class declaration is generated in the .java file: abstract class classname {...} The SpecificationProlog property allows you to add compiler-specific keywords, add a #pragma statement, or wrap a section of code with an #ifdef-#endif pair. For example, to specify that an operation is available only when the code is compiled with \_DEBUG, set the following properties for the operation:

- Set SpecificationProlog to #ifdef \_DEBUG cr.
- Set SpecificationEpilog to #endif.
- Set ImplementationProlog to #ifdef \_DEBUG cr.
- Set ImplementationEpilog to #endif.

The following table lists whether leading and trailing linefeeds are generated.

Metaclass	Leading Linefeed Added?	Trailing Linefeed Added?	Class	Yes	No	Package	Yes	Yes
-----------	-------------------------	--------------------------	-------	-----	----	---------	-----	-----

(Empty MultiLine)

## Static

The Static property is a Boolean value that determines whether class-wide relations are enabled. Class-wide members of a class are shared between all instances of that class and are mapped as static. When a relation is tagged as static:

- The data member is generated as static (with the static keyword).
- The relation accessors are generated as static.
- The mutators of directional relations are generated as static. The mutators of symmetric relations are generated as common (non-static) operations.

Note the following behavior and restrictions:

- If there are links between instances based on static relations, code generation initializes all the valid links. In case of a limited relation size, the last initialization is preserved.
- When you generate instrumented code (animation or tracing), relation NOTIFY calls are not added to static relation mutators.
- Animation associates static relations with the class instances, not the class itself.
- In an instrumented application (animation or tracing), the static relations names appear in each instance node; however, the values of directional static relations are not visible.

See also the `CG::Relation::Containment`, `Containertype::Relationtype::CreateStatic`, and `Containertype::Relationtype::InitStatic` properties.

*Default = Cleared*

## Visibility

*The Visibility property specifies the visibility of that kind of model element. Code generation maps the visibility specified for an element to the same visibility in the generated language. (Default = Public)*

## Requirement

The Requirement metaclass contains properties related to the code generated for requirements.

## DescriptionTemplate

The DescriptionTemplate property is used to specify how the description of an element should be generated in the code.

If the property is left blank, Rational Rhapsody uses the default description generation template.

Note that this property is included in a number of different property metaclasses, allowing you to create different customized description templates for different types of model elements, for example, packages, classes, and operations.

If you are customizing the template, you must include the comment characters in order to have the text generated correctly as a comment, for example, `// Description: $Description`



In addition to providing your own text, you can use the following keywords to customize element descriptions.

- `$Name` - the element name
- `$FullName` - the full path of the element (P1::P2::C.a)
- `$Description` - the text provided on the Description tab of the Features window for the element
- `$Requirements` - There is a global option on the Settings tab of the Features window for configurations that allows you to request that comments be generated for any requirements that are realized by model elements in the configuration. If you elect to turn off this option, you can use the `$Requirements` keyword in the `DescriptionTemplate` property to get such comments generated for specific model elements.
- `$Login` - the login name of the user who generated the file
- `$CodeGeneratedDate` - the date on which the code was generated
- `$CodeGeneratedTime` - the time at which the code was generated
- `$RhapsodyVersion` - the version of Rhapsody that was used to generate the file
- For operations: `$Signature` - the operation signature, `$Arguments` - the text provided on the Description tab of the Features window for the operation arguments
- For attributes: `$Type` - the type of the attribute
- For arguments: `$Type` - the argument type, `$Direction` - the argument direction (In, Out, or InOut)
- For events: `$Arguments` - the text provided on the Description tab of the Features window for the event arguments
- For association ends: `$Target` - the other end of the association
- For requirements: `$ID` - the ID specified for the requirement, `$Specification` - the specification text for the requirement
- The values of tags can be included by using the syntax `$tag_name`. For example, if you use a tag called `author` for your classes, you can use the keyword `$author` in the value of `DescriptionTemplate` in order to have the value of the `author` tag included in the element description.
- The values of properties can be included by using the syntax `$property_name`. For example, you can use the keyword `$Animate` in the value of `DescriptionTemplate` in order to have the value of the `Animate` property included in the element description.

In cases where there are conflicts, keywords are resolved in the following order:

- Predefined keywords (such as `$Name`)
- Tag keywords
- Property keywords

If the value of a keyword is a MultiLine value, then each new line (except the first one) starts with the value of the `<lang>_CG::Configuration::DescriptionBeginLine` property, and each line ends with the value of the `<lang>_CG::Configuration::DescriptionEndLine` property.

*Default = Realizes requirement \$Name #ID: \$Specification*

# Statechart

The Statechart metaclass contains the statechart code generation properties.

## GenerateActionOnExitOrderForNestedStatechartOldWay

Before version 7.5.3, the code generated for actions on exit was not put in the correct location in the generated code. This was corrected in version 7.5.3. In order to maintain the previous code generation behavior for older models, a property called `[lang]_CG::Statechart::GenerateActionOnExitOrderForNestedStatechartOldWay` was added to the C, CPP, and Java backward compatibility profiles for 7.5.3 with a value of True.

## StatechartImplementation

Prior to version 7.3 of Rational Rhapsody, the transition-handling code that is generated by Rational Rhapsody used a switch statement to represent the possible states. Beginning with version 7.3, this code uses an if/else structure. To allow older models to use the previous code generation behavior, a property called `StatechartImplementation` was added to the Pre73 profiles that ensure compatibility with earlier versions. The possible values for the property are:

- `SwitchOnly` - transition-handling code uses a switch statement to represent the possible states
- `Default` - the transition-handling code uses an if/else structure to represent the possible states

*Default = SwitchOnly*

# Type

The Type metaclass contains a property that affects the visibility of data types.

## AnimEnumerationTypeImage

*The `AnimEnumerationTypeImage` property is a Boolean value that determines whether the `Image` attribute is used for enumerated types when you use animation. (Default = False)*

## DeclarationPosition

The `DeclarationPosition` property specifies where the type declaration is displayed. The possible values are as follows:

- `BeforeClassRecord` - The type declaration is displayed before the class record (CR) declaration if CR has a visibility set to public, and before the class record forward declaration if CR has a visibility set to

private.

- **AfterClassRecord** - The type declaration is displayed after the class record declaration if CR has a visibility set to public, and after the class record forward declaration if CR has a visibility set to private.
- **StartOfDeclaration** - The type declaration is displayed among the first declarations (together with other types having the same settings) in the public section if CR has a visibility set to public, and among the first declarations in the private section (together with other types having the same settings) if CR has a visibility set to private.
- **EndOfDeclaration** - The type declaration is displayed among the last declarations (together with other types having the same settings) in the public section if CR has a visibility set to public, and among the last declarations in the private section (together with other types having the same settings) if CR has a visibility set to private.

*(Default = BeforeClassRecord)*

## EnumerationAsTypedef

*The EnumerationAsTypedef property specifies whether the generated enum should be wrapped by a typedef. This property is applicable to enumeration types in C and C++. (Default = Checked)*

## In

The In property specifies how code is generated when the type is used with an argument that has the modifier "In".

*Default = \$type*

## InOut

The InOut property specifies how code is generated when the type is used with an argument that has the modifier "InOut".

*Default = \$type*

## IsLimited

*The IsLimited property determines whether the class or record type is generated as limited. (Default = False)*

## LanguageMap

*The LanguageMap property specifies the Ada declaration for Rational Rhapsody language-independent types. (Default = empty string)*

## Out

The Out property specifies how code is generated when the type is used with an argument that has the modifier "Out".

*Default = \$type*

## PrivateName

*The PrivateName property specifies the pattern used to generate names of private operations in C. (Default = \$typeName)*

## PublicName

*The PublicName property specifies the pattern used to generate names of public operations in C. (Default = \$objectName\_\$typeName)*

## ReferenceImplementationPattern

*The ReferenceImplementationPattern property specifies how the Reference option for attribute/typedefs (composite types) is mapped to code. See the Rational Rhapsody help for information about composite types. (Default = "\*\*")*

## ReturnType

The ReturnType property specifies how code is generated when the type is used as a return type.

*Default = \$type*

## StructAsTypedef

*The StructAsTypedef property specifies whether the generated struct should be wrapped by a typedef. This property is applicable to structure types in C and C++. (Default = Checked)*

## TriggerArgument

The TriggerArgument property is used for mapping event and triggered operation arguments to code instead of the In, InOut, and Out properties. A different property is required because of code generation limitations related to event arguments. See also:

- In
- InOut
- Out

*Default = \$type*

## UnionAsTypedef

*The UnionAsTypedef property specifies whether the generated union should be wrapped by a typedef. This property is applicable to union types in C and C++. (Default = Checked)*

## Visibility

The Visibility property specifies the visibility of the model element. Code generation maps the visibility specified for an element to the same visibility in the generated language. The possible values are as follows:

- Public - The model element is public.
- Protected - The model element is protected.
- Private - The element is private.

*(Default = Public)*

# JAVA\_ReverseEngineering

---

In addition to the ReverseEngineering subject, Rational Rhapsody provides language-specific subjects to control how the product imports legacy code. Because most of the properties are identical for each language, they are represented with the JAVA tag, where JAVA can be C, CPP, or Java. Any language-specific properties are clearly labeled. In general, most of the reverse engineering (RE) properties have graphical representation in the Reverse Engineering Advanced Options window. You should change the options by using this window instead of the corresponding properties.

## Filtering

The Filtering metaclass contains properties that control which items are analyzed during the reverse engineering operation.

### AnalyzeGlobalFunctions

*The AnalyzeGlobalFunctions property specifies whether to analyze global functions. (Default = True)*

### AnalyzeGlobalTypes

*The AnalyzeGlobalTypes property specifies whether to analyze global types. (Default = True)*

### AnalyzeGlobalVariables

*The AnalyzeGlobalVariables property specifies whether to analyze global variables. (Default = True)*

### CreateReferenceClasses

The CreateReferenceClasses property specifies whether to create external classes for undefined classes that result from forward declarations and inheritance. By default, reference classes are created (as in previous versions of Rational Rhapsody). If the incomplete class cannot be resolved, the tool deletes the incomplete class if this property is set to Cleared. In some cases, the class cannot be deleted (for example, a class referenced by a typedef type).

*Default = Checked*

### IncludeInheritanceInReference

The IncludeInheritanceInReference property specifies whether to include inheritance information in reference classes.

*Default = Cleared*

## ReferenceClasses

The ReferenceClasses property specifies which classes to model as reference classes. Reference classes are classes that can be mentioned in the final design as placeholders without having to specify their internal details. For example, you can include the MFC classes as reference classes, without having to specify any of their members or relations. They would simply be modeled as terminals for context, to show that they are acting as superclasses or peers to other classes.

*Default = empty string*

## ReferenceDirectories

The ReferenceDirectories property specifies which directories (and subdirectories) contain reference classes.

*Default = empty string*

## ImplementationTrait

The ImplementationTrait metaclass contains properties that determine the implementation traits used during the reverse engineering operation.

## AnalyzeIncludeFiles

The AnalyzeIncludeFiles property specifies which, if any, include files should be analyzed during reverse engineering. The possible values are as follows:

- AllIncludes—Analyze all include files.
- IgnoreIncludes—Ignore all include files.
- OnlyFromSelected—Analyze the specified include files only.
- OnlyLogicalHeader—Analyze the logical header files only.

*Default = OnlyFromSelected*

## CreateDependencies

The CreateDependencies property (under C and JAVA\_ReverseEngineering::ImplementationTrait) is used during reverse engineering (RE) for creating dependencies from include statements found in the imported code. This property determines whether the RE utility creates dependencies. Reverse engineering imports include statements as dependencies if the option Create Dependencies from Includes is set in the Rational Rhapsody GUI. This operation is successful if the reverse engineering utility analyzes both the included file and the source - and the source and included files contain class declarations for creating the dependencies between them. If there is not enough information, the includes are not converted dependencies. This can happen in the following cases:

- The include file was not found, or is not in the scope Input tab settings.
- A class is not defined in the include file or source file, so the dependency could not be created.

If the dependency is not created successfully, the include files that were not converted to dependencies are imported to the `JAVA_CG::Class::SpecIncludes` or `ImpIncludes` properties so you do not have to re-create them manually. If the include file is in the specification file, the information is imported to the `SpecIncludes` property; if it is in the implementation file, the information is imported to the `ImpIncludes` property. If a file contains several classes, include information is imported for all the classes in the file.

The possible values for this property are as follows:

- **None** - Nothing is imported from include statements.
- **DependenciesOnly** - Model dependencies are created from include statements when it is possible to do so. This is the RE behavior of previous versions of Rational Rhapsody.
- **All** - The reverse engineering utility attempts to map the include file as a dependency. If it fails, the information is written to a property.

In previous versions of Rational Rhapsody, this property was a Boolean value. For compatibility with earlier versions, the old values are mapped as follows:

The old **Checked** value is mapped as the new **DependenciesOnly** value

The old **Unchecked** value is mapped as the **None** value

In addition to influencing reverse engineering, the `CreateDependencies` property also impacts the reverse engineering of user code added to model elements. The rules for interpreting `#include` and `friend` declarations for reverse engineering are as follows:

- Any `#include OTHER` in `FILE` is represented as a **Uses** dependency between each (outer) packages or classes in `FILE` to any (outer) packages or class in `OTHER`.
- If `OTHER` is not a specification file, the information is lost.
- If `FILE` is a specification file, the `RefereeEffect` is `Specification`. If `FILE` is an implementation file, the `RefereeEffect` is `Implementation`. Otherwise, the information is lost.

1. The way to decide if a file is a specification or an implementation file is defined elsewhere.
2. Any forward of a class or a package (by way of a namespace) `E` in `FILE` is represented as a **Uses** dependency between each (outer) packages/classes in `FILE` to `E`. The `RefereeEffect` is `Existence`
3. This dependency is not added, if a **Uses** dependency can be matched.
4. Redundant **Uses** dependencies are removed. For example, when a relation is synthesized from a pointer to `B`, it is not necessary to add a **Uses** dependency.
5. A friend `F` (only when `F` is a class) of class `C` is represented as a dependency with `DependencyType` to be `Friendship` from `F` to `C`.

*Default = All*

## CreateFilesIn

The `CreateFilesIn` property is a placeholder for the reverse engineering option `Create File-s In` option. See



the Rational Rhapsody help for more information. You should not set this value directly. The default value for C is Package; the default values for the other languages is None.

## DataTypesLibrary

The Mapping tab of the Reverse Engineering Options window allows you to specify a list of types that should be modeled as "Language" types. You can add individual types to the list or groups of types that you have previously defined as data types for a specific library.

If you select the option of adding a library, you are presented with a drop-down list of libraries to choose from. The libraries on this list are taken from the value of the DataTypesLibrary property. You can add a number of libraries to the drop-down list by using a comma-separated list of names as the value for this property.

When you select a library from the drop-down list, all of the types that were defined for that library are added to the list of types.

You define types for a library by carrying out the following steps:

- In the relevant .prp file, under the Java\_ReverseEngineering subject, add a metaclass with the name of the library (use the same name you used in the value of the DataTypesLibrary property).
- Under the new metaclass, add a property called DataTypes.
- For the value of the DataTypes property that you added, enter a comma-separated list of the types that you want to include for that library.
- Now, if you select the library from the drop-down list displayed on the Mapping tab, the types you defined with the DataTypes property is automatically added to the list of types that should be modeled as "Language" types.

*Default = Blank*

## ImportAsExternal

The ImportAsExternal property specifies whether the elements contained in the files you are reverse engineering should be brought into the model as "external" elements. This means that code is not generated for these elements during code generation.

This property corresponds to the Import as External check box on the Mapping tab of the Reverse Engineering Options window.

*Default = Cleared*

## ImportDefineAsType

The ImportDefineAsType property is a Boolean value that specifies how to import a #define. Note that models created before Version 5.2 automatically have this property overridden (set to True) when the model is loaded. The possible values are as follows:

- True—Import a #define as a user type.

- False—Import a #define as a constant variable, constant function, or type according to the following policy:
- If the #define has parameters, Rational Rhapsody creates a constant function. This applies to Rational Rhapsody Developer for C only.
- If the #define does not have parameters and its value includes only one line, Rational Rhapsody creates a constant variable. In Rational Rhapsody Developer for C++, the CG::Attribute::ConstantVariableAsDefine property is set to True.
- If the #define was not imported as a variable or function, Rational Rhapsody creates a type (the behavior of Rational Rhapsody 5.0.1).

*Default = False*

## ImportJavaAnnotation

The ImportJavaAnnotation property allows you to specify how reverse engineering should handle Java annotations in your Java code. The property can take any of the following values:

- None - Code relating to Java annotations is ignored (and therefore annotations do not appear in the code that is later generated from the model).
- Model - All annotation-related code is brought into the model as elements that are visible in the browser (AnnotationType, JavaAnnotation, and AnnotationUsage).
- Verbatim - Code relating to Java annotations is processed. AnnotationTypes is brought into the model as visible elements, but annotation usage for individual elements are not translated into elements in the model. For annotation usage, the text is stored by using JavaAnnotation properties so that the annotations can be included in the code that is generated from the model.
- Mixed - Rational Rhapsody tries to bring annotation-related code into the model as visible elements. Where this is not possible, the product stores the text as property values so that the annotations can be regenerated in the code.

*Default = Verbatim*

## ImportStructAsClass

The ImportStructAsClass property is a Boolean value specifies how structs in external code are imported during reverse engineering. The possible values are as follows:

- True—structs are imported as classes (as in Rational Rhapsody 5.0 and earlier).
- False—structs are imported as types of kind Structure.

*Default = False*

## MapToPackage

The MapToPackage property allows you to specify how the code elements you are reverse engineering should be divided into packages.

The property represents the options that appear in the Map to Package section of the Mapping tab in the Reverse Engineering Options window.

When the value of the property is set to Directory, a separate package is created for each subdirectory in the directory you have chosen to reverse engineer. The elements found in the files in each subdirectory is added to the package that corresponds to that subdirectory.

If you set the value of this property to User, then Rational Rhapsody puts all reverse engineered elements into a single package in the model. The name of the package is taken from the `Java_ReverseEngineering::ImplementationTrait::UserPackage` property.

*Default = Directory*

## ModelStyle

The ModelStyle property determines how model elements are opened in the browser after reverse engineering - by using a file-based functional approach or by using an object-oriented approach based on classes (the corresponding property values are Functional and ObjectBased).

This property corresponds to the Modeling Policy radio buttons on the Mapping tab of the Reverse Engineering Options window.

Note that for C++ and Java, the file-based approach can only be used for visualization purposes. Rational Rhapsody does not generate code from the model for elements imported that uses the Functional option. (Notice that in the Reverse Engineering Options window, you can only select the File radio button if you first select the Visualization Only option.)

*Default = Functional in RiC, ObjectBased in RiC++ and RiJ*

## PackageForExternals

If the value of the UsePackageForExternals property is set to True, the Rational Rhapsody reverse engineering feature puts all external elements in a separate package. You can control the name of this package by changing the value of the PackageForExternals property.

*Default = Externals*

## PreCommentSensibility

During reverse engineering, a comment that comes immediately before the code for an element is considered a comment for that element, and the comment text is brought into Rational Rhapsody as the description for that element.

The PreCommentSensibility property is used to specify the maximum number of lines by which a comment can precede the code for an element and still be considered a comment for that element. Any comment that precedes an element by more than the number of lines specified is considered a global comment.

A value of 1 means that a comment must appear on the line prior to the code for an element to be considered a comment for that element.

*Default = 2*

## ReflectDataMembers

The ReflectDataMembers property determines how the visibility of attributes is brought into the model when code is reverse engineered. The property affects both the visibility of the attribute in the regenerated code and the generation of get and set operations for the attribute. The property can take any of the following values:

- None - The visibility used for attributes is the same as that specified in the code that was reverse engineered. However, Rational Rhapsody generates public get/set operations for the attributes regardless of the visibility specified.
- VisibilityOnly - The visibility used for attributes is the same as that specified in the code that was reverse engineered. In addition, Rational Rhapsody generates get/set operations for the attribute with the same visibility. For example, if the visibility for an attribute in the original code was private, the visibility is private in the regenerated code and the code also includes private get/set operations for the attribute.
- VisibilityAndHelpers - The visibility used for attributes is the same as that specified in the code that was reverse engineered. Rational Rhapsody does not generate get/set operations for the attribute if the original code did not contain such operations.

Note that when the property is set to VisibilityAndHelpers, get/set operations are not generated for attributes, and Rational Rhapsody does not generate any of its automatically-generated operations such as default constructors.

*Default = VisibilityAndHelpers*

## RespectCodeLayout

The RespectCodeLayout property determines whether or not Rational Rhapsody saves information about the mapping of classes to files when reverse engineering code. The possible values for the property are:

- Mapping - Rational Rhapsody remembers which classes are contained in each of the files reverse engineered. When code is regenerated after reverse engineering, the classes are generated in the same files, such that if a file contained more than one class, it still contains more than one class when the code is regenerated.
- None - Rational Rhapsody does not store any information regarding the mapping of classes to files. When code is regenerated after reverse engineering, each class is generated in its own file.

*Default = None*

## RootDirectory

This property specifies the root directory for reverse engineering. This root directory might contain all the folders that should become package during the reverse engineering process. Rhapsody builds the package hierarchy according to the folder tree from the specified path.

*Default = empty string*

## RoundtripArgumentAsExistingType

Prior to version 8.0.3 of Rational Rhapsody, operation arguments whose type was a type defined in the model were imported into the model as "on-the-fly" types. Beginning in version 8.0.3, for CPP and Java code, if the type of the argument is a class defined in the model, the argument type is identified correctly when imported into the model. To preserve the previous code generation behavior for pre-8.0.3 models, the RoundtripArgumentAsExistingType property was added to the backward compatibility settings for C++ and Java with a value of False.

## RoundtripOperationReturnTypeAsExistingType

Prior to version 8.0.5 of Rational Rhapsody, operation return types which were types defined in the model were imported into the model as "on-the-fly" types. Beginning in version 8.0.5, for CPP and Java code, if the return type is a class defined in the model, the return type is identified correctly when imported into the model. To preserve the previous code generation behavior for pre-8.0.5 models, the RoundtripOperationReturnTypeAsExistingType property was added to the backward compatibility settings for C++ and Java with a value of False.

## UseCalculatedRootDirectory

This property controls the use of the <lang>\_ReverseEngineering::Implementation::RootDirectory property.

The possible values are:

- Never - Do not calculate the root directory.
- Always - Calculate the root directory and override the RootDirectory property.
- Auto - Ask the user if they want to override the value in the RootDirectory property if it is different from the calculated root directory. If the RootDirectory property is empty, Rational Rhapsody uses the calculated value without asking. This is the default value.

*Default = Auto*

## UsePackageForExternals

When Rational Rhapsody generates code, it does not regenerate code for elements that have been brought in as "external" elements. If you would like the reverse engineering feature to put all external elements into a separate package in the model, set the value of the UsePackageForExternals property to Checked. When a separate package is used, the name of the package is taken from the value of the PackageForExternals property.

*Default = Cleared*

## UserDataTypes

The UserDataTypes specifies classes to be modeled as data types. This property corresponds to types entered in the Add Type window.

*Default = empty string*

## UserPackage

When reverse engineering files, Rational Rhapsody allows you the option of having packages created for each subdirectory or having all of the reverse-engineered elements placed in a single package. This option is controlled by the property `Java_ReverseEngineering::ImplementationTrait::MapToPackage`.

When `MapToPackage` is set to "User", you can use the `UserPackage` property to provide the name that you would like Rational Rhapsody to use for the single package that contains all of the reverse-engineered elements.

You can specify a nested package by using the following syntax: `package1::package2`

If the model already contains a package with the specified name, the reverse-engineered elements are put in that package. If not, Rational Rhapsody creates the package.

This property corresponds to the text field provided for the package name in the Map to Package section of the Mapping tab in the Reverse Engineering Options window.

*Default = ReverseEngineering*

## Main

The metaclass `Main` contains properties that define the file extensions used for filtering files in the reverse engineering file selection window, as well as properties that enable jumping to problematic lines of code by double-clicking messages in the Output window.

## ErrorMessageTokensFormat

When errors are encountered during reverse engineering, they are displayed in the Rational Rhapsody Output window. If you double-click the error message, you are taken to the problematic line in the relevant source file.

This ability is made possible by the values provided for the `ParseErrorMessage` and `ErrorMessageTokensFormat` properties.

The value of the `ParseErrorMessage` property is a regular expression that extracts the relevant filename and line number information from the Rational Rhapsody-generated error message. The value of the `ErrorMessageTokensFormat` property is then used to interpret the information that was extracted from the error message.

The value of the `ErrorMessageTokensFormat` property consists of a comma-separated list of keyword-value pairs representing the number of tokens contained in the extracted information, which token represents the filename, and which token represents the line number.

Users should not change the value of this property.

*Default = ToTalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2*

## ImplementationExtension

The ImplementationExtension property is only used for C and C++. It has no effect in Rational Rhapsody Developer for Java.

## ParseErrorMessage

When errors are encountered during reverse engineering, they are displayed in the Rational Rhapsody Output window. If you double-click the error message, you are taken to the problematic line in the relevant source file.

This ability is made possible by the values provided for the ParseErrorMessage and ErrorMessageTokensFormat properties.

The value of the ParseErrorMessage property is a regular expression that extracts the relevant filename and line number information from the Rational Rhapsody-generated error message. The value of the ErrorMessageTokensFormat property is then used to interpret the information that was extracted from the error message.

Users should not change the value of this property.

*Default = "([a-zA-Z\_]+[:0-9a-zA-Z\_\\.\\])\*"[:][ ]\*LINE[ ]\*([0-9]+)*

## SpecificationExtension

The SpecificationExtension property is used to specify the filename extensions that should be used to filter files in the reverse engineering file selection window.

You can specify a number of extensions. They should be entered as a comma-separated list.

*Default = java*

## MFC

The MFC metaclass contains a property that affects the MFC type library.

## DataTypes

The DataTypes property specifies classes to be modeled as MFC data types. There is only one predefined library (MFC) that contains only one class (CString). You can, however, expand this short list of classes by

the addition of classes in this property or the creation of new libraries in the property files factory.prpfactory and site.prpsite.

*Default = Cstring*

## MSVC60

The MSVC60 metaclass contains properties used to control the Microsoft Visual C++ environment.

### Defined

The Defined property specifies symbols that are defined for the Microsoft Visual C++ version 6.0 (MSVC60) preprocessor. These symbols are automatically filled into the Name list of the Preprocessing tab of the Reverse Engineering Options window when you select Add > Dialect: MSVC60. The default value is as follows:

```
__STDC__, __STDC_VERSION__, __cplusplus, __DATE__,  
__TIME__, WIN32, __cdecl, __cdecl, __int64=int, __stdcall,  
__export, __export, AFX_PORTABLE, _M_IX86=500, __declspec,  
__MSC_VER=1200, __inline=inline, __far, __near, __far, __near,  
__pascal, __pascal, __asm, __finally=catch, __based,  
__inline=inline, __single_inheritance, __cdecl, __int8=int,  
__stdcall, __declspec, __int16=int, __int32=int, __try=try,  
__int64=int, __virtual_inheritance, __except=catch, __leave=catch, __fastcall, __multiple_inheritance)
```

### IncludePath

The IncludePath property specifies necessary include paths for the Microsoft Visual C++ preprocessor. It is possible to specify the path to the site installation of the compiler as part of the site.prp, thus doing it only once and not for every project.

*Default = empty string*

### Undefined

The Undefined property specifies symbols that must be undefined for the Microsoft Visual C++ preprocessor.

*Default = empty string*

## Parser



The metaclass Parser contains properties that can be used to modify the way the parser handles code during reverse engineering.

## Defined

The Defined property specifies symbols and macros to be defined by using `#define`. For example, you can enter the following to define `name>` as text with the appropriate intermediate character: `/D name{=|#}text`

*Default = empty string*

## Dialects

The Dialects property specifies which symbols are added to the Preprocessing tab of the Reverse Engineering window when that dialect is selected. The default value is `MSVC60`, which is itself defined by a metaclass of the same name under the `JAVA_ReverseEngineering` subject. This dialect specifies the symbols that are defined for the Microsoft Visual C++ environment. You can define your own dialect (in the `site.prp` file) and select it in the Dialects property.

*Default = Empty string.*

## IncludePath

The Preprocessing tab of the Reverse Engineering Options window allows you to specify an include path (classpath for Java) for the parser to use. The `In propertycludePath` represents this path.

For the value of this property, you can enter a comma-separated list of directories. Note that you have to specify subdirectories individually.

The directories you list here is combined with the directories specified in `#include` statements in order to find the necessary files. For example, if you have `c:\d1\d2\d3\file.h`, you can enter `c:\d1\d2` as the value of this property and then use `d3\file.h` in the `#include` statement.

You should take into account that the value of this property also determines the structure of the source file directory when code is generated from the model. So, in the above example, the top-level directory created is `d3`.

*Default = Blank*

## Undefined

The Undefined property specifies symbols and macros to be undefined by using `#undef`.

*Default = empty string*

# Promotions

The metaclass Promotion contains a number of properties used to specify whether Rational Rhapsody should add various advanced modeling constructs to your model based on relationships/patterns uncovered during reverse engineering.

## EnableAttributeToRelation

The EnableAttributeToRelation property is used to specify whether Rational Rhapsody should add Associations to the model for attributes whose type is another class in the model.

For example, if you have two classes, A and B, and B contains an attribute of type A, Rational Rhapsody adds an Association to the model reflecting this relationship.

*Default = Checked*

## EnableResolveIncompleteClasses

Sometimes, during reverse engineering, Rational Rhapsody is not able to find the base class for a given class. The EnableResolveIncompleteClasses property is used to specify that if Rational Rhapsody finds a class with the same name as the base class in a different location, it should assume that this class is the missing base class.

*Default = Checked*

# JAVA\_Roundtrip

---

The JAVA\_Roundtrip subject contains metaclasses that contain properties that affect roundtripping. Most of the properties are used by all three languages. However, any language-specific properties are clearly labeled.

## General

The General metaclass contains properties that control how changes to code are roundtripped in Rational Rhapsody.

### CancelIfReadOnly

Set this property to Checked in order to cancel roundtripping if there are any read-only (checked-in) units that are related to one of the files to be roundtripped.

By default, roundtripping changes only read/write units and provides warnings in the Output window about any attempted changes to Read Only units.

*Default = Cleared*

### NotifyOnInvalidatedModel

*The NotifyOnInvalidatedModel property is a Boolean value that determines whether a warning window is shown during a round trip. This warning is shown when information might get lost because the model was changed between the last code generation and the round trip operation. (Default = True)*

### ParserErrors

The ParserErrors property specifies the behavior of a round trip when a parser error is encountered. The possible values are as follows:

- Abort - Stop the round trip whenever there is a parser error in the code. No changes is applied to the model.
- AskUser - When Rational Rhapsody encounters an error, the program displays a message asking what you want to do.
- AbortOnCritical - Stop the round trip if any critical parser errors are encountered in the code.
- Ignore - Continue roundtrip, ignoring any parser errors that are encountered.

*Default = AskUser*

### PredefineIncludes

The PredefineIncludes property specifies the predefined include path for roundtripping.

*Default = \$OMROOT\LangJava\src,D:\jdk1.2.2\src*

## PredefineMacros

The PredefineMacros property specifies the predefined macros for roundtripping. The default value is as follows:

```
DECLARE_META(class_0\,animClass_0), DECLARE_REACTIVE_META(class_0\,animClass_0),
OMINIT_SUPERCLASS(class_0Super\,animClass_0Super),
OMREGISTER_CLASS\,DECLARE_META_T(class_0\, ttype\,animClass_0),
DECLARE_REACTIVE_META_T(class_0\, ttype\,animClass_0),
DECLARE_META_SUBCLASS_T(class_0\, ttype\,animClass_0),
DECLARE_REACTIVE_META_SUBCLASS_T(class_0\, ttype\,animClass_0),
DECLARE_MEMORY_ALLOCATOR(CLASSNAME\,INITNUM),
IMPLEMENT_META(class_0\,Default\,FALSE),
IMPLEMENT_META_S(class_0\,FALSE\,class_1\,animClass_1\, animClass_0),
IMPLEMENT_META_M(class_0\, FALSE\, class_0Super\, 2\,animClass_0),
IMPLEMENT_REACTIVE_META(class_0\,Default\,FALSE),
IMPLEMENT_REACTIVE_META_S(class_0\,FALSE\,class_1\, animClass_1\,animClass_0),
IMPLEMENT_REACTIVE_META_M(class_0\, FALSE\, class_0Super\, 2 \,animClass_0),
IMPLEMENT_REACTIVE_META_SIMPLE(class_0\,Default\,FALSE),
IMPLEMENT_REACTIVE_META_S_SIMPLE(class_0\,FALSE\,class_1\ ,animClass_1\,animClass_0),
IMPLEMENT_REACTIVE_META_M_SIMPLE(class_0\, FALSE\, class_0Super\, 2 \,animClass_0),
IMPLEMENT_META_T(class_0\, Default\, FALSE\, animClass_0),
IMPLEMENT_META_S_T(class_0\,FALSE\,class_0Super\,animclas s_0Super\,animClass_0),
IMPLEMENT_META_M_T(class_0\, FALSE\, class_0Super\, 2 \,animClass_0),
IMPLEMENT_META_OBJECT(class_0\,class_type\,Default\, FALSE),
IMPLEMENT_META_S_OBJECT(class_0\,class_type\,FALSE\, class_1\,animClass_1\,animClass_0),
IMPLEMENT_META_M_OBJECT(class_0\,class_type\,FALSE\, class_0Super\, 2 \,animClass_0),
IMPLEMENT_REACTIVE_META_OBJECT(class_0\,class_type\, Default\,FALSE),
IMPLEMENT_REACTIVE_META_S_OBJECT(class_0\,class_type\,
FALSE\,class_1\,animClass_1\,animClass_0),
IMPLEMENT_REACTIVE_META_M_OBJECT(class_0\,class_type\, FALSE\, class_0Super\, 2
\,animClass_0), IMPLEMENT_REACTIVE_META_SIMPLE_OBJECT(class_0\,
class_type\,Default\,FALSE), IMPLEMENT_REACTIVE_META_S_SIMPLE_OBJECT(class_0\,
class_type\,FALSE\,class_1\,animClass_1\,animClass_0),
IMPLEMENT_REACTIVE_META_M_SIMPLE_OBJECT(class_0\, class_type\,FALSE\, class_0Super\,
2 \,animClass_0), IMPLEMENT_META_T_OBJECT(class_0\,class_type\, Default\, FALSE\,
animClass_0), IMPLEMENT_META_S_T_OBJECT(class_0\,class_type\,FALSE\,
class_0Super\,animclass_0Super\,animClass_0),
IMPLEMENT_META_M_T_OBJECT(class_0\,class_type\, FALSE\, class_0Super\, 2 \,animClass_0),
IMPLEMENT_MEMORY_ALLOCATOR(CLASSNAME\,INITNUM\,
INCREMENTNUM\,ISPROTECTED), DECLARE_META_PACKAGE(Default),
DECLARE_PACKAGE(Default), IMPLEMENT_META_PACKAGE(Default\,Default),
DECLARE_META_EVENT(event_0), DECLARE_META_SUBEVENT(event_0\,event_0Super\,
event_0SuperNamespace), IMPLEMENT_META_EVENT(event_0\,Default\,event_0),
IMPLEMENT_META_EVENT_S(words\, words\, baseWords),
DECLARE_OPERATION_CLASS(mangledName), DECLARE_META_OP(mangledName),
OM_OP_UNSER(type\, name), OP_UNSER(func\, name), OP_SET_RET_VAL(retVal),
OM_OP_SET_RET_VAL(retVal), IMPLEMENT_META_OP(animatedClassName\, mangledName\,
opNameStr\, isStatic\, signatureStr\, numOfArgs), IMPLEMENT_OP_CALL(mangledName\,
```

userClassName\, call\, retExp), STATIC\_IMPLEMENT\_OP\_CALL(mangledName\, userClassName\, call\, retExp), OMDefaultThread=0, NULL=0, OMDECLARE\_GUARDED  
OM\_DECLARE\_COMPOSITE\_OFFSET

## ReportChanges

The ReportChanges property defines which changes are reported (and displayed) by the roundtrip operation. The possible values are as follows:

- None - No changes are displayed in the output window.
- AddRemove - Only the elements added to, or removed from, the model are displayed in the output window.
- UpdateFailures - Only unsuccessful changes to the model are displayed in the output window.
- All - All changes to the model are displayed in the output window.

*Default = AddRemove*

## RestrictedMode

The RestrictedMode property is a Boolean value that specifies whether restricted-mode roundtripping is enabled. This property can be modified on the configuration level. Restricted mode of full roundtrip roundtrips unusual usage of Rational Rhapsody elements, such as a class declaration in a user-defined type. Restricted mode has more limitations, but preserves the model from unexpected changes. The additional limitations for restricted mode are as follows:

- User-defined types cannot be removed or changed on roundtrip because Rational Rhapsody code generation adds the Ignore annotation for a user-defined type declaration.
- Relations cannot be removed or changed on roundtrip.
- New classes are not added to the model.

*(Default = False)*

## RoundtripNewFiles

When working in Eclipse platform integration, use this property to set Roundtrip policy for the files that the user added to the Eclipse project and are connected to the active Rational Rhapsody Eclipse Configuration.

Optional values:

- AskUser - A message is displayed and asks the user whether or not to Roundtrip the new file.
- Always - New file is Roundtripped into Rational Rhapsody model with no message.
- Never - New file is never Roundtripped into Rational Rhapsody model with no message.

*Default = AskUser*

## RoundtripScheme

The RoundtripScheme property specifies whether to perform a basic or full roundtrip. Batch and online roundtrips change their behavior according to the specified value.

*Default = Advanced*

## Type

The Type metaclass contains a property that controls whether user-defined types are ignored during the roundtrip operation.

### Ignore

*The Ignore property is a Boolean value that specifies whether to include user-defined types in a roundtrip operation. Types with the Ignore property set to True are generated with an Ignore annotation and does not get changed when a roundtrip is performed. The default value of this property is True, which allows no deletion or change to be done on types. Setting this property to False reflects changes to the types declaration and deletion of types during roundtrip. Modifying the name of an existing type results in the addition of a new type, and removal of the model type (if the AcceptChanges property allows element removal), and the references for the model to the removed type is lost (such as appearance in diagrams, property settings, and so on). You can set this property either on the configuration or on specific elements in the model (which affects the elements and its aggregates). (Default = True)*

## Update

The Update metaclass contains a property that controls the update process used during roundtripping.

### AcceptChanges

The AcceptChanges property is an enumerated type that specifies which changes are applied to each CG element (attribute, operation, type, class, or package). You can apply separate properties to each type of CG element. The possible values are as follows:

- All - All the changes can be applied to the model element.
- Default = 1) Rhapsody does not roundtrip deletions if the updated code results in parser errors. 2) Rhapsody does not roundtrip the deletion of classes.
- NoDelete - All the changes except deletion can be applied to the model element. This setting prevents accidental removal of operations, constructors, attributes, relations, variables, instances, and functions.
- AddOnly - Apply only the addition of an aggregate to the model element. You cannot delete or change elements.
- NoChanges - Do not apply any changes to the model element.

Note that the value of the property is propagated to all the aggregates of an element. Therefore, if a package has the NoChanges value, no elements in that package is changed.

*Default = "Default"*

## **UpdateExternalElements**

Ordinarily, if an element in a model has been defined as an external element (meaning that the UseAsExternal property is set to Checked), Rational Rhapsody does not generate code for the element nor does it roundtrip into the model changes made to the element code.

However, if you set the value of the UpdateExternalElements property to Checked, Rational Rhapsody roundtrips into the model changes made to the relevant external elements.

*Default = Checked*

# Model

---

The Model subject contains metaclasses that contain properties that control prefixes added to attributes, variables, and arguments to reflect their type.

## ARBMT

The ARBMT metaclass contains AUTOSAR properties.

### ARAtomicSoftwareComponentTypeMetaclasses

Identifies the names of the AUTOSAR Atomic Software Component Type metaclass names

Default value: ApplicationSoftwareComponent-Type,ComplexDeviceDriverComponentType,  
EcuAbstractionComponent-Type,SensorActuatorSoftwareComponentType,ServiceComponentType

### ARInternalBehaviorMetaclass

Identifies the names of the AUTOSAR Internal Behavior metaclass name.

Default value: InternalBehavior

### ARRTEEventMetaclasses

Identifies the names of the AUTOSAR RTE Event metaclass name.

Default value: DataReceivedEvent,  
OperationInvokedEvent,DataSendCompletedEvent,DataReceiveErrorEvent,ModeSwitchEvent,  
Mode-SwitchedAckEvent,AsynchronousServerCallReturnsEvent,TimingEvent

### ARRunnableEntityMetaclasses

Identifies the name of the AUTOSAR Runnable Entity metaclass.

Default value: RunnableEntity

### RimbObjectMetaclass

Identifies the name of the RIMB Object's metaclass.

Default value: "RIMBO"



## AR\_BMT

The AR\_BMT metaclass contains the RIMBO properties.

### CreatedARBMTPackageName

Configure the name of the ARBMTPackage generated by the “Create RIMBO” utility.

- SoftwareComponentName: The name of the selected SWC.
- Index: An index for resolving conflicting names.

Default value: `?<IsConditionalProperty>$<SoftwareComponentName>ImplPkg?<begin>$<Index>?<=>0?<?>?<:>$<Index>?<end>`

### CreatedInterfaceName

Configure the name of the RIMB Ports Interfaces generated by the “Create RIMBO” utility.

- SoftwareComponentName: The name of the selected SWC.
- ARBMTPackage: The ARBMTPackage name in which the RIMB will be created.
- RIMBName: The name of the RIMB class owning the created port.

Default value: `?<IsConditionalProperty>?<begin>$<InterfaceType>?<=>RIMBClientInterface?<?>CL?<:>?<begin>$<InterfaceType>?<=>RIMBServerInterface?<?>SV?<:>?<begin>$<InterfaceType>?<=>RIMBSenderInterface?<?>SN?<:>?<begin>$<InterfaceType>?<=>RIMBReceiverInterface?<?>RC?<:>?<end>?<end>?<end>?<end>_?<InterfaceName>`

### CreatedPortName

Configure the name of the RIMB Ports Interfaces generated by the “Create RIMBO” utility

- SoftwareComponentName: The name of the selected SWC.
- ARBMTPackage: The ARBMTPackage name in which the RIMB will be created.
- RIMBName: The name of the RIMB class owning the created port.
- PortName: The name of the created port.
- PortType: The type (metaclass) of the created port.

Default value: `?<IsConditionalProperty>$<PortName>`

### CreatedRIMBName

Configure the name of the RIMB generated by the “Create RIMBO” utility.

- **SoftwareComponentName:** The name of the selected SWC.
- **ARBMTPackage:** The ARBMTPackage name in which the RIMB will be created.

Default value: `?<IsConditionalProperty>$<SoftwareComponentName>Impl`

## CreatedRIMBOName

Configure the name of the RIMBO generated by the “Create RIMBO” utility.

- **SoftwareComponentName:** The name of the selected SWC.
- **ARBMTPackage:** The ARBMTPackage name in which the RIMB will be created.
- **RIMBName:** The name of the RIMB class owning the created port.

Default value: `?<IsConditionalProperty>$<SoftwareComponentName>ImplObj?<begin>$<Index>?<=>0?<?>?<:>$<Index>?<end>`

## CreatedSWCImplDiagramName

Configure the name of the SWCImpl Diagram generated by the “Create RIMBO” utility.

Available tokens: **SoftwareComponentName:** The name of the selected SWC.

Default value: `?<IsConditionalProperty>$<SoftwareComponentName>Impl?<begin>$<Index>?<=>0?<?>?<:>$<Index>?<end>`

## Attribute

The Attribute metaclass contains a property that controls whether extra prefixes are added to attributes, variables, and arguments.

## IsTemplateParameterType

Indicates that the attribute represents a template parameter. This property is used internally by Rational Rhapsody. Under normal circumstances, there is no reason to modify the value of this property.

## Prefix

The Prefix property specifies the prefix added to the model attributes, variables, and arguments, if the `Model::Attribute::UsePrefix` property is set to True.

Note that when Rational Rhapsody generates the code, the accessor and mutators do not include the prefix. For example, if you have an attribute named A, and UsePrefix is set to True, Prefix is set to "m", and PrefixForAttribute is set to "t" and you change the attribute type:

- The accessor and mutator for the attribute are getA and setA.
- The actual name of the attribute is m\_tA.

You can change the name of a variable, attribute, or argument so it does not obey the prefix. In this case, the element remains "unprefixed" until you change its type.

*Default = m\_*

## PrefixForStatic

The PrefixForStatic property specifies the extra prefix added to the model static attributes, if the Model::Attribute::UseTypePrefix property is set to Checked.

Note that when Rational Rhapsody generates the code, the accessor and mutators do not include the prefix. For example, consider an attribute named A. If UsePrefix is set to Checked, Prefix is set to "m", and PrefixForStatic is set to "s" and you change the attribute type:

- The accessor and mutator for the attribute is setA and getA.
- The actual name of the attribute is m\_sA.

You can change the name of a variable, attribute, or argument so it does not obey the prefix. In this case, the element remains "unprefixed" until you change its type.

Note that template attributes do not use prefixes.

*Default = s*

## UsePrefix

The UsePrefix property is a Boolean property that specifies whether prefixes are added to attributes, variables, and arguments to reflect their type. You set this property at the project level.

When this property is set to Checked, the name of the variable, attribute, or argument is updated automatically when you change the type of the variable, attribute, argument by using the Features window. However, the name is not changed automatically when the name of the type itself is changed.

Note the following restrictions:

- Template attributes do not use prefixes.
- Existing models are not automatically changed to obey the specified prefix. You can write a VBA macro to modify the model so it uses the prefixes.

Note that you specify the prefix to be added to the name by setting the Prefix, PrefixForStatic, and Model::Type::PrefixForAttribute properties.

*Default = Cleared*

# Class

The metaclass Class contains property that indicates whether the class represents a template parameter.

## CommonList

The CommonList property controls which elements appear in the top section of the Add New menu (referred to as the common list), when applicable. You can re-order, remove, or re-add any of these elements by doing so through the CommonList property.

Whatever element that is removed from CommonList is displayed in the middle portion of the Add New menu if that element is relevant for your project. The element must appear somewhere if it is a valid element.

*Default =*

*Attribute, PrimitiveOperation, TriggeredOperation, Reception, Constructor, Initializer, Destructor, CleanUp, Event, Function, Variable*

## InstanceReferenceTarget

Rational Rhapsody includes an Instance Reference Browser tool that lets you browse the model in "Instances" view, and create an "InstanceReference" that defines an Instance in the model. The Instance target is the last element in the full path for the instance.

You can define the metaclass or new term on which to filter the instances view. The filter by metaclasses or new terms is determined by the InstanceReferenceTarget property. You define this property by using a comma-separated list of strings that are the names of the metaclasses or valid names of new terms.

The list of strings are translated into a command on the popup menu like so:

Create Instance Reference > [String1] [String2] and so on.

For example: Create Instance Reference > Attribute Object Port

Note that the Create Instance Reference > ... pop-up menu appears only for the element created out of the term that defined this property.

The following list shows the possible metaclasses that can be used as targets:

- AssociationEnd
- Attribute
- CleanUp
- Comment
- Constraint
- ControlledFile

- Dependency
- Derivation
- Flow
- FlowItem
- FlowPort
- Generalization
- HyperLink
- \_INITIALIZER
- Object
- Port
- PrimitiveOperation
- Realization
- Reception
- Requirement
- Tag
- TriggeredOperation
- Type

*Default = Empty string. When no value is entered for this property, the Create Instance Reference command is not available on the popup menu.*

Note that the model needs to include the following stereotypes and terms:

- InstanceReference - Term applicable to Comment
- l\_base - Term applicable to Dependency
- l\_context- Term applicable to Dependency
- l\_target - Term applicable to Dependency

## **IsTemplateParameterType**

Indicates that the class represents a template parameter. This property is used internally by Rational Rhapsody. Under normal circumstances, there is no reason to modify the value of this property.

## **SubstituteTerm**

Indicates that a class is replaced by a block so that an operation, such as exposing an implicit part class or creating a part + class by typing a name of the object : new class name in a diagram, creates an instance of the term instead of a class.

*Default = block*

## Comment

Contains properties to define comments in SysML projects.

### CallOutCompartments

If the IsCallOut property is checked, the displayed comment uses the text in this field as the allocated from and to labels.

*Default = AllocatedFrom,AllocatedTo*

### IsCallOut

If this property is checked, the comment in the drawing becomes a "callout" showing the allocated from/to.

*Default = Cleared*

## ControlledFile

The ControlledFile metaclass allows you to create controlled files and then use their features.

- Controlled Files, such as project specifications files (for example, Word, Excel files) are typically added to a project for reference purposes and can be controlled through Rational Rhapsody.
- A controlled file can be a file of any type (.doc, .txt, .xls, and so on).
- Controlled files are added into the project from the Rational Rhapsody browser.
- Controlled files can be added to diagrams by using drag-and-drop from the browser.
- Currently, only Tag and Dependency features can be added to a controlled file.
- By default all controlled files are opened by their Windows-default programs (for example, Microsoft Excel for .xls files).
- The programs that are associated with controlled files can be changed by way of the Properties tab in the controlled files window.

### DeleteUnderlyingFileWhenDeletingTheElement

The DeleteUnderlyingFileWhenDeletingTheElement property specifies whether Rational Rhapsody should delete the underlying file when a controlled file element is removed from a model. The possible values are:

- Never - the underlying file should not be deleted.

- Always - the underlying file should be deleted.
- AskUser - the user should be asked whether the underlying file should be deleted when the element is removed from the model.

*Default = AskUser*

## FileTypes

The FileTypes property can be used to filter the files shown in the file browsing window that is displayed when the user creates a new controlled file and the Model::ControlledFile::NewPolicy property is set to the Browser value.

*Default = \*.\**

## NewCommand

The NewCommand property specifies the command that should be executed when the user selects the option of adding a new controlled file. This command is executed only if the Model::ControlledFile::NewPolicy property is set to the UseNewCommand value.

Default for UNIX = touch \$name.ext

Default for Windows = \$OMROOT\etc\touch.exe "\$file.ext"

## NewPolicy

The NewPolicy property determines how controlled files are created. The possible values are:

- Browse - when you right-click on an object in the browser and select Add NewControlled File, a standard file browsing window is displayed. The window displays all file types, unless you have modified the Model::ControlledFile::FileTypes property.
- UseNewCommand - when Add NewControlled File is selected from the browser, the command in the NewCommand property is executed. The default value of the command is "touch \$name.ext" for UNIX and "\$OMROOT\etc\touch.exe "\$file.ext"" for Windows. The "\$file" is expanded to a file name for the controlled file. If a stereotype is specified, Rational Rhapsody uses the stereotype as the file name and adds a sequence of numbers to it. If no stereotype is specified, then it uses the name 'Controlled\_File' and adds a sequence number to the end of the file name.

*Default = Browse*

## OpenCommand

The OpenCommand property specifies the command that should be executed when the user opens a controlled file from the Browser. This command is executed only if the Model::ControlledFile::OpenPolicy property is set to the value UseOpenCommand.

*Default = \$fileName*

## OpenFileAfterCreation

If the OpenFileAfterCreation property is set to Checked, Rational Rhapsody opens controlled files immediately after they are created.

Default= Cleared

## OpenPolicy

The OpenPolicy property determines how Rational Rhapsody opens controlled files. The possible values are:

- SystemDefault - opens the file by using the MIME-type mapping for the system.
- UseOpenCommand - uses the command specified in the Model::ControlledFile::OpenCommand property.
- UseRhapsodyCodeEditor - opens the file by using the Rational Rhapsody internal code editor.
- UseRhapsodyCSVFileViewer - opens the file by using the Rational Rhapsody internal CSV file viewer.
- UseRhapsodyTableViewer - opens the file by using the Rational Rhapsody table viewer. This option is only relevant for CSV files.
- AskUser - window is opened, allowing the user to select one of the open methods.

*Default = SystemDefault*

## WaitTimeAfterFileCreation

The WaitTimeAfterFileCreation property determines the amount of time (in seconds) that Rational Rhapsody waits for an external command to be executed to create a new controlled file before control is returned to Rational Rhapsody.

*Default = 1*

## General

Contains general properties related to models.

## ActionLanguage

For Action Language models, the property Model::General::ActionLanguage is included with a value of True. This property indicates to Rational Rhapsody that the model includes action language code, and therefore the content of fields that support action language, such as operation bodies, should be treated as action language code rather than C++ code.



The value of this property should not be modified. Otherwise, Rational Rhapsody will treat the action language code as ordinary C++ code.

## Importer

Contains properties related to importing files such as ARXML files or Franca IDL files.

### DuplicateTopLevelPackageSuffix

When you import files such as ARXML or Franca IDL files, if you select the "Add as top-level package" option, new packages are created for packages that already exist in the model. The name used for the new package consists of the name of the original package plus a suffix. You can customize the suffix by modifying the value of the property DuplicateTopLevelPackageSuffix.

### ImportLogFileName

The property ImportLogFileName can be used to specify the name that should be used for the import log file that is created when you import files such as ARXML or Franca IDL files. The value of this property should be the full path for the file, and it can include the keywords: \$projectDirectory and \$codeGenDirectory.

### IncludeInImportMerge

The IncludeInImportMerge property can be used to specify existing model elements that should not be changed in any way when files such as ARXML files or Franca IDL files are imported into Rhapsody. A number of profiles provided with Rhapsody include a stereotype called ExcludeFromImportMerge which has the property IncludeInImportMerge set to False. So in these profiles, you can apply the ExcludeFromImportMerge stereotype to specific elements to ensure that they are not modified when a file is imported.

*Default = True*

### MergeLogFileName

The property MergeLogFileName can be used to specify the name that should be used for the merge log file that is created when you import files such as ARXML or Franca IDL files. The value of this property should be the full path for the file, and it can include the keywords: \$projectDirectory and \$codeGenDirectory.

### TypesAsSeparateSavedUnit

When files such as ARXML files are imported into Rhapsody, the Advanced tab of the dialog that is displayed allows you to select which types of elements should be saved as separate units in the model.

You can use the property `TypesAsSeparateSavedUnit` to customize the list of element types that is displayed in the dialog. The property value should be a comma-separated list of element types.

*Default = ARPpackage, ECUExtractPackage*

## VerboseImportLog

The property `VerboseImportLog` can be used to set the level of information written to the import log file that is created when you import files such as ARXML or Franca IDL files (the log file defined using the property `Model::Importer::ImportLogFileName`). For a more detailed report, set the property `VerboseImportLog` to `True`.

## VerboseMergeLog

The property `VerboseMergeLog` can be used to set the level of information written to the merge log file that is created when you import files such as ARXML or Franca IDL files (the log file defined using the property `Model::Importer::MergeLogFileName`). For a more detailed report, set the property `VerboseMergeLog` to `True`.

# MatrixLayout

The `MatrixLayout` metaclass contains properties that you can use for the design of matrix layouts.

## ShowContainerElementForPorts

This property instructs Rational Rhapsody to look at Ports as well as its container elements when displaying From/To information of Links and Flows.

*Default = Checked*

# MatrixView

The `MatrixView` metaclass contains properties that you can use for the appearance of matrix views.

## ChangeSelectionOnEnterTo

The `ChangeSelectionOnEnterTo` property can be used to control what cell in the table or matrix receives the focus after you press the Enter key. You can use this property to specify that the focus should move to the cell to the right of the current cell, the cell to the left of the current cell, the cell below the current cell, or the cell above the current cell.

When you press the combination of Shift+Enter, the focus will always move in the direction opposite the direction specified with this property. For example, if ChangeSelectionOnEnterTo is set to "right", pressing Shift+Enter will move the focus to the cell to the left of the current cell.

*Default = down*

## HideCellNames

This property show or hides Rational Rhapsody element names in matrix view cells. Select this option if you want to hide element names (an icon displays to indicate the element type).

*Default = Cleared*

## HideEmptyRowsCols

This property shows or hides rows and columns that are not holding any element data. This property also reflects the last selected mode set in the MatrixView toolbar.

*Default = Cleared*

## RowHeaderSize

You can use the RowHeaderSize property to limit the width of the row headers in matrices.

This option can be useful in situations where lengthy text in the row headers results in a small viewing area for the other columns in the matrix.

If you limit the width of the headers with this property, the contained text is truncated if it exceeds the width that you specified, but you can see the entire text in a tooltip by holding your mouse over the relevant header.

The value of this property should be the number of pixels you want to use for the row headers.

If the value of the property is left blank, the width of the row headers is determined by the length of the longest text that must be included in one of the row headers.

*Default = Blank*

## SwitchRowsCols

For matrix views, you can set the property SwitchRowsCols to True in order to have the row data displayed as columns and vice versa.

Note that this is similar to the Switch Rows and Columns option in the Table/Matrix drawing toolbar, however the GUI option only applies until you close the matrix view. The property allows you to save this setting for specific matrix views.

*Default = False*

## Package

The Package metaclass contains properties that specify settings for packages on the Rational Rhapsody Design Manager.

### DMPProjectArea

The DMPProjectArea property specifies the project area on the Rational Design Manager server that this package belongs to.

### OwnerModelOnDmServer

Rhapsody's "external unit" feature allows multiple projects to have write-access to a unit that is located outside the project directory. In Rhapsody Design Manager, however, any unit must belong to a single project, and other projects can then include a read-only reference to the unit. When a model that is being moved to a DM server contains "external" units, the property OwnerModelOnDmServer should be used to specify the name of the model that will serve as the owner of the unit on the server.

*Default = Blank*

### TrackChanges

When you are using the Track Changes feature, you can specify that changes should not be tracked for a specific package by setting the value of the property TrackChanges to False for that package.

*Default = True*

## Profile

The Profile metaclass contains properties that specify the behavior of profiles.

### AdditionalHelpersFiles

The AdditionalHelpersFiles property can be used to specify additional .hep files that should be associated with a profile, beyond the .hep file associated with the profile because it shares the same name as the profile.

The value of this property should be a comma-separated list of the additional .hep files you would like to associate with the profile.

*Default = Blank*

## **AnimateSDLBlockBehavior**

By default, in animated sequence diagrams, SDLBlocks are considered to be black boxes. Internal events within the block are not displayed. If you would like the animated diagrams to display these internal SDL events, set the value of this property to True.

This property is set at the profile level.

*Default = Cleared*

## **DomainDefinition**

The URI to use for the newly-created domain.

Enter a string value that uses this format: `http://.../MyProfile#`

*Default = Blank*

## **DomainLabel**

The name that will be displayed for the domain in previews.

*Default = Blank*

## **DomainTitle**

The name that will be displayed for the domain on the properties page.

*Default = Blank*

## **DomainVersion**

The version of the domain.

*Default = 1.0*

## **GloballyApplied**

The GloballyApplied property makes it possible to selectively apply a profile to elements in your project. When this property is set to Checked, the profile is applied to all elements in your model. When set to Cleared, the profile applies only where you have drawn an AppliedProfile dependency between the element and the profile. For example, you can draw this type of dependency to specify that the profile only be applied to a specific component.

You can use this method to apply a profile to the following elements: package, component, project.

Note that creating an `AppliedProfile` dependency for a project has the same effect as setting the value of the property to `Checked`. However, this approach might be required in cases where a profile is added by reference and therefore you cannot modify the value of the property directly.

## IgnoreNewTermOwnershipRestriction

If this property is cleared, Rational Rhapsody takes into account the `Model::Stereotype::Owners` and `Model::Stereotype::Aggregates` property when the user is trying to move an element from one place to another. If these properties definitions are violated, Rational Rhapsody does not allow the user to move the element. This property is applied for New Term stereotypes only.

*Default = Cleared*

## OriginalFileslocations

When you create a Design Manager domain from a profile, Rhapsody uses the property `OriginalFileslocations` to store the original path of the profile. When users use the domain afterwards, Rhapsody uses this information in order to ensure that all components of the profile, such as helper applications, function correctly.

*Default = Blank*

## PropertyFile

The `PropertyFile` property allows you to specify an additional `.prp` file that should be associated with the profile. Enter the path to the relevant `.prp` file.

Note that in terms of priority, if the file specified here has a property with the same name as a property specified at the factory, site, or profile level, the possible values and default value in this file take precedence.

Default = Blank

## RDFNamespace

The `RDFNamespace` property defines the URI prefix for each resource inside a profile (each URI resource defines an element inside a profile). Enter a string value with a namespace that uses this format: `http://name/.../space#`.

If this property is empty, for UML, "`http://jazz.net/ns/dm/rhapsody/uml#`" is used as the namespace. For SysML, "`http://jazz.net/ns/dm/rhapsody/sysml#`" is used as the namespace.

Example for SysML: `http://jazz.net/ns/dm/rhapsody/sysml#Block`, where "`http://jazz.net/ns/dm/rhapsody/sysml#`" is the namespace and "`Block`" is the new term name.

*Default = Empty string*

## **SDLSignalPrefix**

The naming convention used for the Rational Rhapsody events that represent SDL signals is to add "\_" as a prefix to the original signal name. The SDLSignalPrefix property allows you to change this prefix.

This property is set at the profile level.

*Default = \_*

## **UpdateDomain**

The UpdateDomain property determines the criterion used for automatic updates of the domain.

Note that automatic updates are only carried out by the import engine. This feature does not apply to any other method of moving information to the server.

The possible values are:

- Never - The domain on the server is not updated automatically even if the profile has changed.
- OnChange - The domain on the server is updated if it is determined that the profile has been changed.
- OnVersionChange - The domain on the server is updated if the version number has been changed.

*Default = Never*

## **UseRapidPorts**

By default, SDLBlocks use behavioral ports. The UseRapidPorts property can be used to change this behavior. When set to True, rapid ports is used instead.

This property is set at the profile level.

*Default = Cleared*

## **Query**

Contains properties that relate to user-defined queries.

## **ShowInBrowserFilterList**

Use the property ShowInBrowserFilterList to specify whether or not a specific query should be included as a possible filter in the browser filter list.

When you select a query from the browser filter list, the browser displays only the elements that satisfy the criteria specified in the query.

*Default = True*

## Relation

The Relation metaclass contains properties that specify the behavior of associations.

### Prefix

The Prefix property specifies the prefix to be added to associations if the Model::Relation::UsePrefix property is set to Checked.

Note that when Rational Rhapsody generates the code, the accessors and mutators do not include the prefix. For example, consider an association named A. If the UsePrefix property is set to Checked and the Prefix property is set to "m":

- The accessor and mutator for the association is setA and getA.
- The actual name of the association is mA..

*Default = m\_*

### UsePrefix

The UsePrefix property is a Boolean property that specifies whether prefixes are added to associations.

When this property is set to Checked, the name of the association is updated automatically when you change the type of the association using the Features window. However, the name is not changed automatically when the name of the type itself is changed.

Note the following restrictions:

- Template associations do not use prefixes.
- Existing models are not automatically changed to obey the specified prefix. However, you can write a VBA macro to modify the model so it uses the prefixes.

Note that you specify the prefix to be added to the name by setting the Model::Relation::Prefix property.

*Default = Cleared*

## Stereotype



The Stereotype metaclass includes properties that relate to the use of stereotypes in general, and to the use of "new term" stereotypes in particular.

## ActiveInferredStereotypeName

Allocations that are created automatically have the stereotype "inferred" applied to them. If you want to use a different stereotype to identify automatically-created allocations: 1) Create a stereotype that inherits from the "inferred" stereotype 2) Set the value of the property ActiveInferredStereotypeName to the name of the new stereotype that you created.

*Default = inferred*

## AddToDiagramsToolbar

The property enables a newTerm diagram icon to be added to the Diagrams toolbar (without the need to specify it in the General::Model::DiagramsToolbar property). This property applies to newTerm diagrams.

For example, create a stereotype and set the following in the Features window for the stereotype:

- On the General tab: Set the "Applicable to" field to a diagram type, and select the "New Term" check box.
- On the Properties tab, set the AddToDiagramsToolbar property to Checked.

Close and open your project again, and notice that there is another diagram icon button on the Diagrams toolbar.

*Default = Cleared*

## Aggregates

When you create a "new term" stereotype, the Aggregates property is used to specify what types of elements can be added to this type of element. This is the list of elements that is included in the Add New context menu for elements of this type.

If the value of this property consists of more than one element, the element names should be separated by commas.

If the property is left blank, then the aggregates of the base element are used.

For a list of the strings to use to represent the different elements, see the metaclasses.txt file in <Rational Rhapsody installation path>\Doc.

*Default = Blank*

## AllowChangeToApplicableTo

Set the AllowChangeToApplicableTo property to Cleared if you want to prevent other model elements

from being changed to this "new term" stereotype.

*Default = Checked*

## **AllowChangeToNewTerm**

Set the AllowChangeToNewTerm property to Cleared if you want to prevent this "new term" stereotype from being changed to any other term.

*Default = Checked*

## **AllowedTypes**

Ordinarily, when you create an object, you can select the class on which it should be based from all of the available classes in the model. However, when you define a "new term" stereotype that is applicable to Objects, you can use the AllowedTypes property to limit the classes on which such objects can be based.

When you create an object of this "new term" type, Rational Rhapsody only allows you to base it on one of the classes that is listed in the value of the AllowedTypes property for the relevant "new term" stereotype.

*Default = Blank*

## **AllowInferredAllocateForMetaclasses**

The property AllowInferredAllocateForMetaclasses is used to specify the types of elements that should have allocations automatically created when they are added to a swimlane. The content of the property should be a comma-separated list of element types, using the strings provided in the metaclasses.txt file in <Rational Rhapsody installation path>\Doc, or the names of "new term" stereotypes.

When "Swimlane" is included in the value of the property, an allocation is also created between the swimlane itself and the model element that it represents.

*Default = Swimlane,Action*

## **AlternativeDrawingTool**

In certain cases, a number of different out-of-the-box drawing elements are based on the same metaclass, for example, both Class and Composite Class are based on a metaclass called Class. So, when you add a custom diagram element by using a "new term" stereotype, then in addition to specifying the base metaclass in the "Applicable to" field, you have to provide the name of the base element in the value of the AlternativeDrawingTool property.

This property does not have to be used if you are basing your new element on the "default" element of the metaclass.

This information is used for situations where the ambiguity has to be removed, such as determining what

icon is used to represent the "new term" on a drawing toolbar, if the user has not specified a custom icon.

*Default = Blank*

## **BrowseHierarchyMenuName**

Wherever you use the Hierarchy Browser feature (for example, in AUTOSAR projects), you can use the property BrowseHierarchyMenuName to customize the text that appears for the option in the context menu.

Note that this property has an effect only at the level where you modified the value. So if you set the value for a specific package, you will see the custom text only in the context menu for that package.

The value of the property is also used in the drop-down menu inside the hierarchy browser itself.

*Default = Blank*

## **BrowserGroupIcon**

When you define a "new term" stereotype, you can use the BrowserGroupIcon property to specify an icon that should be used in the browser to represent this category of elements.

Provide the full path to the icon file (.ico).

When entering the path, the extension ".ico" is optional.

*Default = Blank*

## **BrowserIcon**

When you define a "new term" stereotype, you can use the BrowserIcon property to specify an icon that should be used in the browser to represent individual elements of that type.

Provide the full path to the icon file (.ico).

When entering the path, the extension ".ico" is optional.

*Default = Blank*

## **CommentNotation**

The CommentNotation property determines how Comment elements are displayed. This property can be set to one of two styles:

- Note\_Style
- Box\_Style

If CommentNotation is set to Note\_Style, then the appearance of the element is determined by the value selected for the property Comment::ShowForm: Note, Plain, or PushPin.

If CommentNotation is set to Box\_Style, then the element resembles a class-box, and you can control the display of compartments for the element.

*Default = Note\_Style*

## Compartments

The Compartments property determines which of the available compartments are displayed by default for the various types of elements. The value for this property is a comma-delimited string containing the names of the compartments that should be visible. Since the available compartments vary from element to element, it is recommended that you do not try to set the value of this property by using the Properties window or directly in the .prp file. Rather, you should use an element Display Options to set which compartments are visible, and then use the MakeDefault option to apply these settings at the diagram or project level for new elements of this type.

*Default = Empty MultiLine*

## ConstraintNotation

The ConstraintNotation property determines how Constraint elements are displayed. This property can be set to one of two styles:

- Note\_Style
- Box\_Style

If ConstraintNotation is set to Note\_Style, then the appearance of the element is determined by the value selected for the property Constraint::ShowForm: Note, Plain, or PushPin.

If ConstraintNotation is set to Box\_Style, then the element resembles a class-box, and you can control the display of compartments for the element.

*Default = Note\_Style*

## CustomHelpBookName

The CustomHelpBookName property is used in conjunction with the CustomHelpMapFile and CustomHelpURL properties to provide Rational Rhapsody with the necessary parameters for displaying profile-specific context-sensitive help if you have prepared such help text for your profile.

*Default = Blank*

## CustomHelpMapFile

The CustomHelpMapFile property is used in conjunction with the CustomHelpBookName and CustomHelpURL properties to provide Rational Rhapsody with the necessary parameters for displaying

profile-specific context-sensitive help if you have prepared such help text for your profile.

The value of this property should be the URL of the map file, for example, \\share\dodaf\_help\dodaf\_help.map. You can include environment variables in the URL, for example, \$DODAF\_HLP\_ROOT\dodaf\_help.map.

*Default = Blank*

## CustomHelpURL

The CustomHelpURL property is used in conjunction with the CustomHelpBookName and CustomHelpMapFile properties to provide Rational Rhapsody with the necessary parameters for displaying profile-specific context-sensitive help if you have prepared such help text for your profile.

The value of this property should be the URL of the help file, for example, \\share\dodaf\_help\main\_dodaf\_help.html. You can include environment variables in the URL, for example, \$DODAF\_HLP\_ROOT\main\_dodaf\_help.html.

*Default = Blank*

## DefaultName

When you add a new model element, it is assigned a default name based on the type of element, for example, class\_0 or class\_1. The same type of generic approach is used when you create elements based on a new term. So if your model contains a "new term" stereotype called SystemConfiguration, when you add new elements of this type, they will have names such as systemconfiguration\_0 or systemconfiguration\_1.

For "new terms", you have an option to change this default naming scheme. You can provide a value for the property DefaultName (at the level of the relevant stereotype) to provide a different string that should be used as the basis for the default name when you create an element of that type. So, you can call your "new term" stereotype SystemConfiguration but then assign a value such as SC to the DefaultName property for that stereotype. Then, each new element of that type that you create will have a default name such as sc\_0 or sc\_1.

*Default = Blank*

## DependentPairMetaclassName

For model synchronization, the DependentPairMetaclassName property sets the type of the dependent pair that the Pair Matching Wizard creates for paired elements. For example, in UML/SysML to AUTOSAR domains it is DependentPairUML2AR.

This property is relevant for DependentPairSet.

*Default = DependentPairSysMLToAR*

## DependentPairSourceFilters

For model synchronization, the `DependentPairSourceFilters` property defines the steps for the Pair Matching Wizard.

The value for this property is a comma-separated list of tokens. Each token represents a step in the Pair Matching Wizard. For each token you must define two more properties:

- `Model::Stereotype::<token name>Name` defines the title of the wizard for this step.
- `Model::Stereotype::<token name>Pattern` defines the pattern that defines the content of the source domain tree. The content of the source domain tree for the current step is accumulated from the content of the previous step. See the help topic about context patterns for their definitions.

Example: `Model::Stereotype::DependentPairSourceFilters` with a value of `SourceFilterPackages,SourceFilterInterfaces`

In this example, you must also define the following properties:

- `Model::Stereotype::SourceFilterPackagesName`
- `Model::Stereotype::SourceFilterPackagesPattern`
- `Model::Stereotype::SourceFilterInterfacesName`
- `Model::Stereotype::SourceFilterInterfacesPattern`

This property is relevant for `DependentPairSet`.

*Default = SourceFilterPackages, SourceFilterInterfaces, SourceFilterBlocks, SourceFilterParts, SourceFilterPorts, SourceFilterItems, SourceFilterTags, SourceFilterDependencies*

## DiagramDisplayString

When creating a new term element, the name of the element that is displayed in the diagram is the name of the stereotype. If for some reason you want a different name to be displayed, you can specify the new name by using this property.

## DrawingShape

When you define a "new term", you can use the `DrawingShape` property to customize the way the element is displayed when it is added to a diagram. This property can take any of the following values:

- `Default` - the appearance of the "new term" is the same as that of the element on which it is based
- `BasicBox` - the "new term" element is displayed as a rectangular box
- `RoundedBox` - the "new term" element is displayed as a rectangular box with rounded edges

*Default = "Default"*

## DrawingToolbar

If you have defined a custom diagram by using a "new term" stereotype, you can use the `DrawingToolbar` property to provide a comma-separated list of elements that should be included in the drawing toolbar for that type of diagram, for example, `RpyDefault,Firewire`. (`RpyDefault` represents all the elements included

in the drawing toolbar of the base diagram.)

The list can include elements supported by the base diagram, and any "new terms" based on these elements.

The order of appearance in the toolbar reflects the order specified in this property.

If no value is entered, the tools from the base diagram are opened.

*Default = Empty string*

## DrawingToolIcon

When you define a "new term" stereotype to create a new type of diagram or diagram element, you can use the DrawingToolIcon property to specify a custom icon that should be used to represent the diagram on the diagram toolbar, or an element on the drawing toolbar.

If you are adding a new type of diagram, this is the icon that is displayed in the Rational Rhapsody Diagram toolbar.

If you are adding a new type of diagram element, this is the icon that is displayed in the drawing toolbar for the diagram on which it displays. (Keep in mind that the icon only displays in the toolbar if the corresponding element is included in the list of elements specified for the DrawingToolbar property for that diagram.)

If no value is entered for this property, Rational Rhapsody uses the icon that was specified for display in the browser, using the BrowserIcon property. If no value was entered for the BrowserIcon property, the icon for the base element is used.

*Default = Blank*

## DrawingToolLabel

Use this property when you define a stereotype that you want to show in the drawing toolbar for a specific diagram. The property lets you define a label that is shown next to the icon that you defined for a new element using the DrawingToolIcon property (for both diagrams and diagram elements). if no label is defined, then the stereotype is used by default.

*Default = Empty string*

## DrawingToolTip

When you define a "new term" that is to appear in a drawing toolbar, you can use the DrawingToolTip property to provide a tooltip that is opened when you mouse over the icon that was defined for the new element by using the DrawingToolIcon property (for both diagrams and diagram elements).

*Default = Empty string*

## HideName

On diagrams, if a model element has one or more stereotypes applied to it, the stereotypes are displayed above the element name. The display of stereotypes is controlled by the "Show Stereotype Label" display option.

In cases where your diagram contains many elements based on "new term" stereotypes, you may prefer to not have the "new term" stereotype name displayed above the element name.

If you set the value of the HideName property to True, ordinary stereotypes will be displayed on model elements in a diagram, but "new term" stereotypes will not be displayed.

*Default = False*

## HidePopupMenuEntries

The property HidePopupMenuEntries can be used to hide specific items in the pop-up menus. This includes the pop-up menus displayed in the browser, in diagrams, in tables, and in the search results window.

The value of the property should be a comma-separated list of the menu items that should be hidden.

The entries can be individual menu items or full sub-menus. To hide individual items from a sub-menu, use "/" as a separator, for example "Ports/New Port"

The property can be set for any element at any level of the model.

For translated menu items, the text for each entry in the property must match the translated menu item.

*Default = Blank*

## HideTabsInFeaturesDialog

When you create new types of model elements by using the "new term" stereotype mechanism, the Features window for such elements contain the tabs that are included in the Features window for the element on which the "new term" is based. The HideTabsInFeaturesDialog property gives you the option to hide some of these tabs.

To use this feature, enter for the value of this property a comma-separated list of the tabs that you would like to hide, for example:

Description,Relations,Tags

*Default = Blank*

## InferredRelationsPolicy



The property `InferredRelationsPolicy` is used to enable the automatic creation of allocations. The property has the following possible values.

- Disabled
- `AutoUpdateInferredRelations`—allocations are created automatically for elements in swimlanes, and the properties of such "inferred" allocation are updated if you make swimlane-related changes for elements, for example, moving an element from one swimlane to another.
- `ManualUpdateInferredRelations`—allocations are not created automatically, nor are existing inferred allocations modified if you make swimlane-related changes, but you can use the "Refresh inferred" menu option to manually trigger the creation of allocations and the updating of existing inferred allocations.

*Default = AutoUpdateInferredRelations*

## InitialLayoutForTables

This property binds a (table or matrix) layout and view through the use of a New Term stereotype that is applicable to table or matrix view elements. If a stereotype is applicable to a table or matrix view, you can set the `InitialLayoutForTables` property with an existing table or matrix layout name to bind a table or matrix view to its corresponding layout.

Note that such a stereotype must reside within a profile for this property to work.

*Default = Blank*

## InitialMatrixFromScope

When you create a new `MatrixView`, if you do not specify a "from scope" and "to scope", it is assumed that the scope is the whole project. This can be a problem in very large models. With such a broad scope, you may encounter serious performance problems whenever you create and then open a new `MatrixView` without specifying a scope first.

To overcome this problem, you can create a "new term" based on `MatrixView`, and set the value of the properties `InitialMatrixFromScope` and `InitialMatrixToScope` to `None` (rather than `Project`) for the new term. You can then use this new term in your model instead of creating ordinary `MatrixViews`.

*Default = None*

## InitialMatrixToScope

When you create a new `MatrixView`, if you do not specify a "from scope" and "to scope", it is assumed that the scope is the whole project. This can be a problem in very large models. With such a broad scope, you may encounter serious performance problems whenever you create and then open a new `MatrixView` without specifying a scope first.

To overcome this problem, you can create a "new term" based on `MatrixView`, and set the value of the properties `InitialMatrixFromScope` and `InitialMatrixToScope` to `None` (rather than `Project`) for the new term. You can then use this new term in your model instead of creating ordinary `MatrixViews`.

*Default = None*

## InitialTableScope

The value for the InitialTableScope property sets the initial table view scope when the scope is left empty in the Features window. If this property is also empty, the default is the entire project.

*Default = Project*

## ModelTooltipTemplateHTML

Enhanced tooltips include detailed information for the model element, displayed in an HTML table.

The content included in the enhanced tooltip can be customized by modifying the value of the property ModelTooltipTemplateHTML.

Note that the brackets used in the template indicate information that is displayed only if relevant for that type of element.

The following keywords can be included in the value of the property

- \$Classifier - the Type of the element, for elements such as Objects
- \$Contract - the contract details of a port
- \$Dependencies - the dependencies defined for the element
- \$Description - the description defined for the element
- \$DiagramPreviewBig - a large preview of the selected diagram
- \$DiagramPreviewSmall - a small preview of the selected diagram
- \$MainDiagram - the diagram that was specified as the "main diagram" for a class or object
- \$NewTermDescription - for elements based on "new terms", the description defined for the new term
- \$ObjectName - the name of the element
- \$OperationPrototype - for operations, the full signature of the operation
- \$Properties - list of the locally overridden properties for the element
- \$PropertiesLink - hyperlink to open the Properties tab of the Feature dialog for the element
- \$Relations - the element's relations (the list of relations that you see on the Relations tab of the Features dialog when you select the "All" option)
- \$Specification - for requirements, the value of the specification field
- \$Stereotypes - a list of the stereotypes applied to the element
- \$TagBaseDescription - for locally-defined tags that have the same name as a base tag, the description defined for the base tag
- \$TagValue - for tag elements, the value of the tag
- \$Tags - the tags defined for the element, along with their values
- \$Type - the metaclass of the element (or the relevant "new term")

## Name

When you define a "new term", the Name property can be used to specify the name that should be used to represent this type of element in the Add New context menu.

If the value of the property is left blank, Rational Rhapsody uses the name that was given to the "new term" stereotype in the Features window.

*Default = Blank*

## Owners

When you create a "new term" stereotype, the Owners property is used to specify the types of elements to which this type of element can be added.

If the value of this property consists of more than one element, the element names should be separated by commas.

If the value of the property is left blank, then the owners of the base element are used.

*Default = Blank*

## PartMetaclassName

This property tells Rhapsody which metaclass to auto apply when creating an Object or a Part of a Class which is a new term stereotype of a class. In some profiles there are stereotypes which acts like pairs, and where one of them is based on a 'class', and the other is based on an 'Object/Part'. When you create an object/part of a 'class' based stereotype, the object/part which is created must automatically apply the Object based stereotype.

For example, in the SysML profile, there are the following stereotypes:

- ConstraintBlock - a new term based on a 'Class'
- ConstraintProperty - a new term based on an 'Object'

When you want to add a new Object/Part to the 'ConstraintBlock', the newly created Object/Part must have the 'ConstraintProperty' stereotype applied to it. Therefore, the value of the above property on the 'ConstraintBlock' stereotype will be 'ConstraintProperty'.

*Default = Blank*

## PluralName

When you define a "new term", the PluralName property can be used to specify the string that should be used to represent the plural of this type of element in the browser.

If the value of the property is left blank, Rational Rhapsody just adds an "s" to the name used for

individual elements of this type.

*Default = Blank*

## PluginMethodToRunOnApply

The property `PluginMethodToRunOnApply` has been superseded by the property `PluginMethodToRunOnApplyWithGUID`. Use `PluginMethodToRunOnApplyWithGUID` in your model instead.

## PluginMethodToRunOnApplyWithGUID

The property `PluginMethodToRunOnApplyWithGUID` can be used to specify a plugin method that should be run anytime that the stereotype is applied to a model element, or anytime an element of this type is created in the model if the stereotype is a "new term".

The value of the property `PluginMethodToRunOnApplyWithGUID` should be the name of a method that takes a String argument. When the method is invoked by Rhapsody, the GUID of the relevant stereotype is provided as the argument. Note that since this method receives the GUID of the stereotype as an argument, it is possible to check what stereotype was applied (or other properties of the stereotype) by calling the method `IRPProject.findElementByGUID` to return the stereotype itself.

For this mechanism to work, the plugin must already be loaded into Rational Rhapsody, and the method must be contained in the plugin's "main" class. (Rational Rhapsody cycles through each loaded plugin until it finds a plugin whose main class contains a method with the name specified.)

## PropertyFile

When you create a "new term" stereotype, you can use the `PropertyFile` property to specify a property file (.prp) that should be added to `factory.prp` for any project that contains this stereotype.

*Default = Blank*

## RDFContainedBy

The Create Domain from Profile wizard in the Rhapsody DM client handles the creation of domains from Rhapsody profiles. A number of properties are provided to allow you to further customize the stereotypes that will be contained in the newly-created domain:

- `RDFContainedBy` - can be used to specify what type of elements can contain elements of this type
- `RDFContainers` - can be used to specify what type of elements can be contained in elements of this type (use a comma-separated list)
- `RDFOWLClassLabel` - the label that will be displayed for the element in the DM web user interface
- `RDFOWLClassName` - the name of the element in the domain

*Default = Blank*

## RDFContainers

The Create Domain from Profile wizard in the Rhapsody DM client handles the creation of domains from Rhapsody profiles. A number of properties are provided to allow you to further customize the stereotypes that will be contained in the newly-created domain:

- **RDFContainedBy** - can be used to specify what type of elements can contain elements of this type
- **RDFContainers** - can be used to specify what type of elements can be contained in elements of this type (use a comma-separated list)
- **RDFOWLClassLabel** - the label that will be displayed for the element in the DM web user interface
- **RDFOWLClassName** - the name of the element in the domain

*Default = Blank*

## RDFOWLClassLabel

The Create Domain from Profile wizard in the Rhapsody DM client handles the creation of domains from Rhapsody profiles. A number of properties are provided to allow you to further customize the stereotypes that will be contained in the newly-created domain:

- **RDFContainedBy** - can be used to specify what type of elements can contain elements of this type
- **RDFContainers** - can be used to specify what type of elements can be contained in elements of this type (use a comma-separated list)
- **RDFOWLClassLabel** - the label that will be displayed for the element in the DM web user interface
- **RDFOWLClassName** - the name of the element in the domain

*Default = Blank*

## RDFOWLClassName

The Create Domain from Profile wizard in the Rhapsody DM client handles the creation of domains from Rhapsody profiles. A number of properties are provided to allow you to further customize the stereotypes that will be contained in the newly-created domain:

- **RDFContainedBy** - can be used to specify what type of elements can contain elements of this type
- **RDFContainers** - can be used to specify what type of elements can be contained in elements of this type (use a comma-separated list)
- **RDFOWLClassLabel** - the label that will be displayed for the element in the DM web user interface
- **RDFOWLClassName** - the name of the element in the domain

*Default = Blank*

## RequirementNotation

The RequirementNotation property determines how Requirement elements are displayed. This property can be set to one of two styles:

- Note\_Style
- Box\_Style

If RequirementNotation is set to Note\_Style, then the appearance of the element is determined by the value selected for the property Requirement::ShowForm: Note, Plain, or PushPin.

If RequirementNotation is set to Box\_Style, then the element resembles a class-box, and you can control the display of compartments for the element.

*Default = Note\_Style*

## rootSourceContextPattern

The rootSourceContextPattern property specifies the content of the source tree to select from.

This property is relevant for DependentPairSet.

*Default = Package\**

## rootTargetContextPattern

The rootTargetContextPattern property specifies the content of the target tree to select from.

This property is relevant for DependentPairSet.

*Default = ARPackage\**

## ShowAnnotationContents

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the text displayed on the element in the diagram determined by the value that was set for the property ShowAnnotationContents. The property can take one of the following values:

- Name - the name of the element
- Label - the label assigned to the element
- Description - the content of the Description field
- Specification (for Requirements only) - the content of the Specification field

*Default = Description*

## ShowAttributes

The ShowAttributes property is a Boolean property that determines whether attributes is opened. When a stereotype is applied to an element that contains attributes, such as a class, the value of this property overrides the value of ShowAttributes at the element level. This property does not override the element-level property if the user has overridden the element-level property.

*Default = Explicit*

## **ShowInheritedAttributes**

This property filters the list of inherited attributes on the Attributes tab of the Features window. When this property is cleared, the list of inheritance is shown only for the selected object, class, or stereotype. When this property is checked, the list includes inheritance from the base object, class, or stereotype.

*Default = Cleared*

## **ShowInheritedOperations**

This property filters the list of inherited operations on the Operations tab of the Features window. When this property is cleared, the list of inheritance is shown for the selected stereotype only. When this property is checked, the list includes inheritance from the base stereotype.

*Default = Cleared*

## **ShowOperations**

The ShowOperations property is a Boolean property that determines whether operations is opened. When a stereotype is applied to an element that contains operations, such as a class, the value of this property overrides the value of ShowOperations at the element level. This property does not override the element-level property if the user has overridden the element-level property.

*Default = Explicit*

## **SourceFilterBlocksName**

For the Pair Matching Wizard, the SourceFilterBlocksName property defines the title of the wizard for this step. Use this property together with the Model::Stereotype::SourceFilterBlocksPattern property. Use these SourceFilter\* properties with the Model::Stereotype::DependentPairSourceFilters property.

*Default = Match Blocks*

## **SourceFilterBlocksPattern**

For the Pair Matching Wizard, the SourceFilterBlocksPattern property defines the pattern that defines the content of the source domain tree. The content of the source domain tree for the current step is accumulated from the content of the previous step. See the help topic about context patterns for their definitions. Use this property together with the SourceFilterBlocksName property. Use these SourceFilter\* properties with the Model::Stereotype::DependentPairSourceFilters property.

*Default = Package\*, Block / Class*

## SourceFilterDefaultName

For the Pair Matching Wizard, the SourceFilterDefaultName property defines the title of the wizard for this step. Use this property together with the SourceFilterDefaultPattern property. Use these SourceFilter\* properties with the Model::Stereotype::DependentPairSourceFilters property.

*Default = Default*

## SourceFilterDefaultPattern

For the Pair Matching Wizard, the SourceFilterDefaultPattern property defines the pattern that defines the content of the source domain tree. The content of the source domain tree for the current step is accumulated from the content of the previous step. See the help topic about context patterns for their definitions. Use this property together with the SourceFilterDefaultName property. Use these SourceFilter\* properties with the Model::Stereotype::DependentPairSourceFilters property.

*Default = ModelElement\*,type:ModelElement\*/ModelElement\**

## SourceFilterDependenciesName

For the Pair Matching Wizard, the SourceFilterDependenciesName property defines the title of the wizard for this step. Use this property together with the SourceFilterDependenciesPattern property. Use these SourceFilter\* properties with the Model::Stereotype::DependentPairSourceFilters property.

*Default = Match Dependencies*

## SourceFilterDependenciesPattern

For the Pair Matching Wizard, the SourceFilterDependenciesPattern property defines the pattern that defines the content of the source domain tree. The content of the source domain tree for the current step is accumulated from the content of the previous step. See the help topic about context patterns for their definitions. Use this property together with the SourceFilterDependenciesName property. Use these SourceFilter\* properties with the Model::Stereotype::DependentPairSourceFilters property.

*Default = Package\*, ModelElement\* / type:ModelElement\*, Dependency / Link*

## SourceFilterInterfacesName

For the Pair Matching Wizard, the SourceFilterInterfacesName property defines the title of the wizard for this step. Use this property together with the SourceFilterInterfacesPattern property. Use these SourceFilter\* properties with the Model::Stereotype::DependentPairSourceFilters property.

*Default = Match Interfaces*

## SourceFilterInterfacesPattern



For the Pair Matching Wizard, the `SourceFilterInterfacesPattern` property defines the pattern that defines the content of the source domain tree. The content of the source domain tree for the current step is accumulated from the content of the previous step. See the help topic about context patterns for their definitions. Use this property together with the `SourceFilterInterfacesName` property. Use these `SourceFilter*` properties with the `Model::Stereotype::DependentPairSourceFilters` property.

*Default = Package\*, Interface*

## SourceFilterItemsName

For the Pair Matching Wizard, the `SourceFilterItemsName` property defines the title of the wizard for this step. Use this property together with the `SourceFilterItemsPattern` property. Use these `SourceFilter*` properties with the `Model::Stereotype::DependentPairSourceFilters` property.

*Default = Match Items*

## SourceFilterItemsPattern

For the Pair Matching Wizard, the `SourceFilterItemsPattern` property defines the pattern that defines the content of the source domain tree. The content of the source domain tree for the current step is accumulated from the content of the previous step. See the help topic about context patterns for their definitions. Use this property together with the `SourceFilterItemsName` property. Use these `SourceFilter*` properties with the `Model::Stereotype::DependentPairSourceFilters` property.

*Default = Package\*, Class\* / Block\* / Object!\*, type:Object!\*, type:Port\*, providedInterface:Attribute\* / providedInterface:Operation\* / requiredInterface:Attribute\* / requiredInterface:Operation\* / type:Attribute\* / type:Operation\**

## SourceFilterPackagesName

For the Pair Matching Wizard, the `SourceFilterPackagesName` property defines the title of the wizard for this step. Use this property together with the `SourceFilterPackagesPattern` property. Use these `SourceFilter*` properties with the `Model::Stereotype::DependentPairSourceFilters` property.

*Default = Match Packages*

## SourceFilterPackagesPattern

For the Pair Matching Wizard, the `SourceFilterPackagesPattern` property defines the pattern that defines the content of the source domain tree. The content of the source domain tree for the current step is accumulated from the content of the previous step. See the help topic about context patterns for their definitions. Use this property together with the `SourceFilterPackagesName` property. Use these `SourceFilter*` properties with the `Model::Stereotype::DependentPairSourceFilters` property.

*Default = Package\**

## SourceFilterPartsName

For the Pair Matching Wizard, the `SourceFilterPartsName` property defines the title of the wizard for this step. Use this property together with the `SourceFilterPartsPattern` property. Use these `SourceFilter*` properties with the `Model::Stereotype::DependentPairSourceFilters` property.

*Default = Match Parts*

## SourceFilterPartsPattern

For the Pair Matching Wizard, the `SourceFilterPartsPattern` property defines the pattern that defines the content of the source domain tree. The content of the source domain tree for the current step is accumulated from the content of the previous step. See the help topic about context patterns for their definitions. Use this property together with the `SourceFilterPartsName` property. Use these `SourceFilter*` properties with the `Model::Stereotype::DependentPairSourceFilters` property.

*Default = Package\*, Class / Block / Object!, type:Object!\**

## SourceFilterPortsName

For the Pair Matching Wizard, the `SourceFilterPortsName` property defines the title of the wizard for this step. Use this property together with the `SourceFilterPortsPattern` property. Use these `SourceFilter*` properties with the `Model::Stereotype::DependentPairSourceFilters` property.

*Default = Match Ports*

## SourceFilterPortsPattern

For the Pair Matching Wizard, the `SourceFilterPortsPattern` property defines the pattern that defines the content of the source domain tree. The content of the source domain tree for the current step is accumulated from the content of the previous step. See the help topic about context patterns for their definitions. Use this property together with the `SourceFilterPortsName` property. Use these `SourceFilter*` properties with the `Model::Stereotype::DependentPairSourceFilters` property.

*Default = Package\*, Class / Block / Object!, type:Object!\*, type:Port*

## SourceFilterTagsName

For the Pair Matching Wizard, the `SourceFilterTagsName` property defines the title of the wizard for this step. Use this property together with the `SourceFilterTagsPattern` property. Use these `SourceFilter*` properties with the `Model::Stereotype::DependentPairSourceFilters` property.

*Default = Match Tags*

## SourceFilterTagsPattern

For the Pair Matching Wizard, the `SourceFilterTagsPattern` property defines the pattern that defines the content of the source domain tree. The content of the source domain tree for the current step is

accumulated from the content of the previous step. See the help topic about context patterns for their definitions. Use this property together with the SourceFilterTagsName property. Use these SourceFilter\* properties with the Model::Stereotype::DependentPairSourceFilters property.

*Default = Package\*, ModelElement\* / type:ModelElement\*, Tag*

## SourceRootNewTerm

For model synchronization, the SourceRootNewTerm property specifies the filter for selecting the source package, as used in the "Setup" step in the Pair Matching Wizard.

This property is relevant for DependentPairSet.

*Default = Package*

## Sources

When you create a New Term stereotype that is applicable to a type of relation, for example, a dependency or a flow, Rational Rhapsody will allow you to draw such an element between any two elements that can be connected by that type of relation.

If, however, you would like to limit the types of elements that can be connected by the New Term you defined, you can use the Model::Stereotype::Sources property to list the elements that you want to permit to be the source of such a relation.

The value of the property should be a comma-separated list of metaclasses (using the terms listed in the metaclasses.txt file in <Rational Rhapsody installation path>\Doc) and/or New Terms.

You can use Model::Stereotype::Targets to specify the list of elements that you want to permit to be the target of such a relation.

*Default = Blank*

## TargetRootNewTerm

For model synchronization, the TargetRootNewTerm property specifies the filter for selecting the target package, as seen in the "Setup" step in the Pair Matching Wizard.

This property is relevant for DependentPairSet.

*Default = ARPackage*

## Targets

When you create a New Term stereotype that is applicable to a type of relation, for example, a dependency or a flow, Rational Rhapsody will allow you to draw such an element between any two elements that can be connected by that type of relation.

If, however, you would like to limit the types of elements that can be connected by the New Term you defined, you can use the `Model::Stereotype::Targets` property to list the elements that you want to permit to be the target of such a relation.

The value of the property should be a comma-separated list of metaclasses (using the terms listed in the `metaclasses.txt` file in <Rational Rhapsody installation path>\Doc) and/or New Terms.

You can use `Model::Stereotype::Sources` to specify the list of elements that you want to permit to be the source of such a relation.

*Default = Blank*

## TableLayout

The `TableLayout` metaclass contains properties that you can use for the design of the table layout.

### ContextPattern

When you create a `TableLayout`, you can have each row represent a single element in the model, with each column displaying additional information about that element. Alternatively, you can have a row of data represent a hierarchy of elements in the model. For example, you could have the first column represent top-level packages in the model, the second column represent the classes in those packages, and the third column represent the operations that are contained in those classes.

To create such a hierarchical table, you must use the `ContextPattern` property to define the elements that should be displayed in the various columns.

For example, if you wanted a row to show packages (all levels recursively), followed by the classes in each package, followed by the operations defined in each class, you would set the value of the `ContextPattern` property to: `{pkg}Package*,{cls}Class,{op}Operation`. (Note that in this example, the text inside the brackets serves as category labels and can be any text you like.)

The syntax to use for the property includes quite a few advanced options. For a full explanation of the syntax, search for "context patterns" in the Rhapsody help.

You can find a video that demonstrates the use of context patterns in Rhapsody tables at:

<https://www.youtube.com/watch?v=iTHTxF5vOMc&list=PLZGO0qYNSD4VrcVNWT5ltkBI8vbkmDY0Y&index=7>

*Default = Blank*

### ContextTableColumnNameFormat

When you define a `TableLayout` element, the default value used for the name of a column is the text that appears in the "Property" that was selected for that column. However, if you are using a context pattern for the column, the default value for the column name is the value that was defined for the property `ContextTableColumnNameFormat`.

*Default = \$(Property) in \$(Context)*

## ShowContainerElementForPorts

This property instructs Rational Rhapsody to look at Ports as well as its container elements when displaying From/To information of Links and Flows.

*Default = Checked*

## TableView

The TableView metaclass contains properties that affect the appearance and behavior of TableViews in your model.

## ChangeSelectionOnEnterTo

The ChangeSelectionOnEnterTo property can be used to control what cell in the table or matrix receives the focus after you press the Enter key. You can use this property to specify that the focus should move to the cell to the right of the current cell, the cell to the left of the current cell, the cell below the current cell, or the cell above the current cell.

When you press the combination of Shift+Enter, the focus will always move in the direction opposite the direction specified with this property. For example, if ChangeSelectionOnEnterTo is set to "right", pressing Shift+Enter will move the focus to the cell to the left of the current cell.

*Default = down*

## Type

The Type metaclass contains properties that specify which extra prefixes are added to attributes, variables, and arguments for a specific type.

## PrefixForAttribute

The PrefixForAttribute property specifies the extra prefix added to the model attributes, variables, and arguments of this type, if the Model::Attribute::UsePrefix property is set to True.

Note that when Rational Rhapsody generates the code, the accessor and mutators do not include the prefix. For example, consider an attribute named A. If the Model::Attribute::UsePrefix property is set to True, the Model::Attribute::Prefix property is set to m, and the PrefixForAttribute property is set to t and you change the attribute type:

- The accessor and mutator for the attribute are getA and setA.
- The actual name of the attribute is m\_tA.

You can change the name of a variable, attribute, or argument so it does not obey the prefix. In this case, the element remains without a prefix until you change its type.

*Default = Blank*

# ModelBasedTesting

---

The ModelBasedTesting subject contains metaclasses that contain properties that affect the Automatic Test Generator (ATG) and TestConductor tool. This subject is available only if you have installed TestConductor.

## Settings

The Settings metaclass contains several properties with options for ATG and TestConductor.

### RequirementCoverageExcludedMetaClasses

The RequirementCoverageExcludedMetaClasses property determines meta classes of model elements to be excluded when computing requirement coverage.

If the value of the property is empty, ATG and TestConductor are considering model elements of all meta classes for computation of requirement coverage results.

If the value of the property contains a (comma separated) list of meta classes, ATG and TestConductor are considering only model elements of the meta classes which are not listed in the property for computation of requirement coverage results.

*Default = Empty string*

### RequirementCoverageModelElementsScope

The RequirementCoverageModelElementsScope property determines the scope for model elements to be regarded when computing requirement coverage.

If the value of the property is empty, ATG and TestConductor are considering model elements from the whole project for computation of requirement coverage results.

If the value of the property contains a (comma separated) list of packages (each with its full path name), ATG and TestConductor are considering only model elements located in these packages (and their sub packages) for computation of requirement coverage results.

*Default = Empty string*

### RequirementCoverageRegardedTags

The RequirementCoverageRegardedTags property determines if only requirements with certain tag values are considered when computing requirement coverage.

If the value of the property is empty, ATG and TestConductor are considering any requirement for

computation of requirement coverage results.

If the value of the property contains a (comma separated) list of tag names and values, ATG and TestConductor are considering only requirements with a tag value from the list for computation of requirement coverage results.

Example: If only requirements with a tag RequirementType with the value functional shall be considered, the property value has to be set to: RequirementType=functional

*Default = Empty string*

## **RequirementCoverageRequirementsScope**

The RequirementCoverageRequirementsScope property determines the scope for requirements elements to be regarded when computing requirement coverage.

If the value of the property is empty, ATG and TestConductor are considering requirements from the whole project for computation of requirement coverage results.

If the value of the property contains a (comma separated) list of packages (each with its full path name), ATG and TestConductor are considering only requirements located in these packages (and their sub packages) for computation of requirement coverage results.

*Default = Empty string*

## **RequirementCoverageTransitivityOfDependencies**

The RequirementCoverageTransitivityOfDependencies property controls if refinement of requirements and model elements is considered or not.

If the property is checked, ATG and TestConductor are considering also indirect dependencies between requirements and model elements for computation of requirement coverage.

If the property is cleared, ATG and TestConductor are considering only requirements and model elements with a direct dependency between them for computation of requirement coverage.

*Default = Cleared*

## **StereotypesForDependenciesToRequirements**

The StereotypesForDependenciesToRequirements property controls which dependencies between model elements and requirements are to be regarded when computing requirement coverage.

If the value of the property is empty, ATG and TestConductor are considering all dependencies between model elements and requirements for computation of requirement coverage.

If the value of the property contains a (comma separated) list of stereotype names, ATG and TestConductor are considering only dependencies between model elements and requirements for



computation of requirement coverage which are stereotyped with at least one stereotype from the list.

*Default = trace,satisfy*

# ObjectModelGe

---

The ObjectModelGe subject contains metaclasses that contain properties that determine the appearance and behavior of object model diagrams.

## Actor

The Actor metaclass contains properties that control the appearance of actors in object model diagrams.

### ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values for this property varies for the different elements, as does the default value used. The different values used are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is opened with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

*Default = Relative*

### ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

# Aggregation

The Aggregation metaclass contains a property that controls the appearance of aggregation lines in object model diagrams.

## DefaultLabel

The DefaultLabel property is used to specify the text area that should be highlighted for editing when you draw an association. The options available are:

- the target role
- the source role
- the target multiplicity
- the source multiplicity
- the name of the association (default behavior prior to release 8.1.4)

*Default = Target\_Role*

## line\_style

The line\_style property specifies the default line style for a graphical item.

The possible values are:

- straight\_arrows - straight line.
- rectilinear\_arrows - rectilinear line with right-angled corners placed at appropriate locations, depending on the starting and ending points of the line
- spline\_arrows - curved line without corners
- rounded\_rectilinear\_arrows - similar to ordinary rectilinear lines but use rounded bends rather than straight ninety-degree angles

*Default = spline\_arrows*

*Default = rectilinear\_arrows*

## ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values for this property varies for the different elements, as does the default value used. The different values used are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is opened with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

*Default = Name*

## ShowRedefinesLabel

The ShowRedefinesLabel property displays or hides a Redefines label for a class. This label indicates that the properties of a particular class have been redefined from a set of original properties that were derived from a base class. Redefined elements of a class can include attributes, ports, parts and association ends.

*Default = Enabled*

## ShowRoleVisibility

For associations and links, if you have chosen to display the names of the association ends, you can use the property ShowRoleVisibility to specify that alongside the name there should be an indication of the visibility of the member that represents the association end.

When this property is set to True, the following symbols are displayed to reflect the visibility:

- public +
- protected #
- private -
- static \$

For individual associations or links, you can turn on the display of member visibility information by opening the Display Options dialog and selecting the Visibility check box.

*Default = False*

## ShowSourceMultiplicity

An association, link, aggregation, and composition have two ends and each end might be assigned a multiplicity number (1, 1.x, and so on). This property controls whether the assigned multiplicity number is visible (selected check box) or not (cleared check box) on the "source" end of the line.

*Default = Checked*

## **ShowSourceQualifier**

The ShowSourceQualifier property is used for a number of connector elements, such as Associations. When set to True, the source element attribute defined as a qualifier for the association is displayed alongside the connector, for example attribute\_0. The property can be set at the diagram level. The default value for this property is True for object model diagrams and False for use case diagrams. The Display Options window can be used to change the qualifier show/hide setting for an individual connector. However, it does not change the value of the property at the diagram level.

*Default = Checked*

## **ShowSourceRole**

The ShowSourceRole property is used for a variety of connector elements, such as Links and Associations. When set to True, the source end of the relationship is displayed alongside the connector, for example itsClass\_1. The property is set at the diagram level.

*Default = Cleared*

## **ShowStereotype**

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = None*

## **ShowTargetMultiplicity**

An association, link, aggregation, and composition have two ends and each end might be assigned a multiplicity number (1, 1.x, and so on). This property controls whether the assigned multiplicity number is visible (selected check box) or not (cleared check box) on the "target" end of the line.

*Default = Checked*

## **ShowTargetQualifier**

The ShowTargetQualifier property is used for a number of connector elements, such as Associations. When set to Checked, the target element attribute defined as a qualifier for the association is displayed alongside the connector, for example attribute\_1. The property can be set at the diagram level.

*Default = Checked*

You can use the Display Options window to change the qualifier show/hide setting for an individual connector. However, it does not change the value of the property at the diagram level.

## ShowTargetRole

The ShowTargetRole property is used for a variety of connector elements, such as Links and Associations. When set to True, the target end of the relationship is displayed alongside the connector, for example itsClass\_2. The property is set at the diagram level.

*Default = Cleared*

## Association

The Association metaclass contains properties that control the appearance of association lines in object model diagrams.

### DefaultLabel

The DefaultLabel property is used to specify the text area that should be highlighted for editing when you draw an association. The options available are:

- the target role
- the source role
- the target multiplicity
- the source multiplicity
- the name of the association (default behavior prior to release 8.1.4)

*Default = Target\_Role*

### line\_style

The line\_style property specifies the default line style for a graphical item.

The possible values are:

- straight\_arrows - straight line.
- rectilinear\_arrows - rectilinear line with right-angled corners placed at appropriate locations, depending on the starting and ending points of the line
- spline\_arrows - curved line without corners
- rounded\_rectilinear\_arrows - similar to ordinary rectilinear lines but use rounded bends rather than straight ninety-degree angles

*Default = spline\_arrows*

*Default = rectilinear\_arrows*

## ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values for this property varies for the different elements, as does the default value used. The different values used are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is opened with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

*Default = Name*

## ShowRedefinesLabel

The ShowRedefinesLabel property displays or hides a Redefines label for a class. This label indicates that the properties of a particular class have been redefined from a set of original properties that were derived from a base class. Redefined elements of a class can include attributes, ports, parts and association ends.

*Default = Enabled*

## ShowRoleVisibility

For associations and links, if you have chosen to display the names of the association ends, you can use the property ShowRoleVisibility to specify that alongside the name there should be an indication of the visibility of the member that represents the association end.

When this property is set to True, the following symbols are displayed to reflect the visibility:

- public +
- protected #

- private -
- static \$

For individual associations or links, you can turn on the display of member visibility information by opening the Display Options dialog and selecting the Visibility check box.

*Default = False*

## ShowSourceMultiplicity

An association, link, aggregation, and composition have two ends and each end might be assigned a multiplicity number (1, 1.x, and so on). This property controls whether the assigned multiplicity number is visible (selected check box) or not (cleared check box) on the "source" end of the line.

*Default = Checked*

## ShowSourceQualifier

The ShowSourceQualifier property is used for a number of connector elements, such as Associations. When set to True, the source element attribute defined as a qualifier for the association is displayed alongside the connector, for example attribute\_0. The property can be set at the diagram level. The default value for this property is True for object model diagrams and False for use case diagrams. The Display Options window can be used to change the qualifier show/hide setting for an individual connector. However, it does not change the value of the property at the diagram level.

*Default = Checked*

## ShowSourceRole

The ShowSourceRole property is used for a variety of connector elements, such as Links and Associations. When set to True, the source end of the relationship is displayed alongside the connector, for example itsClass\_1. The property is set at the diagram level.

*Default = Cleared*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = None*



## ShowTargetMultiplicity

An association, link, aggregation, and composition have two ends and each end might be assigned a multiplicity number (1, 1.x, and so on). This property controls whether the assigned multiplicity number is visible (selected check box) or not (cleared check box) on the "target" end of the line.

*Default = Checked*

## ShowTargetQualifier

The ShowTargetQualifier property is used for a number of connector elements, such as Associations. When set to Checked, the target element attribute defined as a qualifier for the association is displayed alongside the connector, for example attribute\_1. The property can be set at the diagram level.

*Default = Checked*

You can use the Display Options window to change the qualifier show/hide setting for an individual connector. However, it does not change the value of the property at the diagram level.

## ShowTargetRole

The ShowTargetRole property is used for a variety of connector elements, such as Links and Associations. When set to True, the target end of the relationship is displayed alongside the connector, for example itsClass\_2. The property is set at the diagram level.

*Default = Cleared*

# Attribute

The Attribute metaclass contains properties that control attributes in object model diagrams.

## Compartments

The Compartments property determines which of the available compartments are displayed by default for the various types of elements. The value for this property is a comma-delimited string containing the names of the compartments that should be visible. Since the available compartments vary from element to element, it is recommended that you do not try to set the value of this property by using the Properties window or directly in the .prp file. Rather, you should use an element Display Options to set which compartments are visible, and then use the Make Default option to apply these settings at the diagram or project level for new elements of this type.

*Default = Empty MultiLine*

## QuickNavigationCategories

Graphic elements in diagrams can include a list of shortcuts that are accessible from the title bar of the element. You can use the property QuickNavigationCategories to specify the categories of items that should be included in this shortcut list and the order in which the categories should appear in the list. The value of the property should be a comma-separated list of categories. The categories that can be included are:

- Activities
- DiagramReferences
- Diagrams
- Hyperlinks
- InternalBlockDiagrams
- MainBehavior
- MainDiagram
- References
- StateCharts

If you want greater control over the content of the navigation list (such as showing/hiding specific elements within a given category, or changing the order of elements within a specific category), use the Compartments list, which can be found under "Display Options".

*Default = Hyperlinks*

## ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values for this property varies for the different elements, as does the default value used. The different values used are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is opened with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

*Default = Name\_only*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## ShowRedefinesLabel

The ShowRedefinesLabel property displays or hides a Redefines label for a class. This label indicates that the properties of a particular class have been redefined from a set of original properties that were derived from a base class. Redefined elements of a class can include attributes, ports, parts and association ends.

*Default = Enabled*

# AutoPopulate

The AutoPopulate metaclass contains properties that can be used to control the appearance of diagrams that are drawn automatically by Rational Rhapsody.

## ArrowDirection

The ArrowDirection property is used when Rational Rhapsody automatically generates a diagram, and it determines whether the flow of connectors in the diagram runs from top to bottom or bottom to top.

There are two situations where Rational Rhapsody automatically generates diagrams:

- If you have selected the Populate Diagrams option for Reverse Engineering (for those diagrams where this feature is supported).
- If you double-click a diagram in the browser that was generated by using the Rational Rhapsody API.

*Default = Bottom-Top*

## EnabledOnUpdateModel

The EnabledOnUpdateModel property specifies whether an object model diagram should be automatically updated when code changes are roundtripped into a model.

For object model diagrams that were initially created by Rational Rhapsody during reverse engineering, the value of this property is set to True.

*Default = False*

## LayoutStyle

The LayoutStyle property is used when Rational Rhapsody automatically generates a diagram, and it determines the general appearance of the diagram - hierarchical or orthogonal.

- Hierarchical - diagram layout reflects a hierarchy, appropriate for relationships such as inheritance.
- Orthogonal - diagram layout resembles a grid, appropriate where there are no clear hierarchical relationships between the elements in the diagram.

There are two situations where Rational Rhapsody automatically generates diagrams:

- If you have selected the Populate Diagrams option for Reverse Engineering (for those diagrams where this feature is supported).
- If you double-click a diagram in the browser that was generated by using the Rational Rhapsody API.

*Default = Hierarchical*

## Class

The Class metaclass contains properties that control the appearance of new class boxes drawn in object model diagrams.

## Compartments

The Compartments property determines which of the available compartments are displayed by default for the various types of elements. The value for this property is a comma-delimited string containing the names of the compartments that should be visible. Since the available compartments vary from element to element, it is recommended that you do not try to set the value of this property by using the Properties window or directly in the .prp file. Rather, you should use an element Display Options to set which compartments are visible, and then use the Make Default option to apply these settings at the diagram or project level for new elements of this type.

*Default = Empty MultiLine*

## PortsSingleLabelLayout

The PortsSingleLabelLayout property provides you with an option to combine the various labels for a port (name, stereotype, interface) into a single line, rather than having them displayed independently.

When you select the single label option, the combined label is also locked so it can not be moved unless you move the port itself. Note also that when you use this option, labels for ports located on the top and

bottom of an element will be rotated 90 degrees in order to avoid overlapping labels.

This property affects both ports and flowports.

The possible values for this property are:

- Off - port labels are not combined into a single label
- Internal - port labels are combined into a single label and positioned inside the parent element
- External - port labels are combined into a single label and positioned outside the parent element

When using the single label option, the order of the components of the single label can be controlled using the properties `ObjectModelGe::Port::SingleLabelFormat` and `ObjectModelGe::flowPort::SingleLabelFormat`.

*Default = Off*

## QuickNavigationCategories

Graphic elements in diagrams can include a list of shortcuts that are accessible from the title bar of the element. You can use the property `QuickNavigationCategories` to specify the categories of items that should be included in this shortcut list and the order in which the categories should appear in the list. The value of the property should be a comma-separated list of categories. The categories that can be included are:

- Activities
- DiagramReferences
- Diagrams
- Hyperlinks
- InternalBlockDiagrams
- MainBehavior
- MainDiagram
- References
- StateCharts

If you want greater control over the content of the navigation list (such as showing/hiding specific elements within a given category, or changing the order of elements within a specific category), use the `Compartment` list, which can be found under "Display Options".

*Default = MainBehavior,MainDiagram,Hyperlinks,Diagrams*

## ShowAttributes

The `ShowAttributes` property specifies which attributes are shown in an object box in a component diagram. The possible values are:

- All - Show all attributes.
- None - Do not show any attributes.

- Public - Show only the public attributes.
- Explicit - Show only those attributes that you have explicitly selected.

*Default = Explicit*

## ShowInheritedAttributes

This property filters the list of inherited attributes on the Attributes tab of the Features window. When this property is cleared, the list of inheritance is shown only for the selected object, class, or stereotype. When this property is checked, the list includes inheritance from the base object, class, or stereotype.

*Default = Cleared*

## ShowInheritedOperations

The ShowInheritedOperations property specifies how operations inherited from a base class are shown.

*Default = Cleared*

## ShowName

The ShowName property specifies how the name of an object should be displayed. The possible values are:

- Full\_path - Show the object name with the full path.. For example, "Default::A.B."
- Relative - Show the object name with a relative path. For example, "A.B."
- Name\_only - Show only the object name without any path information. For example, "B."
- Label - Show only the label.

*Default = Relative*

## ShowOperations

The ShowOperations property specifies which operations to show in an object box in a component or object model diagram. The possible values are:

- All - Show all operations.
- None - Do not show any operations.
- Public - Show only the public operations.
- Explicit - Show only those operations that you have explicitly selected.

*Default = Explicit*

## ShowPorts

The ShowPorts property determines whether ports are displayed in object model diagrams.

*Default = Checked*

## ShowPortsInterfaces

The ShowPortsInterfaces property determines whether port interfaces are displayed in object model diagrams.

*Default = Checked*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## ClassDiagram

The ClassDiagram metaclass contains a property that controls the default fill color of class diagrams in object model diagrams.

### Fillcolor

The Fillcolor property specifies the default fill color for the object.

*Default = 218,218,218*

## TreeContainmentStyle

Rhapsody allows you to display namespace containment in object model diagrams. This type of notation is also referred to as “alternative membership notation.” It depicts the hierarchical relationship between elements and the element that contains them, for example:

- requirements that contain other requirements
- packages that contain classes
- classes that contain other classes

The ability to display namespace containment is controlled by the `TreeContainmentStyle` property, which can be set at the diagram, package, or project level. Namespace containment can only be displayed if the property is set to `True`.

*Default = False (Note that in the SysML profile the default value of the property is True.)*

If you have enabled the display of namespace containment by setting the value of the property to `True`, you can then display namespace containment as follows:

Drag the “container” element and the “contained” elements to the diagram. Then, from the menu, select `Layout > Complete Relations > All`.

The hierarchical relationship between the elements are depicted in the diagram.

Alternatively, you can select the `Populate Diagram` option when creating a new diagram. If you then select elements that have a hierarchical relationship, the diagram created displays the namespace containment for the elements.

Note that there is no drawing tool to manually draw this type of relationship on the canvas. Containment relationships between elements can only be displayed automatically based on existing relationships, using one of the methods described above.

## Comment

The `Comment` metaclass contains properties that control comments in object model diagrams.

### CommentNotation

The `CommentNotation` property determines how `Comment` elements are displayed. This property can be set to one of two styles:

- `Note_Style`
- `Box_Style`

If `CommentNotation` is set to `Note_Style`, then the appearance of the element is determined by the value selected for the property `Comment::ShowForm: Note, Plain, or PushPin`.

If `CommentNotation` is set to `Box_Style`, then the element resembles a class-box, and you can control the display of compartments for the element.

*Default = Note\_Style*

### Compartments

The `Compartments` property determines which of the available compartments are displayed by default for the various types of elements. The value for this property is a comma-delimited string containing the



names of the compartments that should be visible. Since the available compartments vary from element to element, it is recommended that you do not try to set the value of this property by using the Properties window or directly in the .prp file. Rather, you should use an element Display Options to set which compartments are visible, and then use the Make Default option to apply these settings at the diagram or project level for new elements of this type.

*Default = Empty MultiLine*

## QuickNavigationCategories

Graphic elements in diagrams can include a list of shortcuts that are accessible from the title bar of the element. You can use the property QuickNavigationCategories to specify the categories of items that should be included in this shortcut list and the order in which the categories should appear in the list. The value of the property should be a comma-separated list of categories. The categories that can be included are:

- Activities
- DiagramReferences
- Diagrams
- Hyperlinks
- InternalBlockDiagrams
- MainBehavior
- MainDiagram
- References
- StateCharts

If you want greater control over the content of the navigation list (such as showing/hiding specific elements within a given category, or changing the order of elements within a specific category), use the Compartments list, which can be found under "Display Options".

*Default = Hyperlinks*

## ShowAnnotationContents

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the text displayed on the element in the diagram determined by the value that was set for the property ShowAnnotationContents. The property can take one of the following values:

- Name - the name of the element
- Label - the label assigned to the element
- Description - the content of the Description field
- Specification (for Requirements only) - the content of the Specification field

*Default = Description*

## ShowForm

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the appearance of the element is determined by the value that was set for the property ShowForm.

The possible values for this property vary for the different elements, as does the default value used. The values used are:

- Plain - No color background behind text
- Note - Color background behind text
- Pushpin - Color background plus pin icon

*Default = Note*

## ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values for this property varies for the different elements, as does the default value used. The different values used are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is opened with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

*Default = Relative*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## Complete

The metaclass Complete contains properties that determine whether or not Rational Rhapsody automatically draws the relations that exist between an element added to a diagram and elements already on the diagram.

### Complete\_Relation

The Complete\_Relation property is used to specify that when an element is added to a diagram, Rational Rhapsody should automatically draw the relations that exist between the element and elements already on the diagram.

*Default = Cleared*

## Composition

The Composition metaclass contains a property that controls the appearance of compositions in object model diagrams.

### DefaultLabel

The DefaultLabel property is used to specify the text area that should be highlighted for editing when you draw an association. The options available are:

- the target role
- the source role
- the target multiplicity
- the source multiplicity
- the name of the association (default behavior prior to release 8.1.4)

*Default = Target\_Role*

### line\_style

The line\_style property specifies the default line style for a graphical item.

The possible values are:

- straight\_arrows - straight line.
- rectilinear\_arrows - rectilinear line with right-angled corners placed at appropriate locations, depending on the starting and ending points of the line

- `spline_arrows` - curved line without corners
- `rounded_rectilinear_arrows` - similar to ordinary rectilinear lines but use rounded bends rather than straight ninety-degree angles

*Default = `spline_arrows`*

*Default = `rectilinear_arrows`*

## RepresentParts

The `RepresentParts` property determines what type of element is added to the Rational Rhapsody object model diagram when you draw a composition connector (black diamond).

By default, when you add a composition relationship to a diagram, you see an Association that is added in the diagram.

If you prefer that Rational Rhapsody display this relationship as a Part in the diagram, set the value of this property to `True`.

*Default = `Cleared`*

## ShowName

The `ShowName` property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values for this property varies for the different elements, as does the default value used. The different values used are:

- `Description` - the content of the description field; relevant for elements such as comments
- `Full_path` - the full path describing the hierarchical position of an element, for example, `package_1::package_1b::class_0`
- `Label` - the label provided for the element
- `Name` - the name of the element
- `Name_only` - the name of the element only (as opposed to the full or relative path)
- `None` - nothing should be displayed
- `Relative` - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is opened with the class name only if the class is not positioned inside the package in the diagram.
- `Specification` - the content of the specification field; relevant for elements such as constraints

*Default = `Name`*

## ShowRedefinesLabel

The ShowRedefinesLabel property displays or hides a Redefines label for a class. This label indicates that the properties of a particular class have been redefined from a set of original properties that were derived from a base class. Redefined elements of a class can include attributes, ports, parts and association ends.

*Default = Enabled*

## ShowRoleVisibility

For associations and links, if you have chosen to display the names of the association ends, you can use the property ShowRoleVisibility to specify that alongside the name there should be an indication of the visibility of the member that represents the association end.

When this property is set to True, the following symbols are displayed to reflect the visibility:

- public +
- protected #
- private -
- static \$

For individual associations or links, you can turn on the display of member visibility information by opening the Display Options dialog and selecting the Visibility check box.

*Default = False*

## ShowSourceMultiplicity

An association, link, aggregation, and composition have two ends and each end might be assigned a multiplicity number (1, 1.x, and so on). This property controls whether the assigned multiplicity number is visible (selected check box) or not (cleared check box) on the "source" end of the line.

*Default = Checked*

## ShowSourceQualifier

The ShowSourceQualifier property is used for a number of connector elements, such as Associations. When set to True, the source element attribute defined as a qualifier for the association is displayed alongside the connector, for example attribute\_0. The property can be set at the diagram level. The default value for this property is True for object model diagrams and False for use case diagrams. The Display Options window can be used to change the qualifier show/hide setting for an individual connector. However, it does not change the value of the property at the diagram level.

*Default = Checked*

## ShowSourceRole

The ShowSourceRole property is used for a variety of connector elements, such as Links and Associations. When set to True, the source end of the relationship is displayed alongside the connector, for

example itsClass\_1. The property is set at the diagram level.

*Default = Cleared*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = None*

## ShowTargetMultiplicity

An association, link, aggregation, and composition have two ends and each end might be assigned a multiplicity number (1, 1.x, and so on). This property controls whether the assigned multiplicity number is visible (selected check box) or not (cleared check box) on the "target" end of the line.

*Default = Checked*

## ShowTargetQualifier

The ShowTargetQualifier property is used for a number of connector elements, such as Associations. When set to Checked, the target element attribute defined as a qualifier for the association is displayed alongside the connector, for example attribute\_1. The property can be set at the diagram level.

*Default = Checked*

You can use the Display Options window to change the qualifier show/hide setting for an individual connector. However, it does not change the value of the property at the diagram level.

## ShowTargetRole

The ShowTargetRole property is used for a variety of connector elements, such as Links and Associations. When set to True, the target end of the relationship is displayed alongside the connector, for example itsClass\_2. The property is set at the diagram level.

*Default = Cleared*

# Constraint

The Constraint metaclass contains properties that control the constraints in object model diagrams.

## Compartments

The Compartments property determines which of the available compartments are displayed by default for the various types of elements. The value for this property is a comma-delimited string containing the names of the compartments that should be visible. Since the available compartments vary from element to element, it is recommended that you do not try to set the value of this property by using the Properties window or directly in the .prp file. Rather, you should use an element Display Options to set which compartments are visible, and then use the Make Default option to apply these settings at the diagram or project level for new elements of this type.

*Default = Specification*

## ConstraintNotation

The ConstraintNotation property determines how Constraint elements are displayed. This property can be set to one of two styles:

- Note\_Style
- Box\_Style

If ConstraintNotation is set to Note\_Style, then the appearance of the element is determined by the value selected for the property Constraint::ShowForm: Note, Plain, or PushPin.

If ConstraintNotation is set to Box\_Style, then the element resembles a class-box, and you can control the display of compartments for the element.

*Default = Note\_Style*

## QuickNavigationCategories

Graphic elements in diagrams can include a list of shortcuts that are accessible from the title bar of the element. You can use the property QuickNavigationCategories to specify the categories of items that should be included in this shortcut list and the order in which the categories should appear in the list. The value of the property should be a comma-separated list of categories. The categories that can be included are:

- Activities
- DiagramReferences
- Diagrams
- Hyperlinks
- InternalBlockDiagrams
- MainBehavior
- MainDiagram
- References

- StateCharts

If you want greater control over the content of the navigation list (such as showing/hiding specific elements within a given category, or changing the order of elements within a specific category), use the Compartments list, which can be found under "Display Options".

*Default = Hyperlinks*

## ShowAnnotationContents

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the text displayed on the element in the diagram determined by the value that was set for the property ShowAnnotationContents. The property can take one of the following values:

- Name - the name of the element
- Label - the label assigned to the element
- Description - the content of the Description field
- Specification (for Requirements only) - the content of the Specification field

*Default = Description*

## ShowForm

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the appearance of the element is determined by the value that was set for the property ShowForm.

The possible values for this property vary for the different elements, as does the default value used. The values used are:

- Plain - No color background behind text
- Note - Color background behind text
- Pushpin - Color background plus pin icon

*Default = Note*

## ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values for this property varies for the different elements, as does the default value used. The different values used are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element



- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is opened with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

*Default = Relative*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## ContainArrow

### line\_style

The line\_style property specifies the default line style for a graphical item.

The possible values are:

- straight\_arrows - straight line.
- rectilinear\_arrows - rectilinear line with right-angled corners placed at appropriate locations, depending on the starting and ending points of the line
- spline\_arrows - curved line without corners
- rounded\_rectilinear\_arrows - similar to ordinary rectilinear lines but use rounded bends rather than straight ninety-degree angles

*Default = spline\_arrows*

*Default = rectilinear\_arrows*

### ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values for this property varies for the different elements, as does the default value used. The different values used are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is opened with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

*Default = Name*

## Depends

The Depends metaclass contains a property that controls the appearance of dependency relation lines in object model diagrams.

### line\_style

The line\_style property specifies the default line style for a graphical item.

The possible values are:

- straight\_arrows - straight line.
- rectilinear\_arrows - rectilinear line with right-angled corners placed at appropriate locations, depending on the starting and ending points of the line
- spline\_arrows - curved line without corners
- rounded\_rectilinear\_arrows - similar to ordinary rectilinear lines but use rounded bends rather than straight ninety-degree angles

*Default = spline\_arrows*

*Default = straight\_arrows*

## ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values for this property varies for the different elements, as does the default value used. The different values used are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is opened with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

*Default = None*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## DiagramFrame

The DiagramFrame metaclass contains properties that control the appearance of a diagram frame.

### DefaultSize

The DefaultSize property specifies default values for diagram frame dimensions.

*Default = 20, 20, 590, 500*

## ShowName

The ShowName property determines the text to display next to a graphic element in a diagram. For most elements, you can provide a name and a label. Your descriptive label can be useful in cases where the name itself might not be sufficient due to various constraints, such as the inability to use spaces if the name of the element is to appear in the code.

The possible values for this property vary for the different elements, as does the default value used. The possible values are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is displayed with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

Note that for some elements, such as destruction elements in sequence diagrams and timing diagrams, the property is a boolean property that hides/shows the name of the element.

*Default = Name*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## Flow

The Flow metaclass contains properties that control how information flows are displayed in object model

diagrams.

## flowKeyword

The flowKeyword property is a Boolean value that specifies whether the flow keyword for the information flow is displayed in the diagram.

*Default = Checked*

## line\_style

The line\_style property specifies the default line style for a graphical item.

The possible values are:

- straight\_arrows - straight line.
- rectilinear\_arrows - rectilinear line with right-angled corners placed at appropriate locations, depending on the starting and ending points of the line
- spline\_arrows - curved line without corners
- rounded\_rectilinear\_arrows - similar to ordinary rectilinear lines but use rounded bends rather than straight ninety-degree angles

*Default = spline\_arrows*

*Default = rectilinear\_arrows*

## ShowConveyed

The ShowConveyed property determines whether or not item flows should be displayed alongside the flows that convey them, and if so, what text should be displayed for the item flow. The property can take any of the following values:

- Name - the name of the item flow
- Label - the label of the item flow
- None - nothing should be displayed for the item flow

Note that this property only affects the display of new flows added to a diagram. The display of item flows for flows already on a diagram can be controlled by selecting the Display Options... item from the context menu for flows.

*Default = Name*

## flowPort

The flowport metaclass contains properties that control how information flowports are displayed in object model diagrams.

## ShowName

The ShowName property determines the text to display next to a graphic element in a diagram. For most elements, you can provide a name and a label. Your descriptive label can be useful in cases where the name itself might not be sufficient due to various constraints, such as the inability to use spaces if the name of the element is to appear in the code.

The possible values for this property vary for the different elements, as does the default value used. The possible values are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is displayed with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

Note that for some elements, such as destruction elements in sequence diagrams and timing diagrams, the property is a boolean property that hides/shows the name of the element.

*Default = NameAndType*

## ShowRedefinesLabel

The ShowRedefinesLabel property displays or hides a Redefines label for a class. This label indicates that the properties of a particular class have been redefined from a set of original properties that were derived from a base class. Redefined elements of a class can include attributes, ports, parts and association ends.

*Default = Enabled*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.

- None - Do not show stereotypes in diagrams.

*Default = None*

## SingleLabelFormat

If you are using the option of combining port labels into a single one-line label (controlled by the property PortsSingleLabelLayout), you can use the SingleLabelFormat property to specify the order to use for the combined label.

*Default = \$Stereotype \$Name*

## fullPort

The fullPort metaclass contains properties that control how full ports are displayed.

### ShowName

The ShowName property determines the text to display next to a graphic element in a diagram. For most elements, you can provide a name and a label. Your descriptive label can be useful in cases where the name itself might not be sufficient due to various constraints, such as the inability to use spaces if the name of the element is to appear in the code.

The possible values for this property vary for the different elements, as does the default value used. The possible values are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is displayed with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

Note that for some elements, such as destruction elements in sequence diagrams and timing diagrams, the property is a boolean property that hides/shows the name of the element.

*Default = NameAndType*

## ShowRedefinesLabel

The ShowRedefinesLabel property displays or hides a Redefines label for a class. This label indicates that the properties of a particular class have been redefined from a set of original properties that were derived from a base class. Redefined elements of a class can include attributes, ports, parts and association ends.

*Default = Enabled*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = None*

## SingleLabelFormat

If you are using the option of combining port labels into a single one-line label (controlled by the property PortsSingleLabelLayout), you can use the SingleLabelFormat property to specify the order to use for the combined label.

*Default = \$Stereotype \$Name*

# General

Contains properties relating to the general appearance of object model diagrams.

## SysMLDiagramKindTitle

When a diagram is created in SysML projects, the diagram frame contains a header for the diagram that includes information such as the name of the package that contains the diagram. The property SysMLDiagramKindTitle is used to specify a prefix for the diagram frame header that reflects the type of diagram. For example, when you create an internal block diagram, the header in the diagram frame starts with the prefix "ibd".

Note that the structure of the entire header is controlled by the property General::Graphics::DiagramHeader.



# Inheritance

The Inheritance metaclass contains properties that control the appearance of inheritance lines in object model diagrams.

## line\_style

The line\_style property specifies the default line style for a graphical item.

The possible values are:

- straight\_arrows - straight line.
- rectilinear\_arrows - rectilinear line with right-angled corners placed at appropriate locations, depending on the starting and ending points of the line
- spline\_arrows - curved line without corners
- rounded\_rectilinear\_arrows - similar to ordinary rectilinear lines but use rounded bends rather than straight ninety-degree angles

*Default = spline\_arrows*

*Default = tree*

## ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values are:

- Name - the name of the element
- Label - the label provided for the element
- None - nothing should be displayed

*Default = None*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## InstanceSpecification

Contains properties relating to the display of InstanceSpecification elements in object model diagrams.

### QuickNavigationCategories

Graphic elements in diagrams can include a list of shortcuts that are accessible from the title bar of the element. You can use the property QuickNavigationCategories to specify the categories of items that should be included in this shortcut list and the order in which the categories should appear in the list. The value of the property should be a comma-separated list of categories. The categories that can be included are:

- Activities
- DiagramReferences
- Diagrams
- Hyperlinks
- InternalBlockDiagrams
- MainBehavior
- MainDiagram
- References
- StateCharts

If you want greater control over the content of the navigation list (such as showing/hiding specific elements within a given category, or changing the order of elements within a specific category), use the Compartments list, which can be found under "Display Options".

*Default = MainBehavior,Hyperlinks*

### ShowInnerSlots

The property ShowInnerSlots can be used to determine the level of detail displayed for slots that represent complex data types in:

- Instance Specification elements in object model diagrams
- Object Node elements that are associated with Instance Specifications, in activities

The property can be set to one of the following values:

- None - only the value of the complex type represented by the slot is displayed, not the values of the individual primitive types that it contains
- ExcludeReferencesToOtherInstances - the values of the individual primitive types that make up the complex type represented by the slot are displayed, but not the value of the complex type itself

- All - both the value of the complex type and the values of the individual primitive types that make up the complex type are displayed

*Default = None*

## Interface

The Interface metaclass contains properties that control the appearance of interfaces in object model diagrams.

## Compartment

The Compartments property determines which of the available compartments are displayed by default for the various types of elements. The value for this property is a comma-delimited string containing the names of the compartments that should be visible. Since the available compartments vary from element to element, it is recommended that you do not try to set the value of this property by using the Properties window or directly in the .prp file. Rather, you should use an element Display Options to set which compartments are visible, and then use the Make Default option to apply these settings at the diagram or project level for new elements of this type.

*Default = Empty MultiLine*

## PortSingleLabelLayout

The PortSingleLabelLayout property provides you with an option to combine the various labels for a port (name, stereotype, interface) into a single line, rather than having them displayed independently.

When you select the single label option, the combined label is also locked so it can not be moved unless you move the port itself. Note also that when you use this option, labels for ports located on the top and bottom of an element will be rotated 90 degrees in order to avoid overlapping labels.

This property affects both ports and flowports.

The possible values for this property are:

- Off - port labels are not combined into a single label
- Internal - port labels are combined into a single label and positioned inside the parent element
- External - port labels are combined into a single label and positioned outside the parent element

When using the single label option, the order of the components of the single label can be controlled using the properties `ObjectModelGe::Port::SingleLabelFormat` and `ObjectModelGe::flowPort::SingleLabelFormat`.

*Default = Off*

## QuickNavigationCategories

Graphic elements in diagrams can include a list of shortcuts that are accessible from the title bar of the element. You can use the property `QuickNavigationCategories` to specify the categories of items that should be included in this shortcut list and the order in which the categories should appear in the list. The value of the property should be a comma-separated list of categories. The categories that can be included are:

- Activities
- DiagramReferences
- Diagrams
- Hyperlinks
- InternalBlockDiagrams
- MainBehavior
- MainDiagram
- References
- StateCharts

If you want greater control over the content of the navigation list (such as showing/hiding specific elements within a given category, or changing the order of elements within a specific category), use the `Compartments` list, which can be found under "Display Options".

*Default = MainBehavior,MainDiagram,Hyperlinks,Diagrams*

## ShowAttributes

The `ShowAttributes` property specifies which attributes are shown in an object box in a component diagram. The possible values are:

- All - Show all attributes.
- None - Do not show any attributes.
- Public - Show only the public attributes.
- Explicit - Show only those attributes that you have explicitly selected.

*Default = Explicit*

## ShowInheritedAttributes

The `ShowInheritedAttributes` property specifies how attributes inherited from a base class are shown.

*Default = Cleared*

## ShowInheritedOperations

The `ShowInheritedOperations` property specifies how operations inherited from a base class are shown.

*Default = Cleared*

## ShowName

The ShowName property specifies how the name of an object should be displayed. The possible values are:

- Full\_path - Show the object name with the full path.. For example, "Default::A.B."
- Relative - Show the object name with a relative path. For example, "A.B."
- Name\_only - Show only the object name without any path information. For example, "B."
- Label - Show only the label.

*Default = Relative*

## ShowOperations

The ShowOperations property specifies which operations to show in an object box in a component or object model diagram. The possible values are:

- All - Show all operations.
- None - Do not show any operations.
- Public - Show only the public operations.
- Explicit - Show only those operations that you have explicitly selected.

*Default = Explicit*

## ShowPorts

The ShowPorts property is a Boolean value that determines whether ports are displayed in object model diagrams.

*Default = Checked*

## ShowPortsInterfaces

The ShowPortsInterfaces property is a Boolean value that determines whether port interfaces are displayed in object model diagrams.

*Default = Checked*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.

- None - Do not show stereotypes in diagrams.

*Default = Label*

## Link

The Link metaclass contains a property that controls how links are displayed in object model diagrams.

### line\_style

The line\_style property specifies the default line style for a graphical item.

The possible values are:

- straight\_arrows - straight line.
- rectilinear\_arrows - rectilinear line with right-angled corners placed at appropriate locations, depending on the starting and ending points of the line
- spline\_arrows - curved line without corners
- rounded\_rectilinear\_arrows - similar to ordinary rectilinear lines but use rounded bends rather than straight ninety-degree angles

*Default = spline\_arrows*

*Default = rectilinear\_arrows*

### ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values for this property varies for the different elements, as does the default value used. The different values used are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is opened with the class name only if the class is not positioned inside the package in the diagram.

- Specification - the content of the specification field; relevant for elements such as constraints

*Default = Name*

## ShowRoleVisibility

For associations and links, if you have chosen to display the names of the association ends, you can use the property ShowRoleVisibility to specify that alongside the name there should be an indication of the visibility of the member that represents the association end.

When this property is set to True, the following symbols are displayed to reflect the visibility:

- public +
- protected #
- private -
- static \$

For individual associations or links, you can turn on the display of member visibility information by opening the Display Options dialog and selecting the Visibility check box.

*Default = False*

## ShowSourceMultiplicity

An association, link, aggregation, and composition have two ends and each end might be assigned a multiplicity number (1, 1.x, and so on). This property controls whether the assigned multiplicity number is visible (selected check box) or not (cleared check box) on the "source" end of the line.

*Default = Cleared*

## ShowSourceQualifier

The ShowSourceQualifier property is used for a number of connector elements, such as Associations. When set to True, the source element attribute defined as a qualifier for the association is displayed alongside the connector, for example attribute\_0. The property can be set at the diagram level. The default value for this property is True for object model diagrams and False for use case diagrams. The Display Options window can be used to change the qualifier show/hide setting for an individual connector. However, it does not change the value of the property at the diagram level.

*Default = Checked*

## ShowSourceRole

The ShowSourceRole property is used for a variety of connector elements, such as Links and Associations. When set to True, the source end of the relationship is displayed alongside the connector, for example itsClass\_1. The property is set at the diagram level.

*Default = Cleared*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = None*

## ShowTargetMultiplicity

An association, link, aggregation, and composition have two ends and each end might be assigned a multiplicity number (1, 1.x, and so on). This property controls whether the assigned multiplicity number is visible (selected check box) or not (cleared check box) on the "target" end of the line.

*Default = Cleared*

## ShowTargetQualifier

The ShowTargetQualifier property is used for a number of connector elements, such as Associations. When set to Checked, the target element attribute defined as a qualifier for the association is displayed alongside the connector, for example attribute\_1. The property can be set at the diagram level.

*Default =*

You can use the Display Options window to change the qualifier show/hide setting for an individual connector. However, it does not change the value of the property at the diagram level.

## ShowTargetRole

The ShowTargetRole property is used for a variety of connector elements, such as Links and Associations. When set to True, the target end of the relationship is displayed alongside the connector, for example itsClass\_2. The property is set at the diagram level.

*Default = Cleared*

## Note

The Note metaclass contains a property that controls how notes are displayed in object model diagrams.



## ShowForm

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the appearance of the element is determined by the value that was set for the property ShowForm.

The possible values for this property vary for the different elements, as does the default value used. The values used are:

- Plain - No color background behind text
- Note - Color background behind text
- Pushpin - Color background plus pin icon

*Default = Note*

## Object

The Object metaclass contains properties that control the appearance of objects drawn in object model diagrams.

## Compartments

The Compartments property determines which of the available compartments are displayed by default for the various types of elements. The value for this property is a comma-delimited string containing the names of the compartments that should be visible. Since the available compartments vary from element to element, it is recommended that you do not try to set the value of this property by using the Properties window or directly in the .prp file. Rather, you should use an element Display Options to set which compartments are visible, and then use the Make Default option to apply these settings at the diagram or project level for new elements of this type.

*Default = Empty MultiLine*

## MultilineNameCompartment

The MultilineNameCompartment property specifies whether the name compartment for objects supports names that span more than one line.

*Default = Cleared*

## PortsSingleLabelLayout

The PortsSingleLabelLayout property provides you with an option to combine the various labels for a port (name, stereotype, interface) into a single line, rather than having them displayed independently.

When you select the single label option, the combined label is also locked so it can not be moved unless

you move the port itself. Note also that when you use this option, labels for ports located on the top and bottom of an element will be rotated 90 degrees in order to avoid overlapping labels.

This property affects both ports and flowports.

The possible values for this property are:

- Off - port labels are not combined into a single label
- Internal - port labels are combined into a single label and positioned inside the parent element
- External - port labels are combined into a single label and positioned outside the parent element

When using the single label option, the order of the components of the single label can be controlled using the properties `ObjectModelGe::Port::SingleLabelFormat` and `ObjectModelGe::flowPort::SingleLabelFormat`.

*Default = Off*

## QuickNavigationCategories

Graphic elements in diagrams can include a list of shortcuts that are accessible from the title bar of the element. You can use the property `QuickNavigationCategories` to specify the categories of items that should be included in this shortcut list and the order in which the categories should appear in the list. The value of the property should be a comma-separated list of categories. The categories that can be included are:

- Activities
- DiagramReferences
- Diagrams
- Hyperlinks
- InternalBlockDiagrams
- MainBehavior
- MainDiagram
- References
- StateCharts

If you want greater control over the content of the navigation list (such as showing/hiding specific elements within a given category, or changing the order of elements within a specific category), use the `Compartment`s list, which can be found under "Display Options".

*Default = MainBehavior,MainDiagram,Hyperlinks,Diagrams*

## ShowAttributes

The `ShowAttributes` property specifies which attributes are shown in an object box in a component diagram. The possible values are as follows:

- All - Show all attributes.

- None - Do not show any attributes.
- Public - Show only the public attributes.
- Explicit - Show only those attributes that you have explicitly selected.

*Default = Public*

## ShowInheritedAttributes

This property filters the list of inherited attributes on the Attributes tab of the Features window. When this property is cleared, the list of inheritance is shown only for the selected object, class, or stereotype. When this property is checked, the list includes inheritance from the base object, class, or stereotype.

*Default = Cleared*

## ShowInheritedOperations

The ShowInheritedOperations property specifies how operations inherited from a base object are shown.

*Default = Cleared*

## ShowName

The ShowName property specifies how the name of an object should be displayed. The possible values are as follows:

- Full\_path - Show the object name with the full path.. For example, "Default::A.B."
- Relative - Show the object name with a relative path. For example, "A.B."
- Name\_only - Show only the object name without any path information. For example, "B."
- Type - Show only the object type. For example, ":b." where B is an object of type b.

*Default = Relative*

## ShowOperations

The ShowOperations property specifies which operations to show in an object box in a component or object model diagram. The possible values are as follows:

- All - Show all operations.
- None - Do not show any operations.
- Public - Show only the public operations.
- Explicit - Show only those operations that you have explicitly selected.

*Default = Public*

## ShowPorts

The ShowPorts property is a Boolean value that determines whether ports are displayed in object model diagrams.

*Default = Checked*

## ShowPortsInterfaces

The ShowPortsInterfaces property (under ObjectModelGe::Class/Object) is a Boolean value that determines whether port interfaces are displayed in object model diagrams.

*Default = Checked*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## ShowStereotypeOfClass

The ShowStereotypeOfClass property is a Boolean property that specifies whether or not the stereotypes of a class for an object should be displayed on the object element when it is added to diagrams.

*Default = Checked*

## ShowRedefinesLabel

The ShowRedefinesLabel property displays or hides a Redefines label for a class. This label indicates that the properties of a particular class have been redefined from a set of original properties that were derived from a base class. Redefined elements of a class can include attributes, ports, parts and association ends.

*Default = Enabled*

# Operation

## Compartment

The Compartments property determines which of the available compartments are displayed by default for the various types of elements. The value for this property is a comma-delimited string containing the names of the compartments that should be visible. Since the available compartments vary from element to element, it is recommended that you do not try to set the value of this property by using the Properties window or directly in the .prp file. Rather, you should use an element Display Options to set which compartments are visible, and then use the Make Default option to apply these settings at the diagram or project level for new elements of this type.

*Default = Empty MultiLine*

## QuickNavigationCategories

Graphic elements in diagrams can include a list of shortcuts that are accessible from the title bar of the element. You can use the property QuickNavigationCategories to specify the categories of items that should be included in this shortcut list and the order in which the categories should appear in the list. The value of the property should be a comma-separated list of categories. The categories that can be included are:

- Activities
- DiagramReferences
- Diagrams
- Hyperlinks
- InternalBlockDiagrams
- MainBehavior
- MainDiagram
- References
- StateCharts

If you want greater control over the content of the navigation list (such as showing/hiding specific elements within a given category, or changing the order of elements within a specific category), use the Compartments list, which can be found under "Display Options".

*Default = Hyperlinks*

## ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values for this property varies for the different elements, as does the default value used. The different values used are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element

- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is opened with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

*Default = Name\_only*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## Package

The Package metaclass contains properties that control the appearance of packages in object model diagrams.

## QuickNavigationCategories

Graphic elements in diagrams can include a list of shortcuts that are accessible from the title bar of the element. You can use the property QuickNavigationCategories to specify the categories of items that should be included in this shortcut list and the order in which the categories should appear in the list. The value of the property should be a comma-separated list of categories. The categories that can be included are:

- Activities
- DiagramReferences
- Diagrams
- Hyperlinks
- InternalBlockDiagrams
- MainBehavior
- MainDiagram
- References

- StateCharts

If you want greater control over the content of the navigation list (such as showing/hiding specific elements within a given category, or changing the order of elements within a specific category), use the Compartments list, which can be found under "Display Options".

*Default = MainBehavior,MainDiagram,Hyperlinks,Diagrams*

## ShowName

The ShowName property specifies how the name of an object should be displayed. The possible values are as follows:

- Full\_path - Show the object name with the full path.. For example, "Default::A.B."
- Relative - Show the object name with a relative path. For example, "A.B."
- Name\_only - Show only the object name without any path information. For example, "B."

*Default = Name\_only*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## Port

The Port metaclass contains properties that control the appearance of ports in object model diagrams.

## ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values are:

- Name - the name of the element
- NameAndType - the name and type provided for the element
- Type - the type provided for the element

- Label - the label provided for the element
- None - nothing should be displayed

*Default = NameAndType*

## ShowRedefinesLabel

The ShowRedefinesLabel property displays or hides a Redefines label for a class. This label indicates that the properties of a particular class have been redefined from a set of original properties that were derived from a base class. Redefined elements of a class can include attributes, ports, parts and association ends.

*Default = Enabled*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = None*

## SingleLabelFormat

If you are using the option of combining port labels into a single one-line label (controlled by the property PortsSingleLabelLayout), you can use the SingleLabelFormat property to specify the order to use for the combined label.

*Default = \$Stereotype \$Name \$Interface*

# proxyPort

The proxyPort metaclass contains properties that control how proxy ports are displayed.

## ShowName

The ShowName property determines the text to display next to a graphic element in a diagram. For most elements, you can provide a name and a label. Your descriptive label can be useful in cases where the name itself might not be sufficient due to various constraints, such as the inability to use spaces if the name of the element is to appear in the code.

The possible values for this property vary for the different elements, as does the default value used. The



possible values are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is displayed with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

Note that for some elements, such as destruction elements in sequence diagrams and timing diagrams, the property is a boolean property that hides/shows the name of the element.

*Default = NameAndType*

## ShowRedefinesLabel

The ShowRedefinesLabel property displays or hides a Redefines label for a class. This label indicates that the properties of a particular class have been redefined from a set of original properties that were derived from a base class. Redefined elements of a class can include attributes, ports, parts and association ends.

*Default = Enabled*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = None*

## SingleLabelFormat

If you are using the option of combining port labels into a single one-line label (controlled by the property PortsSingleLabelLayout), you can use the SingleLabelFormat property to specify the order to use for the combined label.

*Default = \$Stereotype \$Name*

# Realization

The Realization metaclass contains properties that control the appearance of realization lines in object model diagrams.

## line\_style

The line\_style property specifies the default line style for a graphical item.

The possible values are:

- straight\_arrows - straight line.
- rectilinear\_arrows - rectilinear line with right-angled corners placed at appropriate locations, depending on the starting and ending points of the line
- spline\_arrows - curved line without corners
- rounded\_rectilinear\_arrows - similar to ordinary rectilinear lines but use rounded bends rather than straight ninety-degree angles

*Default = spline\_arrows*

*Default = tree*

## ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values are:

- Name - the name of the element
- Label - the label provided for the element
- None - nothing should be displayed

*Default = None*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## Requirement

### Compartments

The Compartments property determines which of the available compartments are displayed by default for the various types of elements. The value for this property is a comma-delimited string containing the names of the compartments that should be visible. Since the available compartments vary from element to element, it is recommended that you do not try to set the value of this property by using the Properties window or directly in the .prp file. Rather, you should use an element Display Options to set which compartments are visible, and then use the Make Default option to apply these settings at the diagram or project level for new elements of this type.

*Default = Specification*

### QuickNavigationCategories

Graphic elements in diagrams can include a list of shortcuts that are accessible from the title bar of the element. You can use the property QuickNavigationCategories to specify the categories of items that should be included in this shortcut list and the order in which the categories should appear in the list. The value of the property should be a comma-separated list of categories. The categories that can be included are:

- Activities
- DiagramReferences
- Diagrams
- Hyperlinks
- InternalBlockDiagrams
- MainBehavior
- MainDiagram
- References
- StateCharts

If you want greater control over the content of the navigation list (such as showing/hiding specific elements within a given category, or changing the order of elements within a specific category), use the Compartments list, which can be found under "Display Options".

*Default = Hyperlinks*

### RequirementNotation

The RequirementNotation property determines how Requirement elements are displayed. This property can be set to one of two styles:

- Note\_Style
- Box\_Style

If RequirementNotation is set to Note\_Style, then the appearance of the element is determined by the value selected for the property Requirement::ShowForm: Note, Plain, or PushPin.

If RequirementNotation is set to Box\_Style, then the element resembles a class-box, and you can control the display of compartments for the element.

*Default = Note\_Style*

## ShowAnnotationContents

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the text displayed on the element in the diagram determined by the value that was set for the property ShowAnnotationContents. The property can take one of the following values:

- Name - the name of the element
- Label - the label assigned to the element
- Description - the content of the Description field
- Specification (for Requirements only) - the content of the Specification field

*Default = Description*

## ShowForm

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the appearance of the element is determined by the value that was set for the property ShowForm.

The possible values for this property vary for the different elements, as does the default value used. The values used are:

- Plain - No color background behind text
- Note - Color background behind text
- Pushpin - Color background plus pin icon

*Default = Note*

## ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values for this property varies for the different elements, as does the default value used. The

different values used are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is opened with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

*Default = Relative*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## SpecificationAnnotationFormat

When a requirement is displayed on a diagram, you can choose to have the element show one of the following: its name, its label, its description, the specification text for the requirement.

If you choose to display the specification text, you can use the property SpecificationAnnotationFormat to customize the text that is displayed.

The following keywords can be included in the value of the property: \$Specification (to include the specification text) and \$ID (to include the ID that was specified in the Features dialog for the requirement).

Note that this property affects the displayed text only when a requirement uses the "Note" notation style. When using the "Box" notation style, only the specification text is displayed, regardless of the value of this property.

*Default = \$Specification*

## standardPort

The standardPort metaclass contains properties that control the appearance of standard ports.

### ShowName

The ShowName property determines the text to display next to a graphic element in a diagram. For most elements, you can provide a name and a label. Your descriptive label can be useful in cases where the name itself might not be sufficient due to various constraints, such as the inability to use spaces if the name of the element is to appear in the code.

The possible values for this property vary for the different elements, as does the default value used. The possible values are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is displayed with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

Note that for some elements, such as destruction elements in sequence diagrams and timing diagrams, the property is a boolean property that hides/shows the name of the element.

*Default = NameAndType*

### ShowRedefinesLabel

The ShowRedefinesLabel property displays or hides a Redefines label for a class. This label indicates that the properties of a particular class have been redefined from a set of original properties that were derived from a base class. Redefined elements of a class can include attributes, ports, parts and association ends.

*Default = Enabled*

### ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values

are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = None*

## SingleLabelFormat

If you are using the option of combining port labels into a single one-line label (controlled by the property PortsSingleLabelLayout), you can use the SingleLabelFormat property to specify the order to use for the combined label.

*Default = \$Stereotype \$Name \$Interface*

## Stereotype

The Stereotype metaclass contains properties that relate to Stereotype elements in object model diagrams.

## Compartments

The Compartments property allows you to specify which compartments should be shown when stereotype elements are displayed in object model diagrams. For the value of this property, you enter a comma-separated list of the compartments you would like Rational Rhapsody to show. The list can include any of the elements that can be added to a stereotype.

*Default = Tag*

## QuickNavigationCategories

Graphic elements in diagrams can include a list of shortcuts that are accessible from the title bar of the element. You can use the property QuickNavigationCategories to specify the categories of items that should be included in this shortcut list and the order in which the categories should appear in the list. The value of the property should be a comma-separated list of categories. The categories that can be included are:

- Activities
- DiagramReferences
- Diagrams
- Hyperlinks
- InternalBlockDiagrams
- MainBehavior

- MainDiagram
- References
- StateCharts

If you want greater control over the content of the navigation list (such as showing/hiding specific elements within a given category, or changing the order of elements within a specific category), use the Compartments list, which can be found under "Display Options".

*Default = Hyperlinks*

## ShowName

The ShowName property allows you to specify how the element name should be shown when stereotype elements are displayed in object model diagrams.

The possible values are:

- Full\_path - The full path of the element, beginning at the level of the root package
- Relative - The relative path of the element
- Name\_only - The name of the element only
- Label - The label of the element only

*Default = Name\_only*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## Tag

The Tag metaclass contains properties that relate to Tag elements in object model diagrams.

## CompartmentDisplayFormat

Use the CompartmentDisplayFormat property to control the information that is displayed for tags when they are displayed in a graphic element's Tags compartment. The property supports the following keywords: \$name, \$type, \$value.



*Default = \$name:\$type=\$value*

## Compartments

The Compartments property allows you to specify which compartments should be shown when tag elements are displayed in object model diagrams. For the value of this property, you enter a comma-separated list of the compartments you would like Rational Rhapsody to show. The list can include any of the elements that can be added to a tag.

*Default = Blank*

## DisplayNameFormat

Use the DisplayNameFormat property to control the information that is displayed for a tag when it is included as a stand-alone element in a diagram. The property supports the following keywords: \$name, \$type, \$value.

*Default = \$name:\$type=\$value*

## QuickNavigationCategories

Graphic elements in diagrams can include a list of shortcuts that are accessible from the title bar of the element. You can use the property QuickNavigationCategories to specify the categories of items that should be included in this shortcut list and the order in which the categories should appear in the list. The value of the property should be a comma-separated list of categories. The categories that can be included are:

- Activities
- DiagramReferences
- Diagrams
- Hyperlinks
- InternalBlockDiagrams
- MainBehavior
- MainDiagram
- References
- StateCharts

If you want greater control over the content of the navigation list (such as showing/hiding specific elements within a given category, or changing the order of elements within a specific category), use the Compartments list, which can be found under "Display Options".

*Default = Hyperlinks*

## ShowName

The ShowName property allows you to specify how the element name should be shown when tag elements are displayed in object model diagrams.

The possible values are:

- Full\_path - The full path of the element, beginning at the level of the root package
- Relative - The relative path of the element
- Name\_only - The name of the element only
- Label - The label of the element only

*Default = Name\_only*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

# Type

## Compartments

The Compartments property determines which of the available compartments are displayed by default for the various types of elements. The value for this property is a comma-delimited string containing the names of the compartments that should be visible. Since the available compartments vary from element to element, it is recommended that you do not try to set the value of this property by using the Properties window or directly in the .prp file. Rather, you should use an element Display Options to set which compartments are visible, and then use the Make Default option to apply these settings at the diagram or project level for new elements of this type.

*Default = Empty MultiLine*

## QuickNavigationCategories

Graphic elements in diagrams can include a list of shortcuts that are accessible from the title bar of the element. You can use the property QuickNavigationCategories to specify the categories of items that should be included in this shortcut list and the order in which the categories should appear in the list. The value of the property should be a comma-separated list of categories. The categories that can be included

are:

- Activities
- DiagramReferences
- Diagrams
- Hyperlinks
- InternalBlockDiagrams
- MainBehavior
- MainDiagram
- References
- StateCharts

If you want greater control over the content of the navigation list (such as showing/hiding specific elements within a given category, or changing the order of elements within a specific category), use the Compartments list, which can be found under "Display Options".

*Default = Hyperlinks*

## ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values for this property varies for the different elements, as does the default value used. The different values used are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is opened with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

*Default = Name\_only*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values

are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## UseCase

The UseCase metaclass contains properties that control the appearance of use cases in object model diagrams.

### ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values for this property varies for the different elements, as does the default value used. The different values used are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is opened with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

*Default = Relative*

### ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.

- None - Do not show stereotypes in diagrams.

*Default = Label*

# OMContainers

---

Rational Rhapsody uses containers to implement to-many relations between objects. These include relationships of one object to many, or many objects to many.

The OMContainers subject contains metaclasses that contain properties that control implementing relations.

Each property in this subject includes the default value for each container type and relation type. For easier readability, the placeholder RelationType in these values represents all the other relation types that are not explicitly detailed. For example, the relation type User might have the default value of an empty string, whereas all the other relation types have the value of an empty MultiLine.

## BoundedOrdered

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed sequentially.

### Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter: \$cname-push\_back(\$item)

For maps (qualified relations), the following command inserts an item into the map based on a key: \$cname-insert(map\$keyType,\$target\*::value\_type( \$keyName,\$item))

The default is \$cname->add(\$item).

### Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector: new vector\$target\* The default is \$cname = \$CreateStatic.

### CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

*For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname: vector\$target\* \$cname() (Default = \$CType \$cname)*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*(Default = new \$CType)*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector\$target\* collection type determines the type of the variable cl on the left side of the assignment:

```
vectorClient*>* cl = new vectorClient*>(2);
```

In this case, the class Client is the \$target of vector operations.

*(Default = OMList<\$RelationTargetType>)*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item: \$cname-find(\$item) The default is \$cname->find(\$item).

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*(Default = \$CType \$cname)*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*(Default = \$cname)*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the at() operation for the container to retrieve the item at the indexed position: \$cname-at(\$index)

The default is \$cname->getAt(\$index).

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end() This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:

```
$cname-operator[]($keyName)
```

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file. (Default)

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the



Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the `#include` is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

(Default = `<oxf/omlist.h>`)

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container `$cname` to initialize it:

`$cname()` (Default)

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: `pos=0; pos$multiplicity; pos++; $cname[pos]=NULL`

## InitStatic

The InitStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to Value.

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: `vector$target*::const_iterator $iterator; $iterator=$cname-begin()`

The default is `$IterType $iterator($cname)`.

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

(Default = `$IterCreate`)

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection:

`*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*(Default = \*\$iterator)*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item: \$iterator++ (Default)

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*(Default = \$IterReset)*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*(Default = \$IterReset)*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection: \$iterator=\$cname-begin()

The default is \$iterator.reset().

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*(Default = \$IterType)*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $sname-end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*(Default = \$IterGetCurrent)*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<*, const_iterator>`, as defined in the STL.

*You can change the iterator type to one of your own choice. (Default = OMIterator<\$RelationTargetType>)*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

The default value for C++ is as follows: `for (int pos = 0; pos $multiplicity; ++pos)` The default value for C is as follows: `int pos; for (pos = 0; pos $multiplicity; ++pos)` The default value for Java is as follows: `for (int pos = 0; pos $multiplicity; pos++)`

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is `$me$name`. The default value for all other subjects is `$name`.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

The default is `$(constant)$target$reference`.

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to be removed and then call erase() to remove it. `vector$target*::iterator pos=find($cname-begin(), $cname-end(), $item); $cname-erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair$keyType,$target* p; p.second=$item; map$keyType,$target*::iterator pos=find($cname-begin(), $cname-end(),p); $cname-erase(pos)`

The default is `$cname->remove($item)`.

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items: `$cname-clear()`

The default is `$cname->removeAll()`.

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved: `$cname = $item`

## Type

The Type property specifies the type of the container as a pointer to the relation.

# BoundedUnordered

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed sequentially.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter: `$cname-push_back($item)`

For maps (qualified relations), the following command inserts an item into the map based on a key: `$cname-insert(map$keyType,$target*::value_type( $keyName,$item))` The default is `$cname->add($item)`.

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector:

`new vector$target*` The default is `$cname = $CreateStatic`.

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`:

`vector$target* $cname()`

The default is `$CType $cname($multiplicity)`.

## CreateStatic

The CreateStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to Reference.

The default is `new $CType($multiplicity)`.

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the `vector$target*` collection type determines the type of the variable `cl` on the left side of the assignment:

`vectorClient*>* cl = new vectorClient*>(2);`

In this case, the class Client is the \$target of vector operations.

*(Default = OMCollection<\$RelationTargetType>)*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

\$cname-find(\$item)

The default is \$cname->find(\$item).

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*(Default = \$CType \$cname)*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*(Default = \$cname)*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the at() operation for the container to retrieve the item at the indexed position:

\$cname-at(\$index)

The default is \$cname->getAt(\$index).

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection:

```
$cname-end()
```

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:

```
$cname-operator[]($keyName)
```

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file. (Default)

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*(Default = <oxf/omcollec.h>)*

## Init

The Init property specifies the command used to initialize the container. For example, the following

command calls the constructor for the container \$cname to initialize it:

\$cname()

The default is \$cname(\$multiplicity).

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example: pos=0; pos\$multiplicity; pos++; \$cname[pos]=NULL

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector$target*::const_iterator $iterator; $iterator=$cname-begin()
```

The default is \$IterType \$iterator(\$cname).

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*(Default = \$IterCreate)*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection:

\*\$iterator

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*(Default = \*\$iterator)*



## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*(Default = \$iterator++)*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*(Default = \$IterReset)*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*(Default = \$IterReset)*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname-begin The default is $iterator.reset().
```

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*(Default = \$IterType)*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $sname-end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as `IterGetCurrent`).

*(Default = `$IterGetCurrent`)*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<T>::const_iterator`. You can change the iterator type to one of your own choice.

*(Default = `OMIterator<$RelationTargetType>`)*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the `OMContainers.StaticArray` metaclass) can use any other property in the same metaclass. Therefore, other properties that use `$Loop` expand the specified Loop property body.

The default value for C++ is as follows: `for (int pos = 0; pos $multiplicity; ++pos)` The default value for C is as follows: `int pos; for (pos = 0; pos $multiplicity; ++pos)` The default value for Java is as follows: `for (int pos = 0; pos $multiplicity; pos++)`

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for `RiCContainers::EmbeddedScalar` is `$me$sname`. The default value for all other subjects is `$sname`.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

The default is `$(constant)$target$reference`.

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the `find()` operation to point the iterator to the item to item to

be removed and then call `erase()` to remove it. `vector$target*::iterator pos=find($cname-begin(), $cname-end(),$item);$cname-erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair `p`, whose second element, `p.second`, is the item to be removed. The `find()` operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair$keyType,$target* p; p.second=$item; map$keyType,$target*::iterator pos=find($cname-begin(), $cname-end(),p); $cname-erase(pos)`

The default is `$cname->remove($item)`.

## RemoveAll

The `RemoveAll` property specifies the command used to remove all items from the container. For example, the following command calls the `clear()` operation for the container to remove all items: `$cname-clear()`

The default is `$cname->removeAll()`.

## RemoveKey

The `RemoveKey` property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the `erase()` operation for the container, passing it the `$keyName`, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

## Set

The `Set` property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, `$cname`, is a role name of the relation itself, because there is only one class involved: `$cname = $item`

## Type

The `Type` property specifies the type of the container as a pointer to the relation.

## EmbeddedFixed

The `EmbeddedFixed` metaclass defines properties for implementing embedded fixed relations.

## Add

The `Add` property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter: `$cname-push_back($item)`

For maps (qualified relations), the following command inserts an item into the map based on a key: `$cname-insert(map$keyType,$target*::value_type( $keyName,$item))`

## Create

The `Create` property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector: `new vector$target*`

## CreateByValue

The `CreateByValue` property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`:

```
vector$target* $cname()
```

*(Default = \$CType)*

## CreateStatic

The `CreateStatic` property defines the initialization of a relation when `CG::Relation::Containment` is set to `Reference`.

## CType

The `CType` property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the `vector$target*` collection type determines the type of the variable `cl` on the left side of the assignment:

```
vectorClient*>* cl = new vectorClient*>(2);
```

In this case, the class `Client` is the `$target` of vector operations.

*(Default = \$(constant)\$target \$cname[\$multiplicity] )*

## Find

The `Find` property specifies the command used to locate an item in a container. For example, the following command calls the `Find()` operation for container `$cname` to locate the `$item`: `$cname-find($item)`

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*(Default = \$RelationTargetType \$cname[\$multiplicity])*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*(Default = &\$cname)*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the at() operation for the container to retrieve the item at the indexed position:

\$cname-at(\$index)

The default is \$RelationTargetType) &\$cname[\$index].

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end() This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:  
`$cname-operator[]($keyName)`

## **IncludeDirective**

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file. (Default)

## **IncludeFiles**

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

## **Init**

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: `$cname()`

## **InitInCtorBody**

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: `pos=0; pos$multiplicity; pos++; $cname[pos]=NULL`

## **InitStatic**

The InitStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to Value.

## **IterCreate**

The `IterCreate` property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: `vector$target*::const_iterator $iterator; $iterator=$cname-begin()`

*(Default = \$IterType \$iterator = 0;)*

## IterCreateByValue

The `IterCreateByValue` property is the same as `IterCreate`, but instantiates the iterator by value.

*(Default = \$IterCreate)*

## IterGetCurrent

The `IterGetCurrent` property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator` This value is the same as that for `IterTest` only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for `IterGetCurrent` returns a pointer to the current item in the collection.

The default is `((($RelationTargetType)&$cname[$iterator]))`.

## IterIncrement

The `IterIncrement` property specifies the code that increments the iterator. For example, the following command moves the `$iterator` ahead one item: `$iterator++` (Default)

## IterIncrementForCleanUp

The `IterIncrementForCleanUp` property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

*(Default = \$IterIncrement)*

## IterIncrementForInit

The `IterIncrementForInit` property specifies the code that increments the iterator, for certain initialization cases.

*(Default = \$IterIncrement)*

## IterInit

The `IterInit` property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default = Blank*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection: \$iterator=\$cname-begin()

The default value is \$iterator = 0.

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*(Default = \$IterType)*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname-end() With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*(Default = \$iterator < \$multiplicity)*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<, as defined in the STL. vector\$target\*::const\_iterator You can change the iterator type to one of your own choice.

*(Default = int)*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

The default value for C++ is as follows: for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)



## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$name. The default value for all other subjects is \$name.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*(Default = \$(constant)\$target\$reference)*

## Remove

The Remove property specifies the command used to remove an item from a relation. For example, the following commands call the find() operation to point the iterator to the item to be removed and then call erase() to remove it. vector\$target\*::iterator pos=find(\$cname-begin(), \$cname-end(),\$item);\$cname-erase(pos) This operation applies only to “to-many” (non-scalar) containers. For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair$keyType,$target* p; p.second=$item; map$keyType,$target*::iterator pos=find($cname-begin(),
$cname-end(),p); $cname-erase(pos)
```

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

```
$cname-clear()
```

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: \$cname-erase(\$keyName)

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

## Type

The Type property specifies the type of the container as a pointer to the relation.

## EmbeddedScalar

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed sequentially.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by \$cname, and passes the item to be added as a formal parameter: `$cname-push_back($item)`

For maps (qualified relations), the following command inserts an item into the map based on a key: `$cname-insert(map$keyType,$target*::value_type( $keyName,$item))`

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector:

```
new vector$target*
```

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector$target* $cname()
```

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector\$target\* collection type determines the type of the variable cl on the left side of the assignment:

```
vectorClient*>* cl = new vectorClient*>(2);
```

In this case, the class Client is the \$target of vector operations.

The default is \$(constant)\$target.

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item: \$cname-find(\$item)

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*(Default = \$CType\$reference \$cname)*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*(Default = &\$cname)*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The `ContainerTypes::RelationType::GetAt` property specifies a template for the body of the operation. For example, the following command generates code that calls the `at()` operation for the container to retrieve the item at the indexed position:

```
$cname-at($index)
```

## GetEnd

The `GetEnd` property specifies the command used to retrieve the last item in the container. For example, the following command calls the `end()` operation for the container to locate the last item in the collection: `$cname-end()` This property and `GetEndGenerate` were created to adhere to the standard library convention for “finding” where iteration should end. `GetEnd` is generated where `Get` is generated. The method name is defined by using the `CG::Relation::GetEnd` and `CG::Relation::GetEndGenerate` properties.

## GetKey

The `GetKey` property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator `[]`, which has been overloaded according to the STL definition for maps:  
`$cname-operator[]($keyName)`

## IncludeDirective

The `IncludeDirective` property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The `#include` directives are added to the header file.
- weak - The `#include` directives are added to the source file with forward declarations in the header file. (Default)

## IncludeFiles

The `IncludeFiles` property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the `Containment` property is set to `Reference`, a forward declaration of the container is added to the class specification file, and the `#include` is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: \$cname()

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: pos=0; pos\$multiplicity; pos++; \$cname[pos]=NULL

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector$target*::const_iterator $iterator; $iterator=$cname-begin()
```

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: \*\$iterator This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item: \$iterator++

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned

inside the `cleanUpRelations()` method.

*(Default = `$IterReset`)*

## IterIncrementForInit

The `IterIncrementForInit` property specifies the code that increments the iterator, for certain initialization cases.

*(Default = `$IterReset`)*

## IterInit

The `IterInit` property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

## IterReset

The `IterReset` property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection: `$iterator=$cname-begin()`

## IterReturnType

The `IterReturnType` property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

The default is `$(constRT)$target*`.

## IterTest

The `IterTest` property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname-end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, `IterTest` retrieves the current item in the collection (the same as `IterGetCurrent`).

## IterType

The `IterType` property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<`, as defined in the STL. `vector$target*::const_iterator` You can change the iterator type to one of your own choice.

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

The default value for C++ is as follows: for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$name. The default value for all other subjects is \$name.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*(Default = \$CType\*)*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. vector\$target\*::iterator pos=find(\$cname-begin(), \$cname-end(),\$item);\$cname-erase(pos) This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. pair\$keyType,\$target\* p; p.second=\$item; map\$keyType,\$target\*::iterator pos=find(\$cname-begin(), \$cname-end(),p); \$cname-erase(pos)

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items: \$cname-clear()

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the

container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed:  
`$cname-erase($keyName)`

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved: `$cname = $item`

## Type

The Type property specifies the type of the container as a pointer to the relation.

## Fixed

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed sequentially.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by \$cname, and passes the item to be added as a formal parameter: `$cname-push_back($item)`

For maps (qualified relations), the following command inserts an item into the map based on a key:  
`$cname-insert(map$keyType,$target*::value_type( $keyName,$item))`

The default is `$cname->add($item)`.

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector: `new vector$target*`

The default is `$cname = $CreateStatic`.

## CreateByValue



The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname: `vector$target* $cname()`

The default is `$CType $cname($multiplicity)`.

## CreateStatic

The CreateStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to Reference.

The default is `new $CType($multiplicity)`.

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the `vector$target*` collection type determines the type of the variable `cl` on the left side of the assignment:

```
vectorClient*>* cl = new vectorClient*>(2);
```

In this case, the class `Client` is the \$target of vector operations.

The default is `OMCollection<$RelationTargetType>`.

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the `Find()` operation for container \$cname to locate the \$item: `$cname-find($item)`

The default is `$cname->find($item)`.

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*(Default = \$CType \$cname)*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$name The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$name is replaced with the name of the container, which is the role name for the relation.

*(Default = \$name)*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the at() operation for the container to retrieve the item at the indexed position:

\$name-at(\$index)

The default is \$name->getAt(\$index).

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection:

\$name-end()

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:

\$name-operator[](\$keyName)

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.

- weak - The #include directives are added to the source file with forward declarations in the header file. (Default)

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

(Default = <oxf/omcollec.h>)

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

```
$cname()
```

The default is \$cname(\$multiplicity).

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example: pos=0; pos\$multiplicity; pos++; \$cname[pos]=NULL

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector$target*::const_iterator $iterator; $iterator=$cname-begin()
```

The default is \$IterType \$iterator(\$sname).

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*(Default = \$IterCreate)*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection:

\*\$iterator

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*(Default = \*\$iterator)*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

\$iterator++ (Default)

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*(Default = \$IterReset)*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*(Default = \$IterReset)*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the

return value of the relation `getRelation>()` generated operation.

## IterReset

The `IterReset` property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname-begin()
```

The default is `$iterator.reset()`.

## IterReturnType

The `IterReturnType` property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*(Default = `$IterType`)*

## IterTest

The `IterTest` property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname-end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, `IterTest` retrieves the current item in the collection (the same as `IterGetCurrent`).

*(Default = `$IterGetCurrent`)*

## IterType

The `IterType` property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<`, as defined in the STL. `vector$target*::const_iterator` You can change the iterator type to one of your own choice.

*(Default = `OMIterator<$RelationTargetType>`)*

## Loop

The `Loop` property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the `OMContainers.StaticArray` metaclass) can use any other property in the same metaclass. Therefore, other properties that use `$Loop` expand the specified `Loop` property body.

The default value for C++ is as follows: `for (int pos = 0; pos $multiplicity; ++pos)` The default value for C is as follows: `int pos; for (pos = 0; pos $multiplicity; ++pos)` The default value for Java is as follows: `for (int pos = 0; pos $multiplicity; pos++)`

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for `RiCContainers::EmbeddedScalar` is `$me$name`. The default value for all other subjects is `$name`.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

The default value is `$(constant)$target$reference`.

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the `find()` operation to point the iterator to the item to item to be removed and then call `erase()` to remove it. `vector$target*::iterator pos=find($cname-begin(), $cname-end(),$item);$cname-erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair `p`, whose second element, `p.second`, is the item to be removed. The `find()` operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair$keyType,$target* p; p.second=$item; map$keyType,$target*::iterator pos=find($cname-begin(), $cname-end(),p); $cname-erase(pos)`

The default value is `$cname->remove($item)`.

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the `clear()` operation for the container to remove all items:

```
$cname-clear()
```

The default value is `$cname->removeAll()`.

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the `erase()` operation for the container, passing it the `$keyName`, which maps into a dictionary used to locate the item to be removed:

```
$cname-erase($keyName)
```

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

## Type

The Type property specifies the type of the container as a pointer to the relation.

## General

The General metaclass contains properties that enable you to set the directives and include files for the container.

### ContainerDirectives

The ContainerDirectives property specifies the preprocessor directives that are necessary when compiling code that uses a particular container library.

No additional directives are required when you use OMContainers.

### ContainerIncludes

The ContainerIncludes property specifies header files that must be included when you use a particular container library.

For example, when you use STLContainers, the following string causes one #include directive per container to be added to generated files (such as #include string): string, algorithm, vector, list, map, iterator Whether the #include directives are added to source or header files depends on the value of the IncludeDirective property.

## Qualified

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed sequentially.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter: `$cname-push_back($item)`

For maps (qualified relations), the following command inserts an item into the map based on a key: `$cname-insert(map$keyType,$target*::value_type( $keyName,$item))`

The default is `$cname->add($keyName,$item)`.

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector:

```
new vector$target*
```

*(Default = \$cname = new \$CType)*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`:

```
vector$target* $cname()
```

*(Default = \$CType \$cname)*

## CreateStatic

The CreateStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to `Reference`.

*(Default = new \$CType)*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the `vector$target*` collection type determines the type of the variable `cl` on the left side of the assignment:



```
vectorClient*>* cl = new vectorClient*>(2);
```

In this case, the class Client is the \$target of vector operations.

*(Default = OMMMap<\$keyType, \$RelationTargetType>)*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname-find($item)
```

The default is \$cname->find(\$item).

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*(Default = \$CType \$cname)*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*(Default = \$cname)*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the at() operation for the container to retrieve the item at the indexed position:

```
$cname-at($index)
```

The default is \$cname->getAt(\$index).

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection:

```
$cname-end()
```

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:

```
$cname-operator[]($keyName)
```

The default is `$cname->getKey($keyName)`.

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file. (Default)

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*(Default = <oxf/ommap.h>)*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

```
$cname() (Default)
```

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example: pos=0; pos\$multiplicity; pos++; \$cname[pos]=NULL

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector$target*::const_iterator $iterator; $iterator=$cname-begin()
```

The default is \$IterType \$iterator(\$cname).

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*(Default = \$IterCreate)*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection:

```
*$iterator
```

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*(Default = \*\$iterator)*

## **IterIncrement**

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

`$iterator++` (Default)

## **IterIncrementForCleanUp**

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

*(Default = \$IterReset)*

## **IterIncrementForInit**

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*(Default = \$IterReset)*

## **IterInit**

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

## **IterReset**

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection:

`$iterator=$cname-begin()`

The default is `$iterator.reset()`.

## **IterReturnType**

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*(Default = \$IterType)*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $name-end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*(Default = \$IterGetCurrent)*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<T>::const_iterator`. You can change the iterator type to one of your own choice.

*(Default = OMIterator<\$RelationTargetType>)*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

The default value for C++ is as follows: `for (int pos = 0; pos $multiplicity; ++pos)` The default value for C is as follows: `int pos; for (pos = 0; pos $multiplicity; ++pos)` The default value for Java is as follows: `for (int pos = 0; pos $multiplicity; pos++)`

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is `$me$name`. The default value for all other subjects is `$name`.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*(Default = \$(constant)\$target\$reference)*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to be removed and then call erase() to remove it. `vector$target*::iterator pos=find($cname-begin(), $cname-end(), $item); $cname-erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair `p`, whose second element, `p.second`, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair$keyType,$target* p; p.second=$item; map$keyType,$target*::iterator pos=find($cname-begin(),
$cname-end(),p); $cname-erase(pos)
```

The default is `$cname->remove($item)`.

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

```
$cname-clear()
```

The default is `$cname->removeAll()`.

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the `$keyName`, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

The default is `$cname->remove($keyName)`.

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, `$cname`, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

## Type

The Type property specifies the type of the container as a pointer to the relation.

## Scalar

The Scalar metaclass defines properties for implementing scalar relations.

### Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter: `$cname-push_back($item)`

For maps (qualified relations), the following command inserts an item into the map based on a key: `$cname-insert(map$keyType,$target*::value_type( $keyName,$item))`

### Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector:

```
new vector$target*
```

### CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`:

```
vector$target* $cname()
```

### CreateStatic

The CreateStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to Reference.

### CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the `vector$target*` collection type determines the type of the variable `cl` on the left side of the assignment:

```
vectorClient*>* cl = new vectorClient*>(2);
```

In this case, the class Client is the \$target of vector operations.

The default is \$(constant)\$target\$reference.

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname-find($item)
```

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*(Default = \$CType \$cname)*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*(Default = \$cname)*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the at() operation for the container to retrieve the item at the indexed position:

```
$cname-at($index)
```

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example,



the following command calls the `end()` operation for the container to locate the last item in the collection:  
`$cname-end()` This property and `GetEndGenerate` were created to adhere to the standard library convention for “finding” where iteration should end. `GetEnd` is generated where `Get` is generated. The method name is defined by using the `CG::Relation::GetEnd` and `CG::Relation::GetEndGenerate` properties.

## GetKey

The `GetKey` property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator `[]`, which has been overloaded according to the STL definition for maps:  
`$cname-operator[]($keyName)`

## IncludeDirective

The `IncludeDirective` property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- `strong` - The `#include` directives are added to the header file.
- `weak` - The `#include` directives are added to the source file with forward declarations in the header file. (Default)

## IncludeFiles

The `IncludeFiles` property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the `Containment` property is set to `Reference`, a forward declaration of the container is added to the class specification file, and the `#include` is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

## Init

The `Init` property specifies the command used to initialize the container. For example, the following command calls the constructor for the container `$cname` to initialize it:

`$cname()`

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos$multiplicity; pos++; $cname[pos]=NULL
```

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector$target*::const_iterator $iterator; $iterator=$cname-begin()
```

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator` This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned

inside the `cleanUpRelations()` method.

## IterIncrementForInit

The `IterIncrementForInit` property specifies the code that increments the iterator, for certain initialization cases.

## IterInit

The `IterInit` property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

## IterReset

The `IterReset` property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname-begin()
```

## IterReturnType

The `IterReturnType` property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

The default is `$(constRT)$target$reference`.

## IterTest

The `IterTest` property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname-end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, `IterTest` retrieves the current item in the collection (the same as `IterGetCurrent`).

## IterType

The `IterType` property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<`, as defined in the STL.

```
vector$target*::const_iterator
```

 You can change the iterator type to one of your own choice.

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

The default value for C++ is as follows: for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$name. The default value for all other subjects is \$name.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*(Default = \$CType)*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. vector\$target\*::iterator pos=find(\$cname-begin(), \$cname-end(),\$item);\$cname-erase(pos) This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed.

The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair$keyType,$target* p; p.second=$item; map$keyType,$target*::iterator pos=find($cname-begin(),
$cname-end(),p); $cname-erase(pos)
```

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

```
$cname-clear()
```

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: \$cname-erase(\$keyName)

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

```
(Default = $cname = $item)
```

## Type

The Type property specifies the type of the container as a pointer to the relation.

## StaticArray

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed sequentially.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter: \$cname-push\_back(\$item)

For maps (qualified relations), the following command inserts an item into the map based on a key: \$cname-insert(map\$keyType,\$target\*::value\_type( \$keyName,\$item))

The default is as follows: \$Loop { if (!\$cname[pos]) { \$cname[pos] = \$item; break; } }

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector:

```
new vector$target*
```

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector$target* $cname()
```

*(Default = \$CType)*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector\$target\* collection type determines the type of the variable cl on the left side of the assignment:

```
vectorClient*>* cl = new vectorClient*>(2);
```

In this case, the class Client is the \$target of vector operations.

*(Default = \$RelationTargetType \$cname[\$multiplicity])*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname-find($item)
```

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*(Default = \$CType)*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*(Default = \$cname)*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the at() operation for the container to retrieve the item at the indexed position:

\$cname-at(\$index)

*(Default = \$cname[\$index])*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection:

\$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:

\$cname-operator[](\$keyName)

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file. (Default)

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

```
$cname()
```

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos$multiplicity; pos++; $cname[pos]=NULL
```

The default is as follows: `$Loop { $cname[pos] = NULL; }`

## InitStatic

The InitStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to



Value.

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector$target*::const_iterator $iterator; $iterator=$cname-begin()
```

*(Default = \$IterType \$iterator = 0;)*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*(Default = \$IterCreate)*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator` This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*(Default = \$cname[\$iterator])*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*(Default = \$iterator++)*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

*(Default = \$IterIncrement)*

## IterIncrementForInit

The `IterIncrementForInit` property specifies the code that increments the iterator, for certain initialization cases.

*(Default = `$IterIncrement`)*

## IterInit

The `IterInit` property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

## IterReset

The `IterReset` property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname-begin()
```

*(Default = `$iterator = 0`)*

## IterReturnType

The `IterReturnType` property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*(Default = `$IterType`)*

## IterTest

The `IterTest` property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname-end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, `IterTest` retrieves the current item in the collection (the same as `IterGetCurrent`).

The default is `($iterator < $multiplicity) && $cname[$iterator]`.

## IterType

The `IterType` property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<`, as defined in the STL. `vector$target*::const_iterator` You can change the iterator type to one of your own choice.

*(Default = `int`)*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

The default value for C++ is as follows: `for (int pos = 0; pos < $multiplicity; ++pos)` The default value for C is as follows: `int pos; for (pos = 0; pos < $multiplicity; ++pos)` The default value for Java is as follows: `for (int pos = 0; pos < $multiplicity; pos++)`

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is `$me$name`. The default value for all other subjects is `$name`.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*(Default = \$(constant)\$target\$reference)*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the `find()` operation to point the iterator to the item to be removed and then call `erase()` to remove it. `vector$target*::iterator pos=find($cname-begin(), $cname-end(),$item);$cname-erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair `p`, whose second element, `p.second`, is the item to be removed. The `find()` operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair$keyType,$target* p; p.second=$item; map$keyType,$target*::iterator pos=find($cname-begin(), $cname-end(),p); $cname-erase(pos)`

The default is as follows: `($Loop { if ($cname[pos] == $item) { $cname[pos] = NULL; } })`

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the `clear()` operation for the container to remove all items:

`$cname-clear()`

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed:

```
$cname-erase($keyName)
```

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

## SetAt

The SetAt property specifies how code is generated for the body of the mutator for a scalar container.

The default is \$cname[\$index] = \$item.

## Type

The Type property specifies the type of the container as a pointer to the relation.

# UnboundedOrdered

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed sequentially.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter: \$cname-push\_back(\$item)

For maps (qualified relations), the following command inserts an item into the map based on a key: \$cname-insert(map\$keyType,\$target\*::value\_type( \$keyName,\$item))

The default is \$cname->add(\$item).

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector: new vector\$target\*

The default is \$cname = \$CreateStatic.

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector$target* $cname()
```

*(Default = \$CType \$cname)*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*(Default = new \$CType)*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector\$target\* collection type determines the type of the variable cl on the left side of the assignment:

```
vectorClient*>* cl = new vectorClient*>(2);
```

In this case, the class Client is the \$target of vector operations.

*(Default = OMList<\$RelationTargetType>)*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

`$cname-find($item)`

The default is `$cname->find($item)`.

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*(Default = \$CType \$cname)*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: `$me$cname` The variable `$me` is replaced with the object context variable as specified by the Me property. The variable `$cname` is replaced with the name of the container, which is the role name for the relation.

*(Default = \$cname)*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the `at()` operation for the container to retrieve the item at the indexed position:

`$cname-at($index)`

The default is `$cname->getAt($index)`.

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the `end()` operation for the container to locate the last item in the collection:

`$cname-end()`

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the `CG::Relation::GetEnd` and `CG::Relation::GetEndGenerate` properties.

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:  
\$cname-operator[](\$keyName)

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file. (Default)

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

(Default = <oxf/omlist.h>)

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

\$cname()

The default is \$cname().

## InitInCtorBody

The `InitInCtorBody` property specifies code to initialize containers for arrays inside the class constructor body. For example: `pos=0; pos$multiplicity; pos++; $cname[pos]=NULL`

## InitStatic

The `InitStatic` property defines the initialization of a relation when `CG::Relation::Containment` is set to `Value`.

## IterCreate

The `IterCreate` property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector$target*::const_iterator $iterator; $iterator=$cname-begin()
```

The default is `$IterType $iterator($cname)`.

## IterCreateByValue

The `IterCreateByValue` property is the same as `IterCreate`, but instantiates the iterator by value.

*(Default = \$IterCreate)*

## IterGetCurrent

The `IterGetCurrent` property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection:

```
*$iterator
```

This value is the same as that for `IterTest` only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for `IterGetCurrent` returns a pointer to the current item in the collection.

*(Default = \*\$iterator)*

## IterIncrement

The `IterIncrement` property specifies the code that increments the iterator. For example, the following command moves the `$iterator` ahead one item:

```
$iterator++ (Default)
```

## IterIncrementForCleanup



The `IterIncrementForCleanUp` property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

*(Default = `$IterReset`)*

## IterIncrementForInit

The `IterIncrementForInit` property specifies the code that increments the iterator, for certain initialization cases.

*(Default = `$IterReset`)*

## IterInit

The `IterInit` property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

## IterReset

The `IterReset` property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname-begin()
```

The default is `$iterator.reset()`.

## IterReturnType

The `IterReturnType` property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*(Default = `$IterType`)*

## IterTest

The `IterTest` property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname-end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, `IterTest` retrieves the current item in the collection (the same as `IterGetCurrent`).

*(Default = `$IterGetCurrent`)*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<, as defined in the STL. vector\$target\*::const\_iterator You can change the iterator type to one of your own choice.

*(Default = OMIterator<\$RelationTargetType>)*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

The default value for C++ is as follows: for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$name. The default value for all other subjects is \$name.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*(Default = \$(constant)\$target\$reference)*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to be removed and then call erase() to remove it. vector\$target\*::iterator pos=find(\$cname-begin(), \$cname-end(),\$item);\$cname-erase(pos) This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed.

The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. pair\$keyType,\$target\* p; p.second=\$item; map\$keyType,\$target\*::iterator pos=find(\$cname-begin(), \$cname-end(),p); \$cname-erase(pos)

The default is \$cname->remove(\$item).

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

```
$cname-clear()
```

The default is `$cname->removeAll()`.

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed:

```
$cname-erase($keyName)
```

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

## SetAt

The SetAt property specifies how code is generated for the body of the mutator for a scalar container.

The default is `$cname[$index] = $item`.

## Type

The Type property specifies the type of the container as a pointer to the relation.

# UnboundedUnordered

Defines properties for implementing relations whose multiplicity is unbounded (\*) and that should be accessed randomly.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter: `$cname-push_back($item)`

For maps (qualified relations), the following command inserts an item into the map based on a key: `$cname-insert(map$keyType,$target*::value_type( $keyName,$item))`

The default is `$cname->add($item)`.

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector:

```
new vector$target*
```

*(Default = \$cname = new \$CType)*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`:

```
vector$target* $cname()
```

*(Default = \$CType \$cname)*

## CreateStatic

The CreateStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to Reference.

*(Default = new \$CType)*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the `vector$target*` collection type determines the type of the variable `cl` on the left side of the assignment:

```
vectorClient*>* cl = new vectorClient*>(2);
```

In this case, the class Client is the \$target of vector operations.

*(Default = OMCollection<\$RelationTargetType>)*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

`$cname-find($item)`

The default is `$cname->find($item)`.

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*(Default = \$CType \$cname)*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*(Default = \$cname)*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the at() operation for the container to retrieve the item at the indexed position:

`$cname-at($index)`

The default is `$cname->getAt($index)`.

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end() This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps: \$cname-operator[](\$keyName)

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file. (Default)

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*(Default = <oxf/omcollec.h>)*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: \$cname() (Default)

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example: pos=0; pos\$multiplicity; pos++; \$name[pos]=NULL

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector$target*::const_iterator $iterator; $iterator=$name-begin()
```

The default is \$IterType \$iterator(\$name).

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*(Default = \$IterCreate)*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: \*\$iterator This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*(Default = \*\$iterator)*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*(Default = \$iterator++)*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*(Default = \$IterReset)*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*(Default = \$IterReset)*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname-begin()
```

*(Default = \$iterator.reset())*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*(Default = \$IterType)*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname-end() With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).



*(Default = \$IterGetCurrent)*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<T>::const_iterator`, as defined in the STL. `vector$target*::const_iterator` You can change the iterator type to one of your own choice.

*(Default = OMIterator<\$RelationTargetType>)*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the `OMContainers.StaticArray` metaclass) can use any other property in the same metaclass. Therefore, other properties that use `$Loop` expand the specified Loop property body.

The default value for C++ is as follows: `for (int pos = 0; pos $multiplicity; ++pos)` The default value for C is as follows: `int pos; for (pos = 0; pos $multiplicity; ++pos)` The default value for Java is as follows: `for (int pos = 0; pos $multiplicity; pos++)`

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for `RiCContainers::EmbeddedScalar` is `$me$name`. The default value for all other subjects is `$name`.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*(Default = \$(constant)\$target\$reference)*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the `find()` operation to point the iterator to the item to be removed and then call `erase()` to remove it. `vector$target*::iterator pos=find($name-begin(), $name-end(),$item);$name-erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair `p`, whose second element, `p.second`, is the item to be removed. The `find()` operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair$keyType,$target* p; p.second=$item; map$keyType,$target*::iterator pos=find($cname-begin(),
$cname-end(),p); $cname-erase(pos)
```

The default is `$cname->remove($item)`.

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the `clear()` operation for the container to remove all items:

```
$cname-clear()
```

The default is `$cname->removeAll()`.

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the `erase()` operation for the container, passing it the `$keyName`, which maps into a dictionary used to locate the item to be removed:

```
$cname-erase($keyName)
```

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, `$cname`, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

## Type

The Type property specifies the type of the container as a pointer to the relation.

## User

The User metaclass defines properties for user-defined implementations of relations.

You can create your own implementations for relations by defining a new set of properties under the User metaclass. Once these are defined, you can give them permanent status by manually saving them in the `factory.prp` file under any other name, for example `MyFaves`.

To complete their installation, you must add the new name as an enumerated value to the `CG::Relation::Implementation` property.

You can create your own implementations for relations by defining a new set of properties under the User metaclass. Once these are defined, you can give them permanent status by manually saving them in the factory.prp file under any other name, for example MyFaves. To complete their installation, you must add the new name as an enumerated value to the CG::Relation::Implementation property.

For example, you can change the definition of the Implementation property as follows:

```
Subject CG Metaclass Relation Property Implementation Enum "Default,Scalar,Fixed,BoundedOrdered,
BoundedUnordered,UnboundedOrdered, UnboundedUnordered,Qualified,MyFaves, User" "Default" end
end
```

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter: \$cname-push\_back(\$item)

For maps (qualified relations), the following command inserts an item into the map based on a key: \$cname-insert(map\$keyType,\$target\*::value\_type( \$keyName,\$item))

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector:

```
new vector$target*
```

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector$target* $cname()
```

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector\$target\* collection type determines the type of the variable cl on the left side of the assignment:

```
vectorClient*>* cl = new vectorClient*>(2);
```

In this case, the class Client is the \$target of vector operations.

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname-find($item)
```

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the at() operation for the container to retrieve the item at the indexed position:

```
$cname-at($index)
```

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end() This property and GetEndGenerate were created to adhere to the standard library

convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:  
\$cname-operator[](\$keyName)

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file. (Default)

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

\$cname()

## InitInCtorBody

The `InitInCtorBody` property specifies code to initialize containers for arrays inside the class constructor body.

For example: `pos=0; pos$multiplicity; pos++; $cname[pos]=NULL`

## InitStatic

The `InitStatic` property defines the initialization of a relation when `CG::Relation::Containment` is set to `Value`.

## IterCreate

The `IterCreate` property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector$target*::const_iterator $iterator; $iterator=$cname-begin()
```

## IterCreateByValue

The `IterCreateByValue` property is the same as `IterCreate`, but instantiates the iterator by value.

## IterGetCurrent

The `IterGetCurrent` property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator` This value is the same as that for `IterTest` only when you use the Rational Rhapsody framework `OMContainers` container set. When you use the STL container set, an operation substituted for `IterGetCurrent` returns a pointer to the current item in the collection.

## IterIncrement

The `IterIncrement` property specifies the code that increments the iterator. For example, the following command moves the `$iterator` ahead one item:

```
$iterator++
```

## IterIncrementForCleanUp

The `IterIncrementForCleanUp` property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

## IterIncrementForInit

The `IterIncrementForInit` property specifies the code that increments the iterator, for certain initialization cases.

## IterInit

The `IterInit` property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

## IterReset

The `IterReset` property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname-begin()
```

## IterReturnType

The `IterReturnType` property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

## IterTest

The `IterTest` property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname-end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, `IterTest` retrieves the current item in the collection (the same as `IterGetCurrent`).

## IterType

The `IterType` property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<`, as defined in the STL.

```
vector$target*::const_iterator
```

## Loop

The `Loop` property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the `OMContainers.StaticArray` metaclass) can use any other property in the same metaclass. Therefore, other properties that use `$Loop` expand the specified `Loop` property body.

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for `RiCContainers::EmbeddedScalar` is `$me$name`. The default value for all other subjects is `$name`.

## RelationTargetType

The `RelationTargetType` property specifies the return type for relation getters, as part of the relation implementation properties.

## Remove

The `Remove` property specifies the command used to remove an item from a relation.

For example, the following commands call the `find()` operation to point the iterator to the item to item to be removed and then call `erase()` to remove it. `vector$target*::iterator pos=find($cname-begin(), $cname-end(),$item);$cname-erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair `p`, whose second element, `p.second`, is the item to be removed. The `find()` operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair$keyType,$target* p; p.second=$item; map$keyType,$target*::iterator pos=find($cname-begin(),
$cname-end(),p); $cname-erase(pos)
```

## RemoveAll

The `RemoveAll` property specifies the command used to remove all items from the container. For example, the following command calls the `clear()` operation for the container to remove all items:

```
$cname-clear()
```

## RemoveKey

The `RemoveKey` property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the `erase()` operation for the container, passing it the `$keyName`, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

## Set

The `Set` property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, `$cname`, is a role name of the relation itself, because there is only one class involved:



\$cname = \$item

## Type

The Type property specifies the type of the container as a pointer to the relation.

# OMCorba2CorbaContainers

---

Rational Rhapsody uses containers to implement to-many relations between objects. These include relationships of one object to many, or many objects to many.

The OMCorba2CorbaContainers subject contains metaclasses that contain properties that control the implementing of relations.

Each property in this section includes the default value for each container type and relation type. For easier readability, the placeholder RelationType in these values represents all the other relation types that are not explicitly detailed. For example, the relation type User might have the default value of an empty string, whereas all the other relation types have the value of an empty MultiLine.

## BoundedOrdered

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed sequentially.

### Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$name, and passes the item to be added as a formal parameter:

```
$name->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$name->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = Empty string*

### Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = Empty string*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector<$target*> $cname()
```

*Default = Empty string*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default =*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = sequence<\$RelationTargetType>*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default = Empty string*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

`$cname->at($index)`

*Default = Empty string*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:

`$cname-operator[]($keyName)`

*Default =*

## **IncludeDirective**

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## **IncludeFiles**

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = Empty string*

## **Init**

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

```
$cname()
```

*Default = Empty string*

## **InitInCtorBody**

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL
```

*Default = Empty string*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = Empty string*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = Empty string*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = Empty string*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default = Empty string*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = Empty string*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = Empty string*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = Empty string*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default =*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end()

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With

OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = Empty string*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<>, as defined in the STL.

vector<\$target\*>::const\_iterator You can change the iterator type to one of your own choice.

*Default = \$(target)Seq*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$name. The default value for all other subjects is \$name.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$target*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to be removed and then call erase() to remove it. vector<\$target\*>::iterator pos=find(\$name>begin(), \$name>end(),\$item);\$name>erase(pos) This operation applies only to "to-many" (non-scalar)



containers.

For maps (qualified relations), the following commands create a pair `p`, whose second element, `p.second`, is the item to be removed. The `find()` operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find(
$name->begin(), $name->end(),p); $name->erase(pos)
```

*Default = Empty string*

## RemoveAll

The `RemoveAll` property specifies the command used to remove all items from the container. For example, the following command calls the `clear()` operation for the container to remove all items:

```
$name->clear()
```

*Default = Empty string*

## RemoveKey

The `RemoveKey` property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the `erase()` operation for the container, passing it the `$keyName`, which maps into a dictionary used to locate the item to be removed:

```
$name-erase($keyName)
```

*Default =*

## Set

The `Set` property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, `$name`, is a role name of the relation itself, because there is only one class involved:

```
$name = $item
```

*Default =*

## Type

The `Type` property specifies the type of the container as a pointer to the relation.

*Default =*

## BoundedUnordered

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed randomly.

### Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = Empty string*

### Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = Empty string*

### CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`:

```
vector<$target*> $cname()
```

*Default = Empty string*

### CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default =*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = sequence<\$RelationTargetType>*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default = Empty string*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

```
$cname->at($index)
```

*Default = Empty string*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps: \$cname-operator[](\$keyName)

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = Empty string*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

```
$cname()
```

*Default = Empty string*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL
```

*Default = Empty string*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the

container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = Empty string*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = Empty string*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = Empty string*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default = Empty string*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

*Default = Empty string*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = Empty string*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = Empty string*

## IterReturntype

The IterReturntype property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default =*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()`

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = Empty string*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<>`, as defined in the STL.

`vector<$target*>::const_iterator` You can change the iterator type to one of your own choice.

*Default = \$(target)Seq*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$name. The default value for all other subjects is \$name.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$target*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname>erase(pos)` This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

`pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find($cname->begin(), $cname->end(),p); $cname->erase(pos)`

*Default = Empty string*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:



`$cname->clear()`

*Default = Empty string*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed:  
`$cname-erase($keyName)`

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

`$cname = $item`

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

# EmbeddedFixed

Defines properties for implementing embedded fixed relations.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter:

`$cname->push_back($item)`

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default =*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector:

```
new vector<$target*>
```

*Default =*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector<$target*> $cname()
```

*Default =*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default =*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default =*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default =*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default =*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default =*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

```
$cname->at($index)
```

*Default =*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection:

\$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:  
\$cname-operator[](\$keyName)

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default =*

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default =*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

```
$cname()
```

*Default =*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL
```

*Default =*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default =*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default =*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: \*\$iterator

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default =*

## **IterIncrement**

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default =*

## **IterIncrementForCleanUp**

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default =*

## **IterIncrementForInit**

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default =*

## **IterInit**

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

*Default =*

## **IterReset**

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default =*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default =*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end()

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default =*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<>, as defined in the STL.

vector<\$target\*>::const\_iterator You can change the iterator type to one of your own choice.

*Default =*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default =*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname>erase(pos)` This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find(
$name->begin(), $name->end(),p); $name->erase(pos)
```

*Default =*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

```
$cname->clear()
```

*Default =*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.



For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## EmbeddedScalar

Defines properties for implementing embedded scalar (one-to-one) relations.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default =*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector:

```
new vector<$target*>
```

*Default =*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector<$target*> $cname()
```

*Default* =

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default* =

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default* =

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default* =

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default =*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$name

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$name is replaced with the name of the container, which is the role name for the relation.

*Default =*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

```
$name->at($index)
```

*Default =*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$name-end()

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:

```
$name-operator[]($keyName)
```

*Default =*

## **IncludeDirective**

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default =*

## **IncludeFiles**

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default =*

## **Init**

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

```
$cname()
```

*Default =*

## **InitInCtorBody**

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL
```

*Default =*

## **InitStatic**

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## **IterCreate**

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default =*

## **IterCreateByValue**

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default =*

## **IterGetCurrent**

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default =*

## **IterIncrement**

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default =*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default =*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default =*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the iterator's begin() operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default =*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default =*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end()

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With

OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default =*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<>, as defined in the STL.

vector<\$target\*>::const\_iterator You can change the iterator type to one of your own choice.

*Default =*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default =*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to be removed and then call erase() to remove it. vector<\$target\*>::iterator pos=find(\$cname>begin(), \$cname>end(),\$item);\$cname>erase(pos) This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair `p`, whose second element, `p.second`, is the item to be removed. The `find()` operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find(
$name->begin(), $name->end(),p); $name->erase(pos)
```

*Default =*

## RemoveAll

The `RemoveAll` property specifies the command used to remove all items from the container. For example, the following command calls the `clear()` operation for the container to remove all items:

```
$name->clear()
```

*Default =*

## RemoveKey

The `RemoveKey` property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the `erase()` operation for the container, passing it the `$keyName`, which maps into a dictionary used to locate the item to be removed:

```
$name-erase($keyName)
```

*Default =*

## Set

The `Set` property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, `$name`, is a role name of the relation itself, because there is only one class involved:

```
$name = $item
```

*Default =*

## Type

The `Type` property specifies the type of the container as a pointer to the relation.

*Default =*



## Fixed

Defines properties for implementing relations of fixed size.

### Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = Empty string*

### Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = Empty string*

### CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`:

```
vector<$target*> $cname()
```

*Default = Empty string*

### CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default =*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = sequence<\$RelationTargetType>*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default = Empty string*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

```
$cname->at($index)
```

*Default = Empty string*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps: \$cname-operator[](\$keyName)

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = Empty string*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

```
$cname()
```

*Default = Empty string*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL
```

*Default = Empty string*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the

container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = Empty string*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = Empty string*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = Empty string*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default = Empty string*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = Empty string*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = Empty string*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = Empty string*

## IterReturntype

The IterReturntype property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default =*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()`

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = Empty string*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<>`, as defined in the STL.

```
vector<$target*>::const_iterator
```

 You can change the iterator type to one of your own choice.

*Default = \$(target)Seq*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$name. The default value for all other subjects is \$name.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$target*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname>erase(pos)` This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

`pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find($cname->begin(), $cname->end(),p); $cname->erase(pos)`

*Default = Empty string*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

\$cname->clear()

*Default = Empty string*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed:  
\$cname-erase(\$keyName)

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

\$cname = \$item

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## General

Defines properties that enable you to set the directives and include files for the container.

## ContainerDirectives

The ContainerDirectives property specifies the preprocessor directives that are necessary when compiling code that uses a particular container library.

No additional directives are required when you use OMContainers.

*Default = Empty MultiLine*



## ContainerIncludes

The ContainerIncludes property specifies header files that must be included when you use a particular container library.

For example, when you use STLContainers, the following string causes one #include directive per container to be added to generated files (such as #include string): string, algorithm, vector, list, map, iterator Whether the #include directives are added to source or header files depends on the value of the IncludeDirective property.

*Default = Empty string*

## Qualified

Defines properties for implementing qualified relations, which are accessed by way of a key.

### Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = Empty string*

### Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = Empty string*

### CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector<$target*> $cname()
```

*Default = Empty string*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default =*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = sequence<\$RelationTargetType>*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default = Empty string*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$name

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$name is replaced with the name of the container, which is the role name for the relation.

*Default = \$name*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

\$name->at(\$index)

*Default = Empty string*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$name-end()

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps: \$name-operator[](\$keyName)

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## **IncludeFiles**

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = Empty string*

## **Init**

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

```
$cname()
```

*Default = Empty string*

## **InitInCtorBody**

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL
```

*Default = Empty string*

## **InitStatic**

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<${target*}>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = Empty string*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = Empty string*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = Empty string*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default = Empty string*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = Empty string*

## **IterIncrementForInit**

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = Empty string*

## **IterInit**

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

*Default =*

## **IterReset**

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = Empty string*

## **IterReturnType**

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default =*

## **IterTest**

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end()

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = Empty string*

## **IterType**

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<>, as defined in the STL.

vector<\$target\*>::const\_iterator You can change the iterator type to one of your own choice.

*Default = \$(target)Seq*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$name. The default value for all other subjects is \$name.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$target*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to be removed and then call erase() to remove it. vector<\$target\*>::iterator pos=find(\$name>begin(), \$name>end(), \$item); \$name>erase(pos) This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

pair<\$keyType, \$target\*> p; p.second=\$item; map<\$keyType, \$target\*>::iterator pos=find(

```
$cname->begin(), $cname->end(),p); $cname->erase(pos)
```

*Default = Empty string*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

```
$cname->clear()
```

*Default = Empty string*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed:

```
$cname-erase($keyName)
```

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

# Scalar

Defines properties for implementing scalar relations.



## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default* =

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector:

```
new vector<$target*>
```

*Default* =

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`:

```
vector<$target*> $cname()
```

*Default* =

## CreateStatic

The CreateStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to Reference.

*Default* =

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = \$RelationTargetType*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default =*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

\$cname->at(\$index)

*Default =*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection:  
\$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:  
\$cname-operator[](\$keyName)

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default =*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

```
$cname()
```

*Default =*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL
```

*Default =*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default =*

## IterCreateByValue

The `IterCreateByValue` property is the same as `IterCreate`, but instantiates the iterator by value.

*Default =*

## **IterGetCurrent**

The `IterGetCurrent` property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for `IterTest` only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for `IterGetCurrent` returns a pointer to the current item in the collection.

*Default =*

## **IterIncrement**

The `IterIncrement` property specifies the code that increments the iterator. For example, the following command moves the `$iterator` ahead one item:

`$iterator++`

*Default =*

## **IterIncrementForCleanUp**

The `IterIncrementForCleanUp` property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

*Default =*

## **IterIncrementForInit**

The `IterIncrementForInit` property specifies the code that increments the iterator, for certain initialization cases.

*Default =*

## **IterInit**

The `IterInit` property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default =*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$RelationTargetType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end()

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default =*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<>, as defined in the STL.

vector<\$target\*>::const\_iterator You can change the iterator type to one of your own choice.

*Default =*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos

\$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$name. The default value for all other subjects is \$name.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$target*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($name>begin(), $name>end(),$item);$name>erase(pos)` This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

`pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find($name->begin(), $name->end(),p); $name->erase(pos)`

*Default =*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

`$name->clear()`

*Default =*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: \$cname-erase(\$keyName)

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

\$cname = \$item

*Default = Empty string*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## StaticArray

Defines properties for implementing static arrays.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter:

\$cname->push\_back(\$item)

For maps (qualified relations), the following command inserts an item into the map based on a key:

\$cname->insert(map<\$keyType,\$target\*>::value\_type( \$keyName,\$item))

*Default = Empty string*



## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = Empty string*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector<$target*> $cname()
```

*Default = Empty string*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default =*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = <\$RelationTargetType>*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

`$cname->find($item)`

*Default = Empty string*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname[\$multiplicity]*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: `$me$cname`

The variable `$me` is replaced with the object context variable as specified by the `Me` property. The variable `$cname` is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The `ContainerTypes::RelationType::GetAt` property specifies a template for the body of the operation. For example, the following command generates code that calls the `container at()` operation to retrieve the item at the indexed position:

`$cname->at($index)`

*Default = Empty string*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the `end()` operation for the container to locate the last item in the collection: `$cname-end()`

This property and `GetEndGenerate` were created to adhere to the standard library convention for “finding” where iteration should end. `GetEnd` is generated where `Get` is generated. The method name is defined by using the `CG::Relation::GetEnd` and `CG::Relation::GetEndGenerate` properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:  
\$cname-operator[](\$keyName)

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = Empty string*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

\$cname()

*Default = Empty string*

## **InitInCtorBody**

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL
```

*Default = Empty string*

## **InitStatic**

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## **IterCreate**

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = Empty string*

## **IterCreateByValue**

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = Empty string*

## **IterGetCurrent**

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = Empty string*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default = Empty string*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = Empty string*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = Empty string*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = Empty string*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default =*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end()

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = Empty string*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<>, as defined in the STL.

vector<\$target\*>::const\_iterator You can change the iterator type to one of your own choice.

*Default = \$(target)Seq*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$cname. The default value for all other subjects is \$cname.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$target*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname>erase(pos)` This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find(
$cname->begin(), $cname->end(),p); $cname->erase(pos)
```

*Default = Empty string*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

```
$cname->clear()
```

*Default = Empty string*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## UnboundedOrdered

Defines properties for implementing relations whose multiplicity is unbounded (\*) and that should be accessed sequentially.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = Empty string*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = Empty string*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.



For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector<$target*> $cname()
```

*Default = Empty string*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default =*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = sequence<\$RelationTargetType>*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default = Empty string*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a

particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$name

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$name is replaced with the name of the container, which is the role name for the relation.

*Default = \$name*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

\$cname->at(\$index)

*Default = Empty string*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps: \$cname-operator[](\$keyName)

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## **IncludeFiles**

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = Empty string*

## **Init**

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

```
$cname()
```

*Default = Empty string*

## **InitInCtorBody**

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL
```

*Default = Empty string*

## **InitStatic**

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<${target*}>::const_iterator $iterator; $iterator=${cname}>begin()
```

*Default = Empty string*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = Empty string*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = Empty string*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default = Empty string*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = Empty string*

## **IterIncrementForInit**

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = Empty string*

## **IterInit**

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## **IterReset**

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = Empty string*

## **IterReturnType**

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default =*

## **IterTest**

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()`

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = Empty string*

## **IterType**

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<>`, as defined in the STL.

`vector<$target*>::const_iterator` You can change the iterator type to one of your own choice.

*Default = \$(target)Seq*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the `OMContainers.StaticArray` metaclass) can use any other property in the same metaclass. Therefore, other properties that use `$Loop` expand the specified Loop property body.

*Default =*

`for (int pos = 0; pos $multiplicity; ++pos)` The default value for C is as follows: `int pos; for (pos = 0; pos $multiplicity; ++pos)` The default value for Java is as follows: `for (int pos = 0; pos $multiplicity; pos++)`

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for `RiCContainers::EmbeddedScalar` is `$me$name`. The default value for all other subjects is `$name`.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$target*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the `find()` operation to point the iterator to the item to be removed and then call `erase()` to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(), $item); $cname>erase(pos)` This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair `p`, whose second element, `p.second`, is the item to be removed. The `find()` operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

`pair<$keyType, $target*> p; p.second=$item; map<$keyType, $target*>::iterator pos=find(`

```
$cname->begin(), $cname->end(),p); $cname->erase(pos)
```

*Default = Empty string*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

```
$cname->clear()
```

*Default = Empty string*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed:

```
$cname-erase($keyName)
```

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## UnboundedUnordered

Defines properties for implementing relations whose multiplicity is unbounded (\*) and that should be accessed randomly.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = Empty string*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = Empty string*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`:

```
vector<$target*> $cname()
```

*Default = Empty string*

## CreateStatic

The CreateStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to Reference.

*Default =*

## CType



The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = sequence<\$RelationTargetType>*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default = Empty string*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

\$cname->at(\$index)

*Default = Empty string*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection:  
\$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:  
\$cname-operator[](\$keyName)

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = Empty string*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

```
$cname()
```

*Default = Empty string*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL
```

*Default = Empty string*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = Empty string*

## IterCreateByValue

The `IterCreateByValue` property is the same as `IterCreate`, but instantiates the iterator by value.

*Default = Empty string*

## IterGetCurrent

The `IterGetCurrent` property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for `IterTest` only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for `IterGetCurrent` returns a pointer to the current item in the collection.

*Default = Empty string*

## IterIncrement

The `IterIncrement` property specifies the code that increments the iterator. For example, the following command moves the `$iterator` ahead one item:

```
$iterator++
```

*Default = Empty string*

## IterIncrementForCleanUp

The `IterIncrementForCleanUp` property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

*Default = Empty string*

## IterIncrementForInit

The `IterIncrementForInit` property specifies the code that increments the iterator, for certain initialization cases.

*Default = Empty string*

## IterInit

The `IterInit` property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = Empty string*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default =*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end()

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = Empty string*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<>, as defined in the STL.

vector<\$target\*>::const\_iterator You can change the iterator type to one of your own choice.

*Default = \$(target)Seq*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos

\$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$name. The default value for all other subjects is \$name.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$target*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname>erase(pos)` This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find(
$name->begin(), $name->end(),p); $name->erase(pos)
```

*Default = Empty string*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

```
$cname->clear()
```

*Default = Empty string*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: \$cname-erase(\$keyName)

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

\$cname = \$item

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## User

Defines properties for user-defined implementations of relations.

You can create your own implementations for relations by defining a new set of properties under the User metaclass. Once these are defined, you can give them permanent status by manually saving them in the factory.prp file under any other name, for example MyFaves. To complete their installation, you must add the new name as an enumerated value to the CG::Relation::Implementation property.

For example, you can change the definition of the Implementation property as follows: Subject CG Metaclass Relation Property Implementation Enum "Default,Scalar,Fixed,BoundedOrdered, BoundedUnordered,UnboundedOrdered, UnboundedUnordered,Qualified,MyFaves, User" "Default" end end

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = Empty string*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = Empty string*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector<$target*> $cname()
```

*Default = Empty string*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default = Empty string*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.



*Default = Empty string*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default = Empty string*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = Empty string*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = Empty string*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

```
$cname->at($index)
```

*Default = Empty string*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps: \$cname-operator[](\$keyName)

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = Empty string*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

```
$cname()
```

*Default = Empty string*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL
```

*Default = Empty string*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = Empty string*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = Empty string*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = Empty string*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default = Empty string*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = Empty string*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = Empty string*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = Empty string*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = Empty string*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end()

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = Empty string*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<>, as defined in the STL.

vector<\$target\*>::const\_iterator You can change the iterator type to one of your own choice.

*Default = Empty string*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$cname. The default value for all other

subjects is \$cname.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = Empty string*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname>erase(pos)` This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find(
$cname->begin(), $cname->end(),p); $cname->erase(pos)
```

*Default = Empty string*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

```
$cname->clear()
```

*Default = Empty string*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

\$cname = \$item

*Default* =

## **Type**

The Type property specifies the type of the container as a pointer to the relation.

*Default* =

# OMCpp2CorbaContainers

---

Rational Rhapsody uses containers to implement to-many relations between objects. These include relationships of one object to many, or many objects to many.

The OMCpp2CorbaContainers subject contains metaclasses that contain properties that control the implementing of relations.

Each property in this subject includes the default value for each container type and relation type. For easier readability, the placeholder RelationType in these values represents all the other relation types that are not explicitly detailed. For example, the relation type User might have the default value of an empty string, whereas all the other relation types have the value of an empty MultiLine.

## BoundedOrdered

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed sequentially.

### Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = \$cname->add(\$item)*

### Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = \$cname = new \$CType*



## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector<$target*> $cname()
```

*Default = \$CType \$cname*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default = new \$CType*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = OMList<\$RelationTargetType>*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default = \$cname->find(\$item)*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

`$cname->at($index)`

*Default = \$cname->getAt(\$index)*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:

`$cname-operator[]($keyName)`

*Default =*

## **IncludeDirective**

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## **IncludeFiles**

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = <oxf/omlist.h>*

## **Init**

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

`$cname()`

*Default = \$cname()*

## **InitInCtorBody**

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

`pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL`

*Default = Empty string*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = \$IterType \$iterator(\$cname)*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = \*\$iterator*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default = \$iterator++*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = \$IterReset*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterReset*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = \$iterator.reset()*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end()

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With

OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = \$IterGetCurrent*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<>, as defined in the STL.

vector<\$target\*>::const\_iterator You can change the iterator type to one of your own choice.

*Default = OMIterator<\$RelationTargetType>*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$name. The default value for all other subjects is \$name.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$(PoaPrefix)\$(MappedTarget)*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to be removed and then call erase() to remove it. vector<\$target\*>::iterator pos=find(\$name>begin(), \$name>end(),\$item);\$name>erase(pos) This operation applies only to "to-many" (non-scalar)

containers.

For maps (qualified relations), the following commands create a pair `p`, whose second element, `p.second`, is the item to be removed. The `find()` operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find(
$name->begin(), $name->end(),p); $name->erase(pos)
```

*Default = \$name->remove(\$item)*

## RemoveAll

The `RemoveAll` property specifies the command used to remove all items from the container. For example, the following command calls the `clear()` operation for the container to remove all items:

```
$name->clear()
```

*Default = \$name->removeAll()*

## RemoveKey

The `RemoveKey` property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the `erase()` operation for the container, passing it the `$keyName`, which maps into a dictionary used to locate the item to be removed:

```
$name-erase($keyName)
```

*Default =*

## Set

The `Set` property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, `$name`, is a role name of the relation itself, because there is only one class involved:

```
$name = $item
```

*Default =*

## Type

The `Type` property specifies the type of the container as a pointer to the relation.

*Default =*

# BoundedUnordered

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed randomly.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = `$cname->add($item)`*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = `$cname = new $CType($multiplicity)`*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`:

```
vector<$target*> $cname()
```

*Default = `$CType $cname($multiplicity)`*

## CreateStatic



The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default = new \$CType(\$multiplicity)*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = OMCollection<\$RelationTargetType>*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default = \$cname->find(\$item)*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

```
$cname->at($index)
```

```
Default = $cname->getAt($index)
```

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

```
Default =
```

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps: \$cname-operator[](\$keyName)

```
Default =
```

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

```
Default = strong
```

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = <oxf/omcollec.h>*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

`$cname()`

*Default = \$cname(\$multiplicity)*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

`pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL`

*Default = Empty string*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the

container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = \$IterType \$iterator(\$cname)*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = \*\$iterator*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default = \$iterator++*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = \$IterReset*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterReset*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = \$iterator.reset()*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()`

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = \$IterGetCurrent*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<>`, as defined in the STL.

`vector<$target*>::const_iterator` You can change the iterator type to one of your own choice.

*Default = OMIterator<\$RelationTargetType>*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$name. The default value for all other subjects is \$name.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$(PoaPrefix)\$(MappedTarget)*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. vector<\$target\*>::iterator pos=find(\$cname>begin(), \$cname>end(), \$item); \$cname->erase(pos) This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find(
$cname->begin(), $cname->end(),p); $cname->erase(pos)
```

*Default = \$cname->remove(\$item)*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

`$cname->clear()`

*Default = `$cname->removeAll()`*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed:  
`$cname-erase($keyName)`

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

`$cname = $item`

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

# EmbeddedFixed

Defines properties for implementing embedded fixed relations.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter:

`$cname->push_back($item)`

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = \$cname->add(\$item)*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = \$cname = new \$CType*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector<$target*> $cname()
```

*Default = \$CType \$cname*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default = new \$CType*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = OMList<\$RelationTargetType>*



## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default = \$cname->find(\$item)*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

```
$cname->at($index)
```

*Default = \$cname->getAt(\$index)*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection:

\$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:  
\$cname-operator[](\$keyName)

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = <oxf/omlist.h>*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

```
$cname()
```

*Default = \$cname()*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL
```

*Default = Empty string*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = \$IterType \$iterator(\$cname)*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The `IterGetCurrent` property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for `IterTest` only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for `IterGetCurrent` returns a pointer to the current item in the collection.

*Default = `*$iterator`*

## **IterIncrement**

The `IterIncrement` property specifies the code that increments the iterator. For example, the following command moves the `$iterator` ahead one item:

`$iterator++`

*Default = `$iterator++`*

## **IterIncrementForCleanUp**

The `IterIncrementForCleanUp` property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

*Default = `$IterReset`*

## **IterIncrementForInit**

The `IterIncrementForInit` property specifies the code that increments the iterator, for certain initialization cases.

*Default = `$IterReset`*

## **IterInit**

The `IterInit` property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the `relation.getRelation>()` generated operation.

*Default =*

## **IterReset**

The `IterReset` property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection:

`$iterator=$cname->begin()`

*Default = \$iterator.reset()*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end()

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = \$IterGetCurrent*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<>, as defined in the STL.

vector<\$target\*>::const\_iterator You can change the iterator type to one of your own choice.

*Default = OMIterator<\$RelationTargetType>*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$cname. The default value for all other

subjects is \$cname.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$(PoaPrefix)\$(MappedTarget)*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(), $item); $cname->erase(pos)` This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

`pair<$keyType, $target*> p; p.second=$item; map<$keyType, $target*>::iterator pos=find($cname->begin(), $cname->end(), p); $cname->erase(pos)`

*Default = \$cname->remove(\$item)*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

`$cname->clear()`

*Default = \$cname->removeAll()*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: `$cname->erase($keyName)`

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## EmbeddedScalar

Defines properties for implementing embedded scalar (one-to-one) relations.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by \$cname, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = \$cname->add(\$item)*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = \$cname = new \$CType*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector<$target*> $cname()
```

*Default = \$CType \$cname*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default = new \$CType*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = OMList<\$RelationTargetType>*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default = \$cname->find(\$item)*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.



*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

`$cname->at($index)`

*Default = \$cname->getAt(\$index)*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:

`$cname-operator[]($keyName)`

*Default =*

## **IncludeDirective**

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## **IncludeFiles**

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = <oxf/omlist.h>*

## **Init**

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

`$cname()`

*Default = \$cname()*

## **InitInCtorBody**

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

`pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL`

*Default = Empty string*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = \$IterType \$iterator(\$cname)*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = \*\$iterator*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default = \$iterator++*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = \$IterReset*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterReset*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = \$iterator.reset()*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end()

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With

OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = \$IterGetCurrent*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<>, as defined in the STL.

vector<\$target\*>::const\_iterator You can change the iterator type to one of your own choice.

*Default = OMIterator<\$RelationTargetType>*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$name. The default value for all other subjects is \$name.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$(PoaPrefix)\$(MappedTarget)*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to be removed and then call erase() to remove it. vector<\$target\*>::iterator pos=find(\$cname>begin(), \$cname>end(),\$item);\$cname>erase(pos) This operation applies only to "to-many" (non-scalar)

containers.

For maps (qualified relations), the following commands create a pair `p`, whose second element, `p.second`, is the item to be removed. The `find()` operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find(
$name->begin(), $name->end(),p); $name->erase(pos)
```

*Default = \$name->remove(\$item)*

## RemoveAll

The `RemoveAll` property specifies the command used to remove all items from the container. For example, the following command calls the `clear()` operation for the container to remove all items:

```
$name->clear()
```

*Default = \$name->removeAll()*

## RemoveKey

The `RemoveKey` property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the `erase()` operation for the container, passing it the `$keyName`, which maps into a dictionary used to locate the item to be removed:

```
$name-erase($keyName)
```

*Default =*

## Set

The `Set` property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, `$name`, is a role name of the relation itself, because there is only one class involved:

```
$name = $item
```

*Default =*

## Type

The `Type` property specifies the type of the container as a pointer to the relation.

*Default =*

## Fixed

Defines properties for implementing relations of fixed size.

### Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = `$cname->add($item)`*

### Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = `$cname = new $CType($multiplicity)`*

### CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`:

```
vector<$target*> $cname()
```

*Default = `$CType $cname($multiplicity)`*

### CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default = new \$CType(\$multiplicity)*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = OMCollection<\$RelationTargetType>*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default = \$cname->find(\$item)*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*



## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

```
$cname->at($index)
```

```
Default = $cname->getAt($index)
```

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

```
Default =
```

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps: \$cname-operator[](\$keyName)

```
Default =
```

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

```
Default = strong
```

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = <oxf/omcollec.h>*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

`$cname()`

*Default = \$cname(\$multiplicity)*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

`pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL`

*Default = Empty string*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the

container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = \$IterType \$iterator(\$cname)*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = \*\$iterator*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default = \$iterator++*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = \$IterReset*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterReset*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = \$iterator.reset()*

## IterReturntype

The IterReturntype property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()`

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = \$IterGetCurrent*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<>`, as defined in the STL.

`vector<$target*>::const_iterator` You can change the iterator type to one of your own choice.

*Default = OMIterator<\$RelationTargetType>*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$name. The default value for all other subjects is \$name.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$(PoaPrefix)\$(MappedTarget)*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. vector<\$target\*>::iterator pos=find(\$cname>begin(), \$cname>end(),\$item);\$cname>erase(pos) This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

pair<\$keyType,\$target\*> p; p.second=\$item; map<\$keyType,\$target\*>::iterator pos=find(\$cname->begin(), \$cname->end(),p); \$cname->erase(pos)

*Default = \$cname->remove(\$item)*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

`$cname->clear()`

*Default = \$cname->removeAll()*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed:  
`$cname-erase($keyName)`

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

`$cname = $item`

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## General

Defines properties that enable you to set the directives and include files for the container.

## ContainerDirectives

The ContainerDirectives property specifies the preprocessor directives that are necessary when compiling code that uses a particular container library.

No additional directives are required when you use OMContainers.

*Default = Empty MultiLine*

## ContainerIncludes

The ContainerIncludes property specifies header files that must be included when you use a particular container library.

For example, when you use STLContainers, the following string causes one #include directive per container to be added to generated files (such as #include string): string, algorithm, vector, list, map, iterator Whether the #include directives are added to source or header files depends on the value of the IncludeDirective property.

*Default = Empty string*

## Qualified

Defines properties for implementing qualified relations, which are accessed by way of a key.

### Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = \$cname->add(\$keyName,\$item)*

### Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = \$cname = new \$CType*

### CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector<$target*> $cname()
```

*Default = \$CType \$cname*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default = new \$CType*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = OMMMap<\$keyType, \$RelationTargetType>*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default = \$cname->find(\$item)*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get



The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$name

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$name is replaced with the name of the container, which is the role name for the relation.

*Default = \$name*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

\$name->at(\$index)

*Default = \$name->getAt(\$index)*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$name-end()

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps: \$name-operator[](\$keyName)

*Default = \$name->getKey(\$keyName)*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## **IncludeFiles**

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = <oxf/ommap.h>*

## **Init**

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

`$cname()`

*Default = \$cname()*

## **InitInCtorBody**

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

`pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL`

*Default = Empty string*

## **InitStatic**

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<${target*}>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = \$IterType \$iterator(\$cname)*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: *\*\$iterator*

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = \*\$iterator*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the *\$iterator* ahead one item:

```
$iterator++
```

*Default = \$iterator++*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = \$IterReset*

## **IterIncrementForInit**

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterReset*

## **IterInit**

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## **IterReset**

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = \$iterator.reset()*

## **IterReturnType**

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## **IterTest**

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()`

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = \$IterGetCurrent*

## **IterType**

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<>, as defined in the STL.

vector<\$target\*>::const\_iterator You can change the iterator type to one of your own choice.

*Default = OMIterator<\$RelationTargetType>*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$name. The default value for all other subjects is \$name.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$(PoaPrefix)\$(MappedTarget)*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to be removed and then call erase() to remove it. vector<\$target\*>::iterator pos=find(\$name>begin(), \$name>end(), \$item); \$name>erase(pos) This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

pair<\$keyType, \$target\*> p; p.second=\$item; map<\$keyType, \$target\*>::iterator pos=find(

```
$cname->begin(), $cname->end(),p); $cname->erase(pos)
```

*Default = \$cname->remove(\$item)*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

```
$cname->clear()
```

*Default = \$cname->removeAll()*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed:

```
$cname-erase($keyName)
```

*Default = \$cname->remove(\$keyName)*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## Scalar

Defines properties for implementing scalar relations.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default* =

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector:

```
new vector<$target*>
```

*Default* =

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`:

```
vector<$target*> $cname()
```

*Default* =

## CreateStatic

The CreateStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to Reference.

*Default* =

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = \$RelationTargetType*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default =*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:



`$cname->at($index)`

*Default =*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection:  
`$cname-end()`

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:  
`$cname-operator[]($keyName)`

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default =*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

```
$cname()
```

*Default =*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL
```

*Default =*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default =*

## IterCreateByValue

The `IterCreateByValue` property is the same as `IterCreate`, but instantiates the iterator by value.

*Default =*

## **IterGetCurrent**

The `IterGetCurrent` property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for `IterTest` only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for `IterGetCurrent` returns a pointer to the current item in the collection.

*Default =*

## **IterIncrement**

The `IterIncrement` property specifies the code that increments the iterator. For example, the following command moves the `$iterator` ahead one item:

`$iterator++`

*Default =*

## **IterIncrementForCleanUp**

The `IterIncrementForCleanUp` property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

*Default =*

## **IterIncrementForInit**

The `IterIncrementForInit` property specifies the code that increments the iterator, for certain initialization cases.

*Default =*

## **IterInit**

The `IterInit` property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default =*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$RelationTargetType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end()

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default =*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<>, as defined in the STL.

vector<\$target\*>::const\_iterator You can change the iterator type to one of your own choice.

*Default =*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos

\$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$name. The default value for all other subjects is \$name.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$(PoaPrefix)\$(MappedTarget)*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname>erase(pos)` This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

`pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find($cname->begin(), $cname->end(),p); $cname->erase(pos)`

*Default =*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

`$cname->clear()`

*Default =*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: \$cname-erase(\$keyName)

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

\$cname = \$item

*Default = \$cname = \$item*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## StaticArray

Defines properties for implementing static arrays.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter:

\$cname->push\_back(\$item)

For maps (qualified relations), the following command inserts an item into the map based on a key:

\$cname->insert(map<\$keyType,\$target\*>::value\_type( \$keyName,\$item))

*Default =*

\$Loop { if (!\$cname[pos]) { \$cname[pos] = \$item; break; } }

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = Empty string*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector<$target*> $cname()
```

*Default = \$CType*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default = new \$CType*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = \$RelationTargetType \$cname[\$multiplicity]*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

`$cname->find($item)`

*Default = `$cname->find($item)`*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = `$CType`*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: `$me$cname`

The variable `$me` is replaced with the object context variable as specified by the `Me` property. The variable `$cname` is replaced with the name of the container, which is the role name for the relation.

*Default = `$cname`*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the `container at()` operation to retrieve the item at the indexed position:

`$cname->at($index)`

*Default = `$cname[$index]`*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the `end()` operation for the container to locate the last item in the collection: `$cname-end()`

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the `CG::Relation::GetEnd` and `CG::Relation::GetEndGenerate` properties.



*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:  
\$cname-operator[](\$keyName)

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = <oxf/omlist.h>*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

\$cname()

*Default = Empty string*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL
```

*Default =*

```
$Loop { $cname[pos] = NULL; }
```

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = \$IterType \$iterator = 0;*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = \$cname[\$iterator]*

## **IterIncrement**

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

`$iterator++`

*Default = \$iterator++*

## **IterIncrementForCleanUp**

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = \$IterIncrement*

## **IterIncrementForInit**

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterIncrement*

## **IterInit**

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

*Default =*

## **IterReset**

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection:

`$iterator=$cname->begin()`

*Default = \$iterator = 0*

## **IterReturnType**

The IterReturnType property specifies the return type for functions such as getters for the various

containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end()

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = (\$iterator < \$multiplicity) && \$cname[\$iterator]*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<>, as defined in the STL.

vector<\$target\*>::const\_iterator You can change the iterator type to one of your own choice.

*Default = int*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

Default for C++ = for (int pos = 0; pos < \$multiplicity; ++pos)

Default for C = int pos; for (pos = 0; pos < \$multiplicity; ++pos)

Default for Java = for (int pos = 0; pos < \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$name. The default value for all other subjects is \$cname.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$(PoaPrefix)\$(MappedTarget)*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(), $item); $cname->erase(pos)` This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find(
$name->begin(), $name->end(),p); $name->erase(pos)
```

*Default =*

```
$Loop { if ($cname[pos] == $item) { $cname[pos] = NULL; } }
```

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

```
$cname->clear()
```

*Default = Empty MultiLine*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

*Default =*

## SetAt

The SetAt property specifies how code is generated for the body of the mutator for a scalar container.

*Default = \$cname[\$index] = \$item*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

# UnboundedOrdered

Defines properties for implementing relations whose multiplicity is unbounded (\*) and that should be accessed sequentially.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = \$cname->add(\$item)*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = \$cname = new \$CType*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector<$target*> $cname()
```

*Default = \$CType \$cname*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default = new \$CType*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = OMList<\$RelationTargetType>*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default = \$cname->find(\$item)*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

`$cname->at($index)`

*Default = \$cname->getAt(\$index)*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: `$cname-end()`

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey



The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:  
\$cname-operator[](\$keyName)

*Default =*

## **IncludeDirective**

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## **IncludeFiles**

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = <oxf/omlist.h>*

## **Init**

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

\$cname()

*Default = \$cname()*

## **InitInCtorBody**

The `InitInCtorBody` property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL
```

*Default = Empty string*

## InitStatic

The `InitStatic` property defines the initialization of a relation when `CG::Relation::Containment` is set to `Value`.

*Default =*

## IterCreate

The `IterCreate` property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = \$IterType \$iterator(\$cname)*

## IterCreateByValue

The `IterCreateByValue` property is the same as `IterCreate`, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The `IterGetCurrent` property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for `IterTest` only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for `IterGetCurrent` returns a pointer to the current item in the collection.

*Default = \*\$iterator*

## IterIncrement

The `IterIncrement` property specifies the code that increments the iterator. For example, the following

command moves the \$iterator ahead one item:

```
$iterator++
```

*Default = \$iterator++*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = \$IterReset*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterReset*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = \$iterator.reset()*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end()

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = \$IterGetCurrent*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<>, as defined in the STL.

vector<\$target\*>::const\_iterator You can change the iterator type to one of your own choice.

*Default = OMIterator<\$RelationTargetType>*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$cname. The default value for all other subjects is \$cname.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$(PoaPrefix)\$(MappedTarget)*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(), $item); $cname->erase(pos)` This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType, $target*> p; p.second=$item; map<$keyType, $target*>::iterator pos=find(
$cname->begin(), $cname->end(), p); $cname->erase(pos)
```

*Default = \$cname->remove(\$item)*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

```
$cname->clear()
```

*Default = \$cname->removeAll()*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: `$cname->erase($keyName)`

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## UnboundedUnordered

Defines properties for implementing relations whose multiplicity is unbounded (\*) and that should be accessed randomly.

### Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = \$cname->add(\$item)*

### Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = \$cname = new \$CType(\$multiplicity)*

### CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`:

```
vector<$target*> $cname()
```

*Default = \$CType \$cname(\$multiplicity)*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default = new \$CType(\$multiplicity)*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = OMCollection<\$RelationTargetType>*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default = \$cname->find(\$item)*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The

variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

`$cname->at($index)`

*Default = \$cname->getAt(\$index)*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: `$cname-end()`

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:  
`$cname-operator[]($keyName)`

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:



- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## **IncludeFiles**

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = <oxf/omcollec.h>*

## **Init**

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

```
$cname()
```

*Default = \$cname(\$multiplicity)*

## **InitInCtorBody**

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL
```

*Default = Empty string*

## **InitStatic**

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = \$IterType \$iterator(\$cname)*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = \*\$iterator*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default = \$iterator++*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = \$IterReset*

## IterIncrementForInit

The `IterIncrementForInit` property specifies the code that increments the iterator, for certain initialization cases.

*Default = `$IterReset`*

## IterInit

The `IterInit` property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## IterReset

The `IterReset` property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection:

`$iterator=$cname->begin()`

*Default = `$iterator.reset()`*

## IterReturnType

The `IterReturnType` property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = `$IterType`*

## IterTest

The `IterTest` property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()`

With `STLContainers`, unlike `OMContainers`, it is possible to store a `NULL` value as the container. With `OMContainers`, `IterTest` retrieves the current item in the collection (the same as `IterGetCurrent`).

*Default = `$IterGetCurrent`*

## IterType

The `IterType` property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<>`, as defined in the STL.

`vector<$target*>::const_iterator` You can change the iterator type to one of your own choice.

*Default = OIterator<\$RelationTargetType>*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$name. The default value for all other subjects is \$name.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$(PoaPrefix)\$(MappedTarget)*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to be removed and then call erase() to remove it. vector<\$target\*>::iterator pos=find(\$name>begin(), \$name>end(), \$item); \$name>erase(pos) This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

pair<\$keyType, \$target\*> p; p.second=\$item; map<\$keyType, \$target\*>::iterator pos=find(\$name->begin(), \$name->end(), p); \$name->erase(pos)

*Default = \$name->remove(\$item)*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

```
$cname->clear()
```

*Default = \$cname->removeAll()*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed:

```
$cname-erase($keyName)
```

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## User

Defines properties for user-defined implementations of relations.

You can create your own implementations for relations by defining a new set of properties under the User metaclass. Once these are defined, you can give them permanent status by manually saving them in the factory.prp file under any other name, for example MyFaves. To complete their installation, you must add the new name as an enumerated value to the CG::Relation::Implementation property.

For example, you can change the definition of the Implementation property as follows: Subject CG Metaclass Relation Property Implementation Enum "Default,Scalar,Fixed,BoundedOrdered,BoundedUnordered,UnboundedOrdered, UnboundedUnordered,Qualified,MyFaves, User" "Default" end end

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = Empty string*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = Empty string*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`:

```
vector<$target*> $cname()
```

*Default = Empty string*

## CreateStatic

The CreateStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to Reference.

*Default = Empty string*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = Empty string*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default = Empty string*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = Empty string*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = Empty string*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the

targets of a to-many relation that uses an index.

The `ContainerTypes::RelationType::GetAt` property specifies a template for the body of the operation. For example, the following command generates code that calls the container `at()` operation to retrieve the item at the indexed position:

```
$cname->at($index)
```

*Default = Empty string*

## GetEnd

The `GetEnd` property specifies the command used to retrieve the last item in the container. For example, the following command calls the `end()` operation for the container to locate the last item in the collection:

```
$cname-end()
```

This property and `GetEndGenerate` were created to adhere to the standard library convention for “finding” where iteration should end. `GetEnd` is generated where `Get` is generated. The method name is defined by using the `CG::Relation::GetEnd` and `CG::Relation::GetEndGenerate` properties.

*Default =*

## GetKey

The `GetKey` property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator `[]`, which has been overloaded according to the STL definition for maps:

```
$cname-operator[]($keyName)
```

*Default =*

## IncludeDirective

The `IncludeDirective` property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- `strong` - The `#include` directives are added to the header file.
- `weak` - The `#include` directives are added to the source file with forward declarations in the header file.

*Default = strong*

## IncludeFiles



The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = Empty string*

## **Init**

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

```
$cname()
```

*Default = Empty string*

## **InitInCtorBody**

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL
```

*Default = Empty string*

## **InitStatic**

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## **IterCreate**

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = Empty string*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = Empty string*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = Empty string*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the `$iterator` ahead one item:

```
$iterator++
```

*Default = Empty string*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

*Default = Empty string*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = Empty string*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = Empty string*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = Empty string*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()`

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = Empty string*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<>`, as defined in the STL.

`vector<$target*>::const_iterator` You can change the iterator type to one of your own choice.

*Default = Empty string*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in

the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$name. The default value for all other subjects is \$name.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = Empty string*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname>erase(pos)` This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

`pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find($cname->begin(), $cname->end(),p); $cname->erase(pos)`

*Default = Empty string*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

`$cname->clear()`

*Default = Empty string*

## **RemoveKey**

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: \$cname-erase(\$keyName)

*Default =*

## **Set**

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

\$cname = \$item

*Default =*

## **Type**

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

# OMCppOfCorbaContainers

---

Rational Rhapsody uses containers to implement to-many relations between objects. These include relationships of one object to many, or many objects to many.

The OMCppOfCorbaContainers subject contains metaclasses that contain properties that control the implementing of relations.

Each property in this subject includes the default value for each container type and relation type. For easier readability, the placeholder RelationType in these values represents all the other relation types that are not explicitly detailed. For example, the relation type User might have the default value of an empty string, whereas all the other relation types have the value of an empty MultiLine.

## BoundedOrdered

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed sequentially.

### Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$name, and passes the item to be added as a formal parameter:

```
$name->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$name->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = \$name->add(\$item)*

### Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = \$name = new \$CType*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector<$target*> $cname()
```

*Default = \$CType \$cname*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default = new \$CType*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = OMList<\$RelationTargetType>*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default = \$cname->find(\$item)*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

`$cname->at($index)`

*Default = \$cname->getAt(\$index)*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:

`$cname-operator[]($keyName)`



*Default =*

## **IncludeDirective**

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## **IncludeFiles**

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = <oxf/omlist.h>*

## **Init**

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

`$cname()`

*Default = \$cname()*

## **InitInCtorBody**

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

`pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL`

*Default = Empty string*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = OMIterator<\$RelationTargetType> \$iterator(\$cname)*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = Empty string*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default = Empty string*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = Empty string*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = Empty string*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = Empty string*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end()

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With

OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = Empty string*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<>, as defined in the STL.

vector<\$target\*>::const\_iterator You can change the iterator type to one of your own choice.

*Default = \$(constant)\$(FixedTarget)Seq\**

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$name. The default value for all other subjects is \$name.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$(PoaPrefix)\$(MappedTarget)*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to be removed and then call erase() to remove it. vector<\$target\*>::iterator pos=find(\$name>begin(), \$name>end(),\$item);\$name>erase(pos) This operation applies only to "to-many" (non-scalar)

containers.

For maps (qualified relations), the following commands create a pair `p`, whose second element, `p.second`, is the item to be removed. The `find()` operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find(
$name->begin(), $name->end(),p); $name->erase(pos)
```

*Default = \$name->remove(\$item)*

## RemoveAll

The `RemoveAll` property specifies the command used to remove all items from the container. For example, the following command calls the `clear()` operation for the container to remove all items:

```
$name->clear()
```

*Default = \$name->removeAll()*

## RemoveKey

The `RemoveKey` property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the `erase()` operation for the container, passing it the `$keyName`, which maps into a dictionary used to locate the item to be removed:

```
$name-erase($keyName)
```

*Default =*

## Set

The `Set` property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, `$name`, is a role name of the relation itself, because there is only one class involved:

```
$name = $item
```

*Default =*

## Type

The `Type` property specifies the type of the container as a pointer to the relation.

*Default =*

# BoundedUnordered

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed randomly.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = `$cname->add($item)`*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = `$cname = new $CType($multiplicity)`*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`:

```
vector<$target*> $cname()
```

*Default = `$CType $cname($multiplicity)`*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default = new \$CType(\$multiplicity)*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = OMCollection<\$RelationTargetType>*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default = \$cname->find(\$item)*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

```
$cname->at($index)
```

```
Default = $cname->getAt($index)
```

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

```
Default =
```

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps: \$cname-operator[](\$keyName)

```
Default =
```

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

```
Default = strong
```



## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = <oxf/omcollec.h>*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

`$cname()`

*Default = \$cname(\$multiplicity)*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

`pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL`

*Default = Empty string*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the

container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = OMIterator<\$RelationTargetType> \$iterator(\$cname)*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = Empty string*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default = Empty string*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = Empty string*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = Empty string*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = Empty string*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()`

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = Empty string*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<>`, as defined in the STL.

`vector<$target*>::const_iterator` You can change the iterator type to one of your own choice.

*Default = \$(constant)\$(FixedTarget)Seq\**

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$name. The default value for all other subjects is \$name.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$(PoaPrefix)\$(MappedTarget)*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. vector<\$target\*>::iterator pos=find(\$cname>begin(), \$cname>end(),\$item);\$cname>erase(pos) This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find(
$cname->begin(), $cname->end(),p); $cname->erase(pos)
```

*Default = \$cname->remove(\$item)*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

`$cname->clear()`

*Default = `$cname->removeAll()`*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed:  
`$cname-erase($keyName)`

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

`$cname = $item`

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

# EmbeddedFixed

Defines properties for implementing embedded fixed relations.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter:

`$cname->push_back($item)`

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = \$cname->add(\$item)*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = \$cname = new \$CType*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector<$target*> $cname()
```

*Default = \$CType \$cname*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default = new \$CType*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = OMList<\$RelationTargetType>*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default = \$cname->find(\$item)*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

```
$cname->at($index)
```

*Default = \$cname->getAt(\$index)*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection:

\$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:  
\$cname-operator[](\$keyName)

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = <oxf/omlist.h>*



## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

```
$cname()
```

*Default = \$cname()*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL
```

*Default = Empty string*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = OMIterator<\$RelationTargetType> \$iterator(\$cname)*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: \*\$iterator

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = Empty string*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default = Empty string*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = Empty string*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = Empty string*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = Empty string*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end()

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = Empty string*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<>, as defined in the STL.

vector<\$target\*>::const\_iterator You can change the iterator type to one of your own choice.

*Default = \$(constant)\$(FixedTarget)Seq\**

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$cname. The default value for all other

subjects is \$cname.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$(PoaPrefix)\$(MappedTarget)*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(), $item); $cname->erase(pos)` This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

`pair<$keyType, $target*> p; p.second=$item; map<$keyType, $target*>::iterator pos=find($cname->begin(), $cname->end(), p); $cname->erase(pos)`

*Default = \$cname->remove(\$item)*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

`$cname->clear()`

*Default = \$cname->removeAll()*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: `$cname->erase($keyName)`

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## EmbeddedScalar

Defines properties for implementing embedded scalar (one-to-one) relations.

### Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = \$cname->add(\$item)*

### Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = \$cname = new \$CType*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector<$target*> $cname()
```

*Default = \$CType \$cname*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default = new \$CType*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = OMList<\$RelationTargetType>*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default = \$cname->find(\$item)*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

`$cname->at($index)`

*Default = \$cname->getAt(\$index)*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:

`$cname-operator[]($keyName)`

*Default =*

## **IncludeDirective**

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## **IncludeFiles**

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = <oxf/omlist.h>*

## **Init**

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

`$cname()`

*Default = \$cname()*

## **InitInCtorBody**

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

`pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL`



*Default = Empty string*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = OMIterator<\$RelationTargetType> \$iterator(\$cname)*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = Empty string*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default = Empty string*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = Empty string*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = Empty string*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = Empty string*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end()

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With

OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = Empty string*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<>, as defined in the STL.

vector<\$target\*>::const\_iterator You can change the iterator type to one of your own choice.

*Default = \$(constant)\$(FixedTarget)Seq\*;*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$name. The default value for all other subjects is \$name.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$(PoaPrefix)\$(MappedTarget)*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to be removed and then call erase() to remove it. vector<\$target\*>::iterator pos=find(\$name>begin(), \$name>end(),\$item);\$name>erase(pos) This operation applies only to "to-many" (non-scalar)

containers.

For maps (qualified relations), the following commands create a pair `p`, whose second element, `p.second`, is the item to be removed. The `find()` operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find(
$name->begin(), $name->end(),p); $name->erase(pos)
```

*Default = \$name->remove(\$item)*

## RemoveAll

The `RemoveAll` property specifies the command used to remove all items from the container. For example, the following command calls the `clear()` operation for the container to remove all items:

```
$name->clear()
```

*Default = \$name->removeAll()*

## RemoveKey

The `RemoveKey` property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the `erase()` operation for the container, passing it the `$keyName`, which maps into a dictionary used to locate the item to be removed:

```
$name-erase($keyName)
```

*Default =*

## Set

The `Set` property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, `$name`, is a role name of the relation itself, because there is only one class involved:

```
$name = $item
```

*Default =*

## Type

The `Type` property specifies the type of the container as a pointer to the relation.

*Default =*

## Fixed

Defines properties for implementing relations of fixed size.

### Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = `$cname->add($item)`*

### Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = `$cname = new $CType($multiplicity)`*

### CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`:

```
vector<$target*> $cname()
```

*Default = `$CType $cname($multiplicity)`*

### CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default = new \$CType(\$multiplicity)*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = OMCollection<\$RelationTargetType>*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default = \$cname->find(\$item)*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

```
$cname->at($index)
```

```
Default = $cname->getAt($index)
```

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

```
Default =
```

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps: \$cname-operator[](\$keyName)

```
Default =
```

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

```
Default = strong
```

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = <oxf/omcollec.h>*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

`$cname()`

*Default = \$cname(\$multiplicity)*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

`pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL`

*Default = Empty string*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the



container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = OMIterator<\$RelationTargetType> \$iterator(\$cname)*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = Empty string*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default = Empty string*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = Empty string*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = Empty string*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = Empty string*

## IterReturntype

The IterReturntype property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()`

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = Empty string*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<>`, as defined in the STL.

`vector<$target*>::const_iterator` You can change the iterator type to one of your own choice.

*Default = \$(constant)\$(FixedTarget)Seq\**

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$name. The default value for all other subjects is \$name.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$(PoaPrefix)\$(MappedTarget)*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. vector<\$target\*>::iterator pos=find(\$cname>begin(), \$cname>end(),\$item);\$cname>erase(pos) This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

pair<\$keyType,\$target\*> p; p.second=\$item; map<\$keyType,\$target\*>::iterator pos=find(\$cname->begin(), \$cname->end(),p); \$cname->erase(pos)

*Default = \$cname->remove(\$item)*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

`$cname->clear()`

*Default = \$cname->removeAll()*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed:  
`$cname-erase($keyName)`

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

`$cname = $item`

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## General

Defines properties that enable you to set the directives and include files for the container.

## ContainerDirectives

The ContainerDirectives property specifies the preprocessor directives that are necessary when compiling code that uses a particular container library.

No additional directives are required when you use OMContainers.

*Default = Empty MultiLine*

## ContainerIncludes

The ContainerIncludes property specifies header files that must be included when you use a particular container library.

For example, when you use STLContainers, the following string causes one #include directive per container to be added to generated files (such as #include string): string, algorithm, vector, list, map, iterator Whether the #include directives are added to source or header files depends on the value of the IncludeDirective property.

*Default = Empty string*

## Qualified

Defines properties for implementing qualified relations, which are accessed by way of a key.

### Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = \$cname->add(\$keyName,\$item)*

### Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = \$cname = new \$CType*

### CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector<$target*> $cname()
```

*Default = \$CType \$cname*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default = new \$CType*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = OMMMap<\$keyType, \$RelationTargetType>*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default = \$cname->find(\$item)*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$name

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$name is replaced with the name of the container, which is the role name for the relation.

*Default = \$name*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

\$name->at(\$index)

*Default = \$name->getAt(\$index)*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$name-end()

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps: \$name-operator[](\$keyName)

*Default = \$name->getKey(\$keyName)*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## **IncludeFiles**

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = <oxf/ommap.h>*

## **Init**

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

`$cname()`

*Default = \$cname()*

## **InitInCtorBody**

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

`pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL`

*Default = Empty string*

## **InitStatic**



The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<${target*}>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = OMIterator<\${RelationTargetType}> \$iterator(\$cname)*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = Empty string*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default = Empty string*

## IterIncrementForCleanup

The IterIncrementForCleanup property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = Empty string*

## **IterIncrementForInit**

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = Empty string*

## **IterInit**

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

*Default =*

## **IterReset**

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = Empty string*

## **IterReturnType**

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## **IterTest**

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end()

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = Empty string*

## **IterType**

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<>, as defined in the STL.

vector<\$target\*>::const\_iterator You can change the iterator type to one of your own choice.

*Default = \$(constant)\$(FixedTarget)Seq\**

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$name. The default value for all other subjects is \$name.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$(PoaPrefix)\$(MappedTarget)*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to be removed and then call erase() to remove it. vector<\$target\*>::iterator pos=find(\$name>begin(), \$name>end(), \$item); \$name>erase(pos) This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

pair<\$keyType, \$target\*> p; p.second=\$item; map<\$keyType, \$target\*>::iterator pos=find(

```
$cname->begin(), $cname->end(),p); $cname->erase(pos)
```

*Default = \$cname->remove(\$item)*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

```
$cname->clear()
```

*Default = \$cname->removeAll()*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed:

```
$cname-erase($keyName)
```

*Default = \$cname->remove(\$keyName)*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## Scalar

Defines properties for implementing scalar relations.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default* =

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector:

```
new vector<$target*>
```

*Default* =

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`:

```
vector<$target*> $cname()
```

*Default* =

## CreateStatic

The CreateStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to Reference.

*Default* =

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = \$RelationTargetType*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default =*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

\$cname->at(\$index)

*Default =*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection:  
\$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:  
\$cname-operator[](\$keyName)

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default =*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

```
$cname()
```

*Default =*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL
```

*Default =*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = OMIterator<\$RelationTargetType> \$iterator(\$cname)*

## IterCreateByValue



The `IterCreateByValue` property is the same as `IterCreate`, but instantiates the iterator by value.

*Default =*

## **IterGetCurrent**

The `IterGetCurrent` property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for `IterTest` only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for `IterGetCurrent` returns a pointer to the current item in the collection.

*Default =*

## **IterIncrement**

The `IterIncrement` property specifies the code that increments the iterator. For example, the following command moves the `$iterator` ahead one item:

`$iterator++`

*Default =*

## **IterIncrementForCleanUp**

The `IterIncrementForCleanUp` property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

*Default =*

## **IterIncrementForInit**

The `IterIncrementForInit` property specifies the code that increments the iterator, for certain initialization cases.

*Default =*

## **IterInit**

The `IterInit` property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default =*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$RelationTargetType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end()

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default =*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<>, as defined in the STL.

vector<\$target\*>::const\_iterator You can change the iterator type to one of your own choice.

*Default =*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos

\$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$name. The default value for all other subjects is \$name.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$(PoaPrefix)\$(MappedTarget)*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname>erase(pos)` This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

`pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find($cname->begin(), $cname->end(),p); $cname->erase(pos)`

*Default =*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

`$cname->clear()`

*Default =*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: \$cname-erase(\$keyName)

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

\$cname = \$item

*Default = \$cname = \$item*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## StaticArray

Defines properties for implementing static arrays.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter:

\$cname->push\_back(\$item)

For maps (qualified relations), the following command inserts an item into the map based on a key:

\$cname->insert(map<\$keyType,\$target\*>::value\_type( \$keyName,\$item))

*Default =*

\$Loop { if (!\$cname[pos]) { \$cname[pos] = \$item; break; } }

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = Empty string*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector<$target*> $cname()
```

*Default = \$CType*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default = new \$CType*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = \$RelationTargetType \$cname[\$multiplicity]*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

`$cname->find($item)`

*Default = `$cname->find($item)`*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = `$CType`*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: `$me$cname`

The variable `$me` is replaced with the object context variable as specified by the `Me` property. The variable `$cname` is replaced with the name of the container, which is the role name for the relation.

*Default = `$cname`*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the `container at()` operation to retrieve the item at the indexed position:

`$cname->at($index)`

*Default = `$cname[$index]`*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the `end()` operation for the container to locate the last item in the collection:  
`$cname-end()`

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the `CG::Relation::GetEnd` and `CG::Relation::GetEndGenerate` properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:  
\$cname-operator[](\$keyName)

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = <oxf/omlist.h>*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

\$cname()

*Default = Empty string*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL
```

*Default =*

```
$Loop { $cname[pos] = NULL; }
```

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = int \$iterator = 0;*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: *\*\$iterator*

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.



*Default = Empty string*

## **IterIncrement**

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default = Empty string*

## **IterIncrementForCleanUp**

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = Empty string*

## **IterIncrementForInit**

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = Empty string*

## **IterInit**

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

*Default =*

## **IterReset**

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = Empty string*

## **IterReturnType**

The IterReturnType property specifies the return type for functions such as getters for the various

containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end()

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = Empty string*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<>, as defined in the STL.

vector<\$target\*>::const\_iterator You can change the iterator type to one of your own choice.

*Default = \$(constant)\$(FixedTarget)Seq\**

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

Default for C++ = for (int pos = 0; pos < \$multiplicity; ++pos)

Default for C = int pos; for (pos = 0; pos < \$multiplicity; ++pos)

Default for Java = for (int pos = 0; pos < \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$name. The default value for all other subjects is \$cname.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$(PoaPrefix)\$(MappedTarget)*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(), $item); $cname->erase(pos)` This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find(
$name->begin(), $name->end(),p); $name->erase(pos)
```

*Default =*

```
$Loop { if ($cname[pos] == $item) { $cname[pos] = NULL; } }
```

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

```
$cname->clear()
```

*Default = Empty MultiLine*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

*Default =*

## SetAt

The SetAt property specifies how code is generated for the body of the mutator for a scalar container.

*Default = \$cname[\$index] = \$item*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

# UnboundedOrdered

Defines properties for implementing relations whose multiplicity is unbounded (\*) and that should be accessed sequentially.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = \$cname->add(\$item)*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = \$cname = new \$CType*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname:

```
vector<$target*> $cname()
```

*Default = \$CType \$cname*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default = new \$CType*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = OMList<\$RelationTargetType>*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default = \$cname->find(\$item)*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

`$cname->at($index)`

*Default = \$cname->getAt(\$index)*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: `$cname-end()`

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:  
\$cname-operator[](\$keyName)

*Default =*

## **IncludeDirective**

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## **IncludeFiles**

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = <oxf/omlist.h>*

## **Init**

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

\$cname()

*Default = \$cname()*

## **InitInCtorBody**

The `InitInCtorBody` property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL
```

*Default = Empty string*

## InitStatic

The `InitStatic` property defines the initialization of a relation when `CG::Relation::Containment` is set to `Value`.

*Default =*

## IterCreate

The `IterCreate` property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = OMIterator<\$RelationTargetType> \$iterator(\$cname)*

## IterCreateByValue

The `IterCreateByValue` property is the same as `IterCreate`, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The `IterGetCurrent` property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for `IterTest` only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for `IterGetCurrent` returns a pointer to the current item in the collection.

*Default =Empty string*

## IterIncrement

The `IterIncrement` property specifies the code that increments the iterator. For example, the following



command moves the \$iterator ahead one item:

```
$iterator++
```

*Default = Empty string*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = Empty string*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = Empty string*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = Empty string*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end()

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = Empty string*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<>, as defined in the STL.

vector<\$target\*>::const\_iterator You can change the iterator type to one of your own choice.

*Default = \$(constant)\$(FixedTarget)Seq\**

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$cname. The default value for all other subjects is \$cname.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$(PoaPrefix)\$(MappedTarget)*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(), $item); $cname->erase(pos)` This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

```
pair<$keyType, $target*> p; p.second=$item; map<$keyType, $target*>::iterator pos=find(
$cname->begin(), $cname->end(), p); $cname->erase(pos)
```

*Default = \$cname->remove(\$item)*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

```
$cname->clear()
```

*Default = \$cname->removeAll()*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: `$cname->erase($keyName)`

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## UnboundedUnordered

Defines properties for implementing relations whose multiplicity is unbounded (\*) and that should be accessed randomly.

### Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = \$cname->add(\$item)*

### Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = \$cname = new \$CType(\$multiplicity)*

### CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`:

```
vector<$target*> $cname()
```

*Default = \$CType \$cname(\$multiplicity)*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default = new \$CType(\$multiplicity)*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = OMCollection<\$RelationTargetType>*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default = \$cname->find(\$item)*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The

variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the container at() operation to retrieve the item at the indexed position:

`$cname->at($index)`

*Default = \$cname->getAt(\$index)*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: `$cname-end()`

This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps: `$cname-operator[]($keyName)`

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = strong*

## **IncludeFiles**

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = <oxf/omcollec.h>*

## **Init**

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

```
$cname()
```

*Default = \$cname(\$multiplicity)*

## **InitInCtorBody**

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL
```

*Default = Empty string*

## **InitStatic**

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = OMIterator<\$RelationTargetType> \$iterator(\$cname)*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = Empty string*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item:

```
$iterator++
```

*Default = Empty string*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = Empty string*

## IterIncrementForInit



The `IterIncrementForInit` property specifies the code that increments the iterator, for certain initialization cases.

*Default = Empty string*

## IterInit

The `IterInit` property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## IterReset

The `IterReset` property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = Empty string*

## IterReturnType

The `IterReturnType` property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The `IterTest` property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()`

With `STLContainers`, unlike `OMContainers`, it is possible to store a `NULL` value as the container. With `OMContainers`, `IterTest` retrieves the current item in the collection (the same as `IterGetCurrent`).

*Default = Empty string*

## IterType

The `IterType` property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<>`, as defined in the STL.

`vector<$target*>::const_iterator` You can change the iterator type to one of your own choice.

*Default = \$(constant)\$(FixedTarget)Seq\**

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$name. The default value for all other subjects is \$name.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$(PoaPrefix)\$(MappedTarget)*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to be removed and then call erase() to remove it. vector<\$target\*>::iterator pos=find(\$name>begin(), \$name>end(), \$item); \$name>erase(pos) This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

pair<\$keyType, \$target\*> p; p.second=\$item; map<\$keyType, \$target\*>::iterator pos=find(\$name->begin(), \$name->end(), p); \$name->erase(pos)

*Default = \$name->remove(\$item)*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

```
$cname->clear()
```

*Default = \$cname->removeAll()*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed:

```
$cname-erase($keyName)
```

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

```
$cname = $item
```

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## User

Defines properties for user-defined implementations of relations.

You can create your own implementations for relations by defining a new set of properties under the User metaclass. Once these are defined, you can give them permanent status by manually saving them in the factory.prp file under any other name, for example MyFaves. To complete their installation, you must add the new name as an enumerated value to the CG::Relation::Implementation property.

For example, you can change the definition of the Implementation property as follows: Subject CG Metaclass Relation Property Implementation Enum "Default,Scalar,Fixed,BoundedOrdered,BoundedUnordered,UnboundedOrdered, UnboundedUnordered,Qualified,MyFaves, User" "Default" end end

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter:

```
$cname->push_back($item)
```

For maps (qualified relations), the following command inserts an item into the map based on a key:

```
$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))
```

*Default = Empty string*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector:

```
new vector<$target*>
```

*Default = Empty string*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`:

```
vector<$target*> $cname()
```

*Default = Empty string*

## CreateStatic

The CreateStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to Reference.

*Default = Empty string*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = Empty string*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item:

```
$cname->find($item)
```

*Default = Empty string*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = Empty string*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = Empty string*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the

targets of a to-many relation that uses an index.

The `ContainerTypes::RelationType::GetAt` property specifies a template for the body of the operation. For example, the following command generates code that calls the container `at()` operation to retrieve the item at the indexed position:

```
$cname->at($index)
```

*Default = Empty string*

## GetEnd

The `GetEnd` property specifies the command used to retrieve the last item in the container. For example, the following command calls the `end()` operation for the container to locate the last item in the collection:

```
$cname-end()
```

This property and `GetEndGenerate` were created to adhere to the standard library convention for “finding” where iteration should end. `GetEnd` is generated where `Get` is generated. The method name is defined by using the `CG::Relation::GetEnd` and `CG::Relation::GetEndGenerate` properties.

*Default =*

## GetKey

The `GetKey` property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator `[]`, which has been overloaded according to the STL definition for maps:

```
$cname-operator[]($keyName)
```

*Default =*

## IncludeDirective

The `IncludeDirective` property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- `strong` - The `#include` directives are added to the header file.
- `weak` - The `#include` directives are added to the source file with forward declarations in the header file.

*Default = strong*

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = Empty string*

## **Init**

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it:

```
$cname()
```

*Default = Empty string*

## **InitInCtorBody**

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body.

For example:

```
pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL
```

*Default = Empty string*

## **InitStatic**

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## **IterCreate**

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()
```

*Default = Empty string*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = Empty string*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = Empty string*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the `$iterator` ahead one item:

```
$iterator++
```

*Default = Empty string*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

*Default = Empty string*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = Empty string*

## IterInit



The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname->begin()
```

*Default = Empty string*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = Empty string*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()`

With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = Empty string*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<>`, as defined in the STL.

`vector<$target*>::const_iterator` You can change the iterator type to one of your own choice.

*Default = Empty string*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in

the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

*Default =*

for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$name. The default value for all other subjects is \$name.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = Empty string*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname>erase(pos)` This operation applies only to "to-many" (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased.

`pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find($cname->begin(), $cname->end(),p); $cname->erase(pos)`

*Default = Empty string*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

`$cname->clear()`

*Default = Empty string*

## **RemoveKey**

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: \$cname-erase(\$keyName)

*Default =*

## **Set**

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved:

\$cname = \$item

*Default =*

## **Type**

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

# OMUContainers

---

Rational Rhapsody uses containers to implement to-many relations between objects. These include relationships of one object to many, or many objects to many.

The OMUContainers subject contains metaclasses that contain properties that control the implementing of relations.

Each property in this subject includes the default value for each container type and relation type. For easier readability, the placeholder RelationType in these values represents all the other relation types that are not explicitly detailed. For example, the relation type User might have the default value of an empty string, whereas all the other relation types have the value of an empty MultiLine.

## BoundedOrdered

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed sequentially.

### Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter: \$cname->push\_back(\$item)

For maps (qualified relations), the following command inserts an item into the map based on a key: \$cname->insert(map<\$keyType,\$target\*>::value\_type( \$keyName,\$item))

*Default = \$cname->add((void\*) \$item)*

### Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector: new vector<\$target\*>

*Default = \$cname = new \$CType*

### CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it

the name stored in \$cname: `vector<$target*> $cname()`

*Default = \$CType \$cname*

## CreateStatic

The CreateStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to Reference.

*Default = new \$CType*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the `vector<$target*>` collection type determines the type of the variable `cl` on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class `Client` is the `$target` of vector operations.

*Default = OMUList*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the `Find()` operation for container `$cname` to locate the `$item`: `$cname->find($item)`

*Default = \$cname->find((void\*) \$item)*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: `$me$cname`

The variable `$me` is replaced with the object context variable as specified by the `Me` property. The

variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the at() operation for the container to retrieve the item at the indexed position: \$cname->at(\$index)

*Default = (\$RelationTargetType) \$cname->getAt(\$index)*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps: \$cname-operator[](\$keyName)

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = weak*

## **IncludeFiles**

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = <oxf/omulist.h>*

## **Init**

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: \$cname()

*Default = \$cname()*

## **InitInCtorBody**

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: pos=0; pos<\$multiplicity; pos++; \$cname[pos]=NULL

*Default = Empty string*

## **InitSimple**

The InitSimple property contains a pattern for the appropriate container initialization call.

*Default =*

## **InitStatic**

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: `vector<$target*>::const_iterator $iterator; $iterator=$sname>begin()`

*Default = \$IterType \$iterator(\$sname)*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = (\$RelationTargetType)\*\$iterator*

## IterIncrement

The IterIncrement property (under most of the ContainerTypes::RelationType metaclasses) specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item: `$iterator++`

*Default = \$iterator++*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

*Default = \$IterReset*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.



*Default = \$IterReset*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection: `$iterator=$cname->begin()`

*Default = \$iterator.reset()*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as `IterGetCurrent()`).

*Default = \*\$iterator*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<>`, as defined in the STL. `vector<$target*>::const_iterator`

You can change the iterator type to one of your own choice.

*Default = OMUIterator*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

Default for C++ = for (int pos = 0; pos \$multiplicity; ++pos)

Default for C = int pos; for (pos = 0; pos < \$multiplicity; ++pos)

Default for Java = for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$target\$reference*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname->erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find($cname->begin(), $cname->end(),p); $cname->erase(pos)`

*Default = \$cname->remove((void\*) \$item)*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the lear() operation for the container to remove all items: `$cname->clear()`

*Default = \$cname->removeAll()*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: \$cname-erase(\$keyName)

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved: \$cname = \$item

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## BoundedUnordered

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed randomly.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter: \$cname->push\_back(\$item)

For maps (qualified relations), the following command inserts an item into the map based on a key: \$cname->insert(map<\$keyType,\$target\*>::value\_type( \$keyName,\$item))

*Default = \$cname->add((void\*) \$item)*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector: `new vector<$target*>`

*Default = \$cname = new \$CType(\$multiplicity)*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname: `vector<$target*> $cname()`

*Default = \$CType \$cname(\$multiplicity)*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default = new \$CType(\$multiplicity)*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the `vector<$target*>` collection type determines the type of the variable `cl` on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class `Client` is the \$target of vector operations.

*Default = OMUCollection*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the `Find()` operation for container \$cname to locate the \$item: `$cname->find($item)`

*Default = \$cname->find((void\*) \$item)*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the at() operation for the container to retrieve the item at the indexed position: \$cname->at(\$index)

*Default = (\$RelationTargetType) \$cname->getAt(\$index)*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps: \$cname-operator[](\$keyName)

*Default =*

## **IncludeDirective**

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = weak*

## **IncludeFiles**

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = <oxf/omucollec.h>*

## **Init**

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: \$cname()

*Default = \$cname(\$multiplicity)*

## **InitInCtorBody**

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: pos=0; pos<\$multiplicity; pos++; \$cname[pos]=NULL

*Default = Empty string*

## **InitSimple**

The InitSimple property contains a pattern for the appropriate container initialization call.

*Default =*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: `vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()`

*Default = \$IterType \$iterator(\$cname)*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = (\$RelationTargetType)\*\$iterator*

## IterIncrement

The IterIncrement property (under most of the ContainerTypes::RelationType metaclasses) specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item: `$iterator++`

*Default = \$iterator++*

## IterIncrementForCleanup

The `IterIncrementForCleanUp` property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

*Default = \$IterReset*

## IterIncrementForInit

The `IterIncrementForInit` property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterReset*

## IterInit

The `IterInit` property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## IterReset

The `IterReset` property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection: `$iterator=$cname->begin()`

*Default = \$iterator.reset()*

## IterReturnType

The `IterReturnType` property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The `IterTest` property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, `IterTest` retrieves the current item in the collection (the same as `IterGetCurrent`).

*Default = \*\$iterator*

## IterType



The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<$target*>::const_iterator`,

You can change the iterator type to one of your own choice.

*Default = OMUIterator*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

Default for C++ = `for (int pos = 0; pos $multiplicity; ++pos)`

Default for C = `int pos; for (pos = 0; pos < $multiplicity; ++pos)`

Default for Java = `for (int pos = 0; pos $multiplicity; pos++)`

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$target\$reference*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the `find()` operation to point the iterator to the item to be removed and then call `erase()` to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname>erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair `p`, whose second element, `p.second`, is the item to be removed. The `find()` operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find($cname->begin(), $cname->end(),p); $cname->erase(pos)`

*Default = \$cname->remove((void\*) \$item)*

## **RemoveAll**

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:  
\$cname->clear()

*Default = \$cname->removeAll()*

## **RemoveKey**

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed:  
\$cname-erase(\$keyName)

*Default =*

## **Set**

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved: \$cname = \$item

*Default =*

## **Type**

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## **EmbeddedFixed**

Defines properties for implementing embedded fixed relations.

## **Add**

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter: `$cname->push_back($item)`

For maps (qualified relations), the following command inserts an item into the map based on a key: `$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))`

*Default = Empty MultiLine*

## Create

The `Create` property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector: `new vector<$target*>`

*Default = Empty string*

## CreateByValue

The `CreateByValue` property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`: `vector<$target*> $cname()`

*Default = \$CType*

## CreateStatic

The `CreateStatic` property defines the initialization of a relation when `CG::Relation::Containment` is set to `Reference`.

*Default =*

## CType

The `CType` property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the `vector<$target*>` collection type determines the type of the variable `cl` on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class `Client` is the `$target` of vector operations.

*Default = \$(constant)\$target \$cname[\$multiplicity]*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item: \$cname->find(\$item)

*Default =*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$RelationTargetType \$cname[\$multiplicity]*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = &\$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the at() operation for the container to retrieve the item at the indexed position: \$cname->at(\$index)

*Default = (\$RelationTargetType) &\$cname[\$index]*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:  
\$cname-operator[](\$keyName)

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = weak*

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default =*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: \$cname()

*Default = Empty string*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: pos=0; pos<\$multiplicity; pos++; \$cname[pos]=NULL

*Default = Empty MultiLine*

## InitSimple

The InitSimple property contains a pattern for the appropriate container initialization call.

*Default =*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: vector<\$target\*>::const\_iterator \$iterator; \$iterator=\$cname>begin()

*Default = \$IterType \$iterator = 0;*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: \*\$iterator

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = ((\$RelationTargetType)&\$cname[\$iterator])*

## IterIncrement

The IterIncrement property (under most of the ContainerTypes::RelationType metaclasses) specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item: \$iterator++

*Default = \$iterator++*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = \$IterIncrement*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterIncrement*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

*Default = Blank*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection: \$iterator=\$cname->begin()

*Default = \$iterator = 0*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $name->end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = \$iterator < \$multiplicity*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<$target*>::const_iterator`

You can change the iterator type to one of your own choice.

*Default = int*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

Default for C++ = `for (int pos = 0; pos < $multiplicity; ++pos)`

Default for C = `int pos; for (pos = 0; pos < $multiplicity; ++pos)`

Default for Java = `for (int pos = 0; pos < $multiplicity; pos++)`

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$target\$reference*



## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname>erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find($cname->begin(), $cname->end(),p); $cname->erase(pos)`

*Default = Empty string*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items: `$cname->clear()`

*Default = Empty string*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved: `$cname = $item`

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## EmbeddedScalar

Defines properties for implementing embedded scalar (one-to-one) relations.

### Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter: `$cname->push_back($item)`

For maps (qualified relations), the following command inserts an item into the map based on a key: `$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))`

*Default =*

### Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector: `new vector<$target*>`

*Default =*

### CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`: `vector<$target*> $cname()`

*Default =*

### CreateStatic

The CreateStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to `Reference`.

*Default =*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the `vector<$target*>` collection type determines the type of the variable `cl` on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class `Client` is the `$target` of vector operations.

*Default = \$(constant)\$target*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the `Find()` operation for container `$cname` to locate the `$item`: `$cname->find($item)`

*Default =*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType\$reference \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: `$me$cname`

The variable `$me` is replaced with the object context variable as specified by the `Me` property. The variable `$cname` is replaced with the name of the container, which is the role name for the relation.

*Default = &\$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The `ContainerTypes::RelationType::GetAt` property specifies a template for the body of the operation. For example, the following command generates code that calls the `at()` operation for the container to retrieve

the item at the indexed position: \$cname->at(\$index)

*Default =*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps: \$cname-operator[](\$keyName)

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = weak*

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default =*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: \$cname()

*Default =*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: pos=0; pos<\$multiplicity; pos++; \$cname[pos]=NULL

*Default =*

## InitSimple

The InitSimple property contains a pattern for the appropriate container initialization call.

*Default =*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: vector<\$target\*>::const\_iterator \$iterator; \$iterator=\$cname>begin()

*Default = gen\_ptr pos; \$IterType \$iterator =*

## IterCreateByValue

The `IterCreateByValue` property is the same as `IterCreate`, but instantiates the iterator by value.

*Default =*

## **IterGetCurrent**

The `IterGetCurrent` property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for `IterTest` only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for `IterGetCurrent` returns a pointer to the current item in the collection.

*Default =*

## **IterIncrement**

The `IterIncrement` property (under most of the `ContainerTypes::RelationType` metaclasses) specifies the code that increments the iterator. For example, the following command moves the `$iterator` ahead one item: `$iterator++`

*Default =*

## **IterIncrementForCleanUp**

The `IterIncrementForCleanUp` property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

*Default =*

## **IterIncrementForInit**

The `IterIncrementForInit` property specifies the code that increments the iterator, for certain initialization cases.

*Default =*

## **IterInit**

The `IterInit` property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## **IterReset**

The `IterReset` property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection: `$iterator=$cname->begin()`

*Default =*

## IterReturnType

The `IterReturnType` property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = `$(constRT)$target*`*

## IterTest

The `IterTest` property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, `IterTest` retrieves the current item in the collection (the same as `IterGetCurrent`).

*Default =*

## IterType

The `IterType` property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<`, as defined in the STL. `vector<$target*>::const_iterator`

You can change the iterator type to one of your own choice.

*Default =*

## Loop

The `Loop` property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the `OMContainers.StaticArray` metaclass) can use any other property in the same metaclass. Therefore, other properties that use `$Loop` expand the specified `Loop` property body.

Default for C++ = `for (int pos = 0; pos $multiplicity; ++pos)`

Default for C = `int pos; for (pos = 0; pos < $multiplicity; ++pos)`

Default for Java = `for (int pos = 0; pos $multiplicity; pos++)`

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default = \$cname*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$CType\**

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname->erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find($cname->begin(), $cname->end(),p); $cname->erase(pos)`

*Default =*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items: `$cname->clear()`

*Default =*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

*Default =*

## Set



The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved: \$cname = \$item

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## Fixed

Defines properties for implementing relations of fixed size.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter: \$cname->push\_back(\$item)

For maps (qualified relations), the following command inserts an item into the map based on a key: \$cname->insert(map<\$keyType,\$target\*>::value\_type( \$keyName,\$item))

*Default = \$cname->add((void\*) \$item)*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector: new vector<\$target\*>

*Default = \$cname = new \$CType(\$multiplicity)*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it

the name stored in \$cname: `vector<$target*> $cname()`

*Default = \$CType \$cname(\$multiplicity)*

## CreateStatic

The CreateStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to Reference.

*Default = new \$CType(\$multiplicity)*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the `vector<$target*>` collection type determines the type of the variable `cl` on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class `Client` is the `$target` of vector operations.

*Default = OMUCollection*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the `Find()` operation for container `$cname` to locate the `$item`: `$cname->find($item)`

*Default = \$cname->find((void\*) \$item)*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: `$me$cname`

The variable `$me` is replaced with the object context variable as specified by the `Me` property. The

variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that calls the at() operation for the container to retrieve the item at the indexed position: \$cname->at(\$index)

*Default = (\$RelationTargetType) \$cname->getAt(\$index)*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps: \$cname-operator[](\$keyName)

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = weak*

## **IncludeFiles**

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = <oxf/omucollec.h>*

## **Init**

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: \$cname()

*Default = \$cname(\$multiplicity)*

## **InitInCtorBody**

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: pos=0; pos<\$multiplicity; pos++; \$cname[pos]=NULL

*Default = Empty string*

## **InitSimple**

The InitSimple property contains a pattern for the appropriate container initialization call.

*Default =*

## **InitStatic**

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: `vector<$target*>::const_iterator $iterator; $iterator=$sname>begin()`

*Default = \$IterType \$iterator(\$sname)*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = (\$RelationTargetType)\*\$iterator*

## IterIncrement

The IterIncrement property (under most of the ContainerTypes::RelationType metaclasses) specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item: `$iterator++`

*Default = \$iterator++*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

*Default = \$IterReset*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterReset*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection: `$iterator=$cname->begin()`

*Default = \$iterator.reset()*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as `IterGetCurrent()`).

*Default = \*\$iterator*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<>`, as defined in the STL. `vector<$target*>::const_iterator`

You can change the iterator type to one of your own choice.

*Default = OMUIterator*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

Default for C++ = for (int pos = 0; pos \$multiplicity; ++pos)

Default for C = int pos; for (pos = 0; pos < \$multiplicity; ++pos)

Default for Java = for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$target\$reference*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname->erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find($cname->begin(), $cname->end(),p); $cname->erase(pos)`

*Default = \$cname->remove((void\*) \$item)*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items: `$cname->clear()`

*Default = \$cname->removeAll()*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: \$cname-erase(\$keyName)

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved: \$cname = \$item

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## General

Defines properties that enable you to set the directives and include files for the container.

### ContainerDirectives

The ContainerDirectives property (under OMContainers::General) specifies the preprocessor directives that are necessary when compiling code that uses a particular container library.

No additional directives are required when you use OMContainers.

*Default = Empty MultiLine*

### ContainerIncludes

The ContainerIncludes property (under OMContainers::General) specifies header files that must be included when you use a particular container library.



For example, when you use STLContainers, the following string causes one `#include` directive per container to be added to generated files (such as `#include string`): `string`, `algorithm`, `vector`, `list`, `map`, `iterator` Whether the `#include` directives are added to source or header files depends on the value of the `IncludeDirective` property.

*Default = Empty string*

## Qualified

Defines properties for implementing qualified relations, which are accessed by way of a key.

### Add

The `Add` property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter: `$cname->push_back($item)`

For maps (qualified relations), the following command inserts an item into the map based on a key: `$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))`

*Default = \$cname->add((void\*)\$keyName,(void\*) \$item)*

### Create

The `Create` property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector: `new vector<$target*>`

*Default = \$cname = new \$CType*

### CreateByValue

The `CreateByValue` property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`: `vector<$target*> $cname()`

*Default = \$CType \$cname*

### CreateStatic

The `CreateStatic` property defines the initialization of a relation when `CG::Relation::Containment` is set to

Reference.

*Default = new \$CType*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = OMUMap*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item: \$cname->find(\$item)

*Default = \$cname->find((void\*) \$item)*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the

targets of a to-many relation that uses an index.

The `ContainerTypes::RelationType::GetAt` property specifies a template for the body of the operation. For example, the following command generates code that calls the `at()` operation for the container to retrieve the item at the indexed position: `$cname->at($index)`

*Default = Empty string*

## GetEnd

The `GetEnd` property specifies the command used to retrieve the last item in the container. For example, the following command calls the `end()` operation for the container to locate the last item in the collection: `$cname-end()`

This property and `GetEndGenerate` were created to adhere to the standard library convention for "finding" where iteration should end. `GetEnd` is generated where `Get` is generated. The method name is defined by using the `CG::Relation::GetEnd` and `CG::Relation::GetEndGenerate` properties.

*Default =*

## GetKey

The `GetKey` property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator `[]`, which has been overloaded according to the STL definition for maps: `$cname-operator[]($keyName)`

*Default = (\$RelationTargetType) \$cname->getKey((void\*)\$keyName)*

## IncludeDirective

The `IncludeDirective` property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The `#include` directives are added to the header file.
- weak - The `#include` directives are added to the source file with forward declarations in the header file.

*Default = weak*

## IncludeFiles

The `IncludeFiles` property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the `#include` is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = <oxf/omumap.h>*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container `$cname` to initialize it: `$cname()`

*Default = \$cname()*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: `pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL`

*Default = Empty string*

## InitSimple

The InitSimple property contains a pattern for the appropriate container initialization call.

*Default =*

## InitStatic

The InitStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: `vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()`

*Default = \$IterType \$iterator(\$cname)*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$Iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = (\$RelationTargetType)\*\$Iterator*

## IterIncrement

The IterIncrement property (under most of the ContainerTypes::RelationType metaclasses) specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item: `$iterator++`

*Default = \$iterator++*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = \$IterReset*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterReset*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection: \$iterator=\$cname->begin()

*Default = \$iterator.reset()*

## IterReturntype

The IterReturntype property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end() With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = \*\$iterator*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<, as defined in the STL. vector<\$target\*>::const\_iterator

You can change the iterator type to one of your own choice.

*Default = OMUIterator*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

Default for C++ = for (int pos = 0; pos \$multiplicity; ++pos)

Default for C = int pos; for (pos = 0; pos < \$multiplicity; ++pos)

Default for Java = for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$target\$reference*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname->erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find($cname->begin(), $cname->end(),p); $cname->erase(pos)`

*Default = \$cname->remove((void\*) \$item)*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items: `$cname->clear()`

*Default = \$cname->removeAll()*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: `$cname->erase($keyName)`

*Default = \$cname->remove((void\*)\$keyName)*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved: \$cname = \$item

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## Scalar

Defines properties for implementing scalar relations.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter: \$cname->push\_back(\$item)

For maps (qualified relations), the following command inserts an item into the map based on a key: \$cname->insert(map<\$keyType,\$target\*>::value\_type( \$keyName,\$item))

*Default =*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector: new vector<\$target\*>

*Default =*

## CreateByValue



The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname: `vector<$target*> $cname()`

*Default =*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default =*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the `vector<$target*>` collection type determines the type of the variable `cl` on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class `Client` is the \$target of vector operations.

*Default = \$(constant)\$target\$reference*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the `Find()` operation for container \$cname to locate the \$item: `$cname->find($item)`

*Default =*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$name

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$name is replaced with the name of the container, which is the role name for the relation.

*Default* = \$name

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that call the at() operation for the container to retrieve the item at the indexed position: \$name->at(\$index)

*Default* =

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$name-end()

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default* =

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps: \$name-operator[](\$keyName)

*Default* =

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = weak*

## **IncludeFiles**

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default =*

## **Init**

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: \$cname()

*Default =*

## **InitInCtorBody**

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: pos=0; pos<\$multiplicity; pos++; \$cname[pos]=NULL

*Default =*

## **InitSimple**

The InitSimple property contains a pattern for the appropriate container initialization call.

*Default =*

## **InitStatic**

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## **IterCreate**

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: `vector<$target*>::const_iterator $iterator; $iterator=$sname>begin()`

*Default =*

## **IterCreateByValue**

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default =*

## **IterGetCurrent**

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default =*

## **IterIncrement**

The IterIncrement property (under most of the ContainerTypes::RelationType metaclasses) specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item: `$iterator++`

*Default =*

## **IterIncrementForCleanUp**

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

*Default =*

## **IterIncrementForInit**

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization

cases.

*Default =*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection: `$iterator=$cname->begin()`

*Default =*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$(constRT)\$target\$reference*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as `IterGetCurrent()`).

*Default =*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<>`, as defined in the STL. `vector<$target*>::const_iterator`

You can change the iterator type to one of your own choice.

*Default =*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

Default for C++ = for (int pos = 0; pos \$multiplicity; ++pos)

Default for C = int pos; for (pos = 0; pos < \$multiplicity; ++pos)

Default for Java = for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$CType*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname>erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find($cname->begin(), $cname->end(),p); $cname->erase(pos)`

*Default =*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

`$cname->clear()`

*Default =*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed:  
`$cname-erase($keyName)`

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved: `$cname = $item`

*Default = \$cname = \$item*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## StaticArray

Defines properties for implementing static arrays.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter: `$cname->push_back($item)`

For maps (qualified relations), the following command inserts an item into the map based on a key:  
`$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))`

*Default = \$Loop { if (!\$cname[pos]) { \$cname[pos] = \$item; break; } }*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector: `new vector<$target*>`

*Default = Empty string*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname: `vector<$target*> $cname()`

*Default = \$CType*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default =*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the `vector<$target*>` collection type determines the type of the variable `cl` on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class `Client` is the \$target of vector operations.

*Default = \$RelationTargetType \$cname[\$multiplicity]*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the `Find()` operation for container \$cname to locate the \$item: `$cname->find($item)`

*Default =*

## FullTypeDefinition



The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$name

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that call the at() operation for the container to retrieve the item at the indexed position: \$cname->at(\$index)

*Default = \$cname[\$index]*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript

operator[], which has been overloaded according to the STL definition for maps:  
\$cname-operator[](\$keyName)

*Default =*

## **IncludeDirective**

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = weak*

## **IncludeFiles**

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default =*

## **Init**

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: \$cname()

*Default = Empty string*

## **InitInCtorBody**

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: pos=0; pos<\$multiplicity; pos++; \$cname[pos]=NULL

*Default = \$Loop { \$cname[pos] = NULL; }*

## InitSimple

The InitSimple property contains a pattern for the appropriate container initialization call.

*Default =*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: `vector<$target*>::const_iterator $iterator; $iterator=$sname>begin()`

*Default = \$IterType \$iterator = 0;*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = \$sname[\$iterator]*

## IterIncrement

The IterIncrement property (under most of the ContainerTypes::RelationType metaclasses) specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item: `$iterator++`

*Default = \$iterator++*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = \$IterIncrement*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterIncrement*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection: \$iterator=\$cname->begin()

*Default = \$iterator = 0*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end() With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = (\$iterator < \$multiplicity) && \$cname[\$iterator]*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<$target*>::const_iterator`,

You can change the iterator type to one of your own choice.

*Default = int*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the `OMContainers.StaticArray` metaclass) can use any other property in the same metaclass. Therefore, other properties that use `$Loop` expand the specified Loop property body.

Default for C++ = `for (int pos = 0; pos < $multiplicity; ++pos)`

Default for C = `int pos; for (pos = 0; pos < $multiplicity; ++pos)`

Default for Java = `for (int pos = 0; pos < $multiplicity; pos++)`

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$target\$reference*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the `find()` operation to point the iterator to the item to be removed and then call `erase()` to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname>erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair `p`, whose second element, `p.second`, is the item to be removed. The `find()` operation points the iterator to the position of the pair in the

collection. Finally, the item at that position is erased. `pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find($cname->begin(), $cname->end(),p); $cname->erase(pos)`

*Default = \$Loop { if (\$cname[pos] == \$item) { \$cname[pos] = NULL; } }*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the `clear()` operation for the container to remove all items: `$cname->clear()`

*Default = Empty MultiLine*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the `erase()` operation for the container, passing it the `$keyName`, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, `$cname`, is a role name of the relation itself, because there is only one class involved: `$cname = $item`

*Default =*

## SetAt

The SetAt property specifies how code is generated for the body of the mutator for a scalar container.

*Default = \$cname[\$index] = \$item*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

# UnboundedOrdered

Defines properties for implementing relations whose multiplicity is unbounded (\*) and that should be accessed sequentially.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter: `$cname->push_back($item)`

For maps (qualified relations), the following command inserts an item into the map based on a key: `$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))`

*Default = \$cname->add((void\*) \$item)*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector: `new vector<$target*>`

*Default = \$cname = new \$CType*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`: `vector<$target*> $cname()`

*Default = \$CType \$cname*

## CreateStatic

The CreateStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to `Reference`.

*Default = new \$CType*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = OMUList*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item: \$cname->find(\$item)

*Default = \$cname->find((void\*) \$item)*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that call the at() operation for the container to retrieve



the item at the indexed position: `$cname->at($index)`

*Default = (\$RelationTargetType) \$cname->getAt(\$index)*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the `end()` operation for the container to locate the last item in the collection: `$cname-end()`

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the `CG::Relation::GetEnd` and `CG::Relation::GetEndGenerate` properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator `[]`, which has been overloaded according to the STL definition for maps: `$cname-operator[]($keyName)`

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The `#include` directives are added to the header file.
- weak - The `#include` directives are added to the source file with forward declarations in the header file.

*Default = weak*

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = <oxf/omulist.h>*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: \$cname()

*Default = \$cname()*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: pos=0; pos<\$multiplicity; pos++; \$cname[pos]=NULL

*Default = Empty string*

## InitSimple

The InitSimple property contains a pattern for the appropriate container initialization call.

*Default =*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: vector<\$target\*>::const\_iterator \$iterator; \$iterator=\$cname>begin()

*Default = \$IterType \$iterator(\$cname)*

## IterCreateByValue

The `IterCreateByValue` property is the same as `IterCreate`, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The `IterGetCurrent` property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for `IterTest` only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for `IterGetCurrent` returns a pointer to the current item in the collection.

*Default = (\$RelationTargetType)\*\$iterator*

## IterIncrement

The `IterIncrement` property (under most of the `ContainerTypes::RelationType` metaclasses) specifies the code that increments the iterator. For example, the following command moves the `$iterator` ahead one item: `$iterator++`

*Default = \$iterator++*

## IterIncrementForCleanUp

The `IterIncrementForCleanUp` property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

*Default = \$IterReset*

## IterIncrementForInit

The `IterIncrementForInit` property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterReset*

## IterInit

The `IterInit` property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## IterReset

The `IterReset` property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection: `$iterator=$cname->begin()`

*Default = `$iterator.reset()`*

## IterReturn Type

The `IterReturn Type` property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = `$IterType`*

## IterTest

The `IterTest` property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, `IterTest` retrieves the current item in the collection (the same as `IterGetCurrent`).

*Default = `*$iterator`*

## IterType

The `IterType` property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<`, as defined in the STL. `vector<$target*>::const_iterator`

You can change the iterator type to one of your own choice.

*Default = `OMUIterator`*

## Loop

The `Loop` property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the `OMContainers.StaticArray` metaclass) can use any other property in the same metaclass. Therefore, other properties that use `$Loop` expand the specified `Loop` property body.

Default for C++ = `for (int pos = 0; pos < $multiplicity; ++pos)`

Default for C = `int pos; for (pos = 0; pos < $multiplicity; ++pos)`

Default for Java = `for (int pos = 0; pos < $multiplicity; pos++)`

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$target\$reference*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname>erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find($cname->begin(), $cname->end(),p); $cname->erase(pos)`

*Default = \$cname->remove((void\*) \$item)*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items: `$cname->clear()`

*Default = \$cname->removeAll()*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved: \$cname = \$item

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## UnboundedUnordered

Defines properties for implementing relations whose multiplicity is unbounded (\*) and that should be accessed randomly.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter: \$cname->push\_back(\$item)

For maps (qualified relations), the following command inserts an item into the map based on a key: \$cname->insert(map<\$keyType,\$target\*>::value\_type( \$keyName,\$item))

*Default = \$cname->add((void\*) \$item)*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector: new vector<\$target\*>

*Default = \$cname = new \$CType*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname: `vector<$target*> $cname()`

*Default = \$CType \$cname*

## CreateStatic

The CreateStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to Reference.

*Default = new \$CType*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the `vector<$target*>` collection type determines the type of the variable `cl` on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class `Client` is the \$target of vector operations.

*Default = OMUCollection*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the `Find()` operation for container \$cname to locate the \$item: `$cname->find($item)`

*Default = \$cname->find((void\*) \$item)*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: `$me$cname`

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that call the at() operation for the container to retrieve the item at the indexed position: \$cname->at(\$index)

*Default = (\$RelationTargetType) \$cname->getAt(\$index)*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps: \$cname-operator[](\$keyName)

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.



- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = weak*

## **IncludeFiles**

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = <oxf/omucollec.h>*

## **Init**

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: \$cname()

*Default = \$cname()*

## **InitInCtorBody**

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: pos=0; pos<\$multiplicity; pos++; \$cname[pos]=NULL

*Default = Empty string*

## **InitSimple**

The InitSimple property contains a pattern for the appropriate container initialization call.

*Default =*

## **InitStatic**

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: `vector<$target*>::const_iterator $iterator; $iterator=$sname>begin()`

*Default = \$IterType \$iterator(\$sname)*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = (\$RelationTargetType)\*\$iterator*

## IterIncrement

The IterIncrement property (under most of the ContainerTypes::RelationType metaclasses) specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item: `$iterator++`

*Default = \$iterator++*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

*Default = \$IterReset*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterReset*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection: `$iterator=$cname->begin()`

*Default = \$iterator.reset()*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = \$IterType*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as `IterGetCurrent()`).

*Default = \*\$iterator*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<>`, as defined in the STL. `vector<$target*>::const_iterator`

You can change the iterator type to one of your own choice.

*Default = OMUIterator*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

Default for C++ = for (int pos = 0; pos \$multiplicity; ++pos)

Default for C = int pos; for (pos = 0; pos < \$multiplicity; ++pos)

Default for Java = for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$target\$reference*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname->erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find($cname->begin(), $cname->end(),p); $cname->erase(pos)`

*Default = \$cname->remove((void\*) \$item)*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items: `$cname->clear()`

*Default = \$cname->removeAll()*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: \$cname-erase(\$keyName)

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved: \$cname = \$item

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## User

Defines properties for user-defined implementations of relations.

You can create your own implementations for relations by defining a new set of properties under the User metaclass. Once these are defined, you can give them permanent status by manually saving them in the factory.prp file under any other name, for example MyFaves. To complete their installation, you must add the new name as an enumerated value to the CG::Relation::Implementation property.

For example, you can change the definition of the Implementation property as follows:

```
Subject CG Metaclass Relation Property Implementation Enum "Default,Scalar,Fixed,BoundedOrdered,
BoundedUnordered,UnboundedOrdered, UnboundedUnordered,Qualified,MyFaves, User" "Default" end
end
```

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class

specified by \$cname, and passes the item to be added as a formal parameter: \$cname->push\_back(\$item)

For maps (qualified relations), the following command inserts an item into the map based on a key: \$cname->insert(map<\$keyType,\$target\*>::value\_type( \$keyName,\$item))

*Default = Empty string*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector: new vector<\$target\*>

*Default = Empty string*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname: vector<\$target\*> \$cname()

*Default = Empty string*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default = Empty string*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = Empty string*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item: \$cname->find(\$item)

*Default = Empty string*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = Empty string*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = Empty string*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that call the at() operation for the container to retrieve the item at the indexed position: \$cname->at(\$index)

*Default = Empty string*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:  
\$cname-operator[](\$keyName)

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = weak*

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = Empty string*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: \$cname()

*Default = Empty string*



## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: `pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL`

*Default = Empty string*

## InitSimple

The InitSimple property contains a pattern for the appropriate container initialization call.

*Default =*

## InitStatic

The InitStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: `vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()`

*Default = Empty string*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = Empty string*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = Empty string*

## IterIncrement

The IterIncrement property (under most of the ContainerTypes::RelationType metaclasses) specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item: \$iterator++

*Default = Empty string*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = Empty string*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = Empty string*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

*Default = Empty string*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection: \$iterator=\$cname->begin()

*Default = Empty string*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = Empty string*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $name->end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = Empty string*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<$target*>::const_iterator`

You can change the iterator type to one of your own choice.

*Default = Empty string*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

Default for C++ = `for (int pos = 0; pos $multiplicity; ++pos)`

Default for C = `int pos; for (pos = 0; pos < $multiplicity; ++pos)`

Default for Java = `for (int pos = 0; pos $multiplicity; pos++)`

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = Empty string*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname>erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find($cname->begin(), $cname->end(),p); $cname->erase(pos)`

*Default = Empty string*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items: `$cname->clear()`

*Default = Empty string*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved: `$cname = $item`

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default* =

# PanelDiagram

---

The PanelDiagram subject contains metaclasses that contain properties that determine the appearance and behavior of panel diagram elements.

## ButtonArray

The ButtonArray metaclass contains properties that determine the appearance and behavior of button array controls on panel diagrams.

### ButtonFont

The ButtonFont property lets you select the font to use for the text on the face of a push button control.

To change the value of the property, click the "... " button in the box next to the property value to open the Font window. The value of the property affects both buttons already on the panel diagram and new buttons added to the diagram. (The display of buttons already on the panel diagram changes only after you refresh the diagram.)

*Default = Arial 10 NoBold NoItalic*

### Direction

The Direction property determines whether the button array controls are used to input data, display data, or both. The possible values are:

- In - The button arrays are only used to input data for the attribute to which it is bound.
- Out - The button arrays are only used to display data for the attribute to which it is bound.
- InOut - The button arrays are used to input data and display data.

*Default = InOut*

### ShowName

The ShowName property determines whether or not a caption is displayed for button array elements, and if so, what text should be displayed. The possible values are:

- BindedElementFullPath - The full path of the attribute that is bound to the button array.
- BindedElement - The name of the attribute that is bound to the button array.
- Name - The name of the button array element.
- None - No text is displayed.

*Default = Name*

# DiagramFrame

The DiagramFrame metaclass contains properties that control the appearance of a diagram frame.

## DefaultSize

The DefaultSize property specifies default values for diagram frame dimensions.

*Default = 20, 20, 590, 500*

## ShowName

The ShowName property determines the text to display next to a graphic element in a diagram. For most elements, you can provide a name and a label. Your descriptive label can be useful in cases where the name itself might not be sufficient due to various constraints, such as the inability to use spaces if the name of the element is to appear in the code.

The possible values for this property vary for the different elements, as does the default value used. The possible values are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is displayed with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

Note that for some elements, such as destruction elements in sequence diagrams and timing diagrams, the property is a boolean property that hides/shows the name of the element.

*Default = Name*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## DigitalDisplay

The DigitalDisplay metaclass contains properties that determine the appearance and behavior of digital display controls on panel diagrams.

### ShowName

The ShowName property determines whether or not a caption is displayed for digital display elements, and if so, what text should be displayed. The possible values are:

- BindedElementFullPath - The full path of the attribute that is bound to the digital display.
- BindedElement - The name of the attribute that is bound to the digital display.
- Name - The name of the digital display element.
- None - No text is displayed.

*Default = Name*

## Gauge

The Gauge metaclass contains properties that determine the appearance and behavior of gauge controls on panel diagrams.

### ShowName

The ShowName property determines whether or not a caption is displayed for gauge elements, and if so, what text should be displayed. The possible values are:

- BindedElementFullPath - The full path of the attribute that is bound to the gauge.
- BindedElement - The name of the attribute that is bound to the gauge.
- Name - The name of the gauge element.
- None - No text is displayed.

*Default = Name*



## General

The General metaclass contains properties that apply to panel diagrams, in general. It does not apply to the specific controls that can be added to panel diagrams.

### AutoLaunchAnimation

The AutoLaunchAnimation property can be set for a panel diagram so that Rational Rhapsody automatically opens the diagram when the application is run in animation mode.

The following values are available:

- Never - The diagram is never opened automatically.
- Always - The diagram is always opened automatically.
- IfInScope - The diagram is opened automatically only if the panel diagram is in the scope of the active configuration.

*Default = IfInScope*

### Fillcolor

The Fillcolor property determines the color used for the background of the panel diagram. Note the following:

- If applied at the diagram level, it changes the background color of that diagram.
- If applied at the package level, it is used as the background color for all new panel diagrams in the package and also changes the background color of all existing panel diagrams in the package unless the property was set at the diagram level for a given diagram.
- Similarly, if applied at the project level, it is used as the background color for all new panel diagrams in the project, unless the property was set directly for individual packages. The selected color is used as the background color for all existing panel diagrams in the project unless the property was set at the package/diagram level for individual packages/diagrams.

*Default = 192,192,192 (RGB values)*

### SysMLDiagramKindTitle

When a diagram is created in SysML projects, the diagram frame contains a header for the diagram that includes information such as the name of the package that contains the diagram. The property SysMLDiagramKindTitle is used to specify a prefix for the diagram frame header that reflects the type of diagram. For example, when you create an internal block diagram, the header in the diagram frame starts with the prefix "ibd".

Note that the structure of the entire header is controlled by the property General::Graphics::DiagramHeader.

## Knob

The Knob metaclass contains properties that determine the appearance and behavior of knob controls on panel diagrams.

### Direction

The Direction property determines whether the knob controls are used to input data, display data, or both. The possible values are:

- In - The knobs are only used to input data for the attribute to which it is bound.
- Out - The knobs are only used to display data for the attribute to which it is bound.
- InOut - The knobs are used to input data and display data.

*Default = InOut*

### ShowName

The ShowName property determines whether or not a caption is displayed for knob elements, and if so, what text should be displayed. The possible values are:

- BindedElementFullPath - The full path of the attribute that is bound to the knob.
- BindedElement - The name of the attribute that is bound to the knob.
- Name - The name of the knob element.
- None - No text is displayed.

*Default = Name*

## Led

The LED metaclass contains properties that determine the appearance and behavior of LED controls on panel diagrams.

### ShowName

The ShowName property determines whether or not a caption is displayed for LED elements, and if so, what text should be displayed. The possible values are:

- BindedElementFullPath - The full path of the attribute that is bound to the LED.
- BindedElement - The name of the attribute that is bound to the LED.
- Name - The name of the LED element.

- None - No text is displayed.

*Default = Name*

## LevelIndicator

The LevelIndicator metaclass contains properties that determine the appearance and behavior of level indicator controls on panel diagrams.

### ShowName

The ShowName property determines whether or not a caption is displayed for level indicator elements, and if so, what text should be displayed. The possible values are:

- BindedElementFullPath - The full path of the attribute that is bound to the level indicator.
- Name - The name of the level indicator element.
- None - No text is displayed.

*Default = Name*

## MatrixDisplay

The MatrixDisplay metaclass contains properties that determine the appearance and behavior of matrix display controls on panel diagrams.

### ShowName

The ShowName property determines whether or not a caption is displayed for matrix display elements, and if so, what text should be displayed. The possible values are:

- BindedElementFullPath - The full path of the attribute that is bound to the matrix display.
- BindedElement - The name of the attribute that is bound to the matrix display.
- Name - The name of the matrix display element.
- None - No text is displayed.

*Default = Name*

## Meter

The Meter metaclass contains properties that determine the appearance and behavior of meter controls on panel diagrams.

## ShowName

The ShowName property determines whether or not a caption is displayed for meter elements, and if so, what text should be displayed. The possible values are:

- `BindedElementFullPath` - The full path of the attribute that is bound to the meter.
- `BindedElement` - The name of the attribute that is bound to the meter.
- `Name` - The name of the meter element.
- `None` - No text is displayed.

*Default = Name*

## OnOffSwitch

The OnOffSwitch metaclass contains properties that determine the appearance and behavior of on/off switch controls on panel diagrams.

### Direction

The Direction property determines whether the on/off switch controls are used to input data, display data, or both. The possible values are:

- `In` - The on/off switches are only used to input data for the attribute to which it is bound.
- `Out` - The on/off switches are only used to display data for the attribute to which it is bound.
- `InOut` - The on/off switches are used to input data and display data.

*Default = InOut*

## ShowName

The ShowName property determines whether or not a caption is displayed for on/off switch elements, and if so, what text should be displayed. The possible values are:

- `BindedElementFullPath` - The full path of the attribute that is bound to the on/off switch.
- `BindedElement` - The name of the attribute that is bound to the on/off switch.
- `Name` - The name of the on/off switch element.
- `None` - No text is displayed.

*Default = Name*

# PushButton

The PushButton metaclass contains properties that determine the appearance and behavior of push button controls on panel diagrams.

## ButtonFont

The ButtonFont property lets you select the font to use for the text on the face of a push button control.

To change the value of the property, click the "..." button in the box next to the property value to open the Font window. The value of the property affects both buttons already on the panel diagram and new buttons added to the diagram. (The display of buttons already on the panel diagram changes only after you refresh the diagram.)

*Default = Arial 10 NoBold NoItalic*

## ShowName

The ShowName property determines whether or not a caption is displayed for push button elements, and if so, what text should be displayed. The possible values are:

- BindedElementFullPath - The full path of the attribute that is bound to the push button.
- BindedElement - The name of the attribute that is bound to the push button.
- Name - The name of the push button element.
- None - No text is displayed.

*Default = Name*

# Slider

The Slider metaclass contains properties that determine the appearance and behavior of slider controls on panel diagrams.

## Direction

The Direction property determines whether slider controls are used to input data, display data, or both. The possible values are:

- In - The sliders are only used to input data for the attribute to which it is bound.
- Out - The sliders are only used to display data for the attribute to which it is bound.
- InOut - The sliders are used to input data and display data.

*Default = InOut*

## ShowName

The ShowName property determines whether or not a caption is displayed for slider elements, and if so, what text should be displayed. The possible values are:

- BindedElementFullPath - The full path of the attribute that is bound to the slider.
- BindedElement - The name of the attribute that is bound to the slider.
- Name - The name of the slider element.
- None - No text is displayed.

*Default = Name*

## TextBox

The TextBox metaclass contains properties that determine the appearance and behavior of text box controls on panel diagrams.

### Direction

The Direction property determines whether text box controls are used to input data, display data, or both. The possible values are:

- In - The text boxes are only used to input data for the attribute to which it is bound.
- Out - The text boxes are only used to display data for the attribute to which it is bound.
- InOut - The text boxes are used to input data and display data.

*Default = InOut*

## ShowName

The ShowName property determines whether or not a caption is displayed for text box elements, and if so, what text should be displayed. The possible values are:

- BindedElementFullPath - The full path of the attribute that is bound to the text box.
- BindedElement - The name of the attribute that is bound to the text box.
- Name - The name of the text box element.
- None - No text is displayed.

*Default = Name*

# QoS

---

The QoS (Quality of Service) subject contains metaclasses that contain properties that provide performance and timing information.

## Class

The Class metaclass contains properties that control the periodicity of messages, the period and jitter times, minimum interarrival times, and so on.

### AverageArrivalTime

The AverageArrivalTime property specifies the average time taken between message arrivals for an active class with an episodic (also referred to as "aperiodic") arrival pattern.

*Default = 0*

### BlockingTime

The BlockingTime property specifies the maximum amount of time that an active class (or operation) can be prohibited from executing by a lower-priority action or task. Blocking can occur when a lower priority action or class locks (in a mutually exclusive way) a resource (such as a class) that is required by a higher priority action or task.

*Default = 0*

### Deadline

The Deadline property specifies the maximum amount of time allowed for all activity by an active class resulting from a message or event reception. This property refers to the maximum amount of time allowable to complete the required response to an initiating action or event (not just handling the message per se, but also performing the required actions).

*Default = 0*

### EstExecutionTime

The EstExecutionTime property specifies the estimated time taken by active class to act on or handle a received message. Early on, the actual time might not be known. This property holds the estimate.

*Default = 0*

## ExecutionTime

The ExecutionTime property specifies the average time taken by an active class to act on or completely handle a received message. This property is redundant with the execution times of the operations involved in the execution of the behavior. By allowing this specification at the active class (thread) level, high level schedulability analysis can be performed. This is normally a measured value.

*Default = 0*

## IsPeriodic

The IsPeriodic property specifies whether an active class can be activated or initiated periodically. A message is said to have an "arrival pattern," which can be periodic or episodic. A task (or thread or active class) is said to have an activation pattern or to "be periodic." Active classes can be periodically initiated.

*Default = Cleared*

## Jitter

The Jitter property specifies the largest interval of time variance between the message reception or resulting task activation by an active class. Periodic messages are characterized by a period with which the messages arrive, and by jitter, which is the variation around the period with which messages actually arrive. Jitter is normally modeled as a uniform random process but always totally within the jitter interval.

*Default = 0*

## MinimumInterarrivalTime

The MinimumInterarrivalTime property specifies the minimum time that must occur between message arrivals for an active class with an episodic arrival pattern. Message arrivals can be episodic or periodic. An episodic arrival pattern is inherently unpredictable, but it can still be bounded. Episodic messages can have a minimum interarrival time, a minimum time that must occur between message arrivals.

*Default = 0*

## Period

The Period property specifies the average amount of time between messages received by an active class. This property applies only to an active class that is, in fact, periodically activated by those messages. An active class can receive messages and then queue them for handling later when its thread has processing focus.

*Default = 0*



## Operation

The Operation metaclass contains properties that control the estimated operation execution times, budgeted time, and blocking times.

### BlockingTime

The BlockingTime property specifies the worst case time the task can be blocked from execution, in nanoseconds (ns).

*Default = 0*

### Budget

The Budget property specifies the amount of time allocated to the worst case execution of an operation, in nanoseconds (ns).

*Default = 0*

### EstExecutionTime

The EstExecutionTime property specifies the estimated worst case execution time, in nanoseconds (ns).

*Default = 0*

### ExecutionTime

The ExecutionTime property specifies the worst case execution time, in nanoseconds (ns).

*Default = 0*

## Resource

The Resource metaclass contains a property that sets the priority ceiling of resources.

### PriorityCeiling

The PriorityCeiling property specifies the priority of the highest priority task that can lock the resource.

*Default = 0*

# ReverseEngineering

---

The ReverseEngineering subject contains metaclasses that contain language-independent properties that affect how Rational Rhapsody imports legacy code.

Rational Rhapsody also includes these language-specific subjects:

- C\_ReverseEngineering
- CPP\_ReverseEngineering
- JAVA\_ReverseEngineering

## Main

The Main metaclass contains properties that determine which legacy files are to be imported, and specify the legal format for license file names used in reverse engineering.

### EnableProgressDialog

This property determines whether or not the Enable Progress window is opened.

*Default = Cleared*

### ExcludeFilesMatching

Use this property to exclude particular files/folders from being reversed engineered. Values should be comma-separated wildcard expressions (for example: res\*, dish\*). Any files or folders that match any of these wildcard expressions is excluded from the list of files that are to be reverse engineered.

*Default = Empty string*

### Files

The Files property specifies legacy files to be imported. You select files in the Open window (Tools > Reverse Engineering > Add).

*Default = empty string*

### FilesToBeRemoved

This property is for internal Rhapsody use only. You should not modify its value.

## License

The License property specifies the location of the license for the parser used by the Reverse Engineering tool.

*Default = empty string*

## ReAnalyzeFiles

The ReAnalyzeFiles property is a Boolean value that determines whether the file that was analyzed once by the reverse engineering tool is reanalyzed during the same RE session. This property is used to improve performance.

*Default = Cleared*

## UseTreeViewByDefault

Use this property to set if the tree view should be used as the default Reverse Engineering user interface. If the value of this property is set to Checked, Rational Rhapsody displays the tree view when the Reverse Engineering tool is opened.

Note that the value of this property might be updated when you close the Reverse Engineering user interface. For example, if you are using the list view before you close the Reverse Engineering user interface, the value of this property is set to Cleared. In addition, note that the change is set for the property at the active configuration level.

*Default = Checked*

## Progress

The Progress metaclass contains properties that control how the progress of the reverse engineering operation is reported.

## AnalyzedCodeConstruct

The AnalyzedCodeConstruct property specifies the analyzed constructs on which to report. The possible values are as follows:

- Class - Report only the analyzed classes.
- File - Report only the analyzed files.
- All - Report all analyzed constructs.

*Default = Class*

## InformationApproximated

The InformationApproximated property specifies how to handle information that can only be approximated. The possible values are as follows:

- Ignore - Ignore this information.
- Report - Report the situation.
- Abort - Terminate importing.

*Default = Report*

## InformationLost

The InformationLost property specifies how to handle information that Rational Rhapsody knows is lost. The possible values are as follows:

- Ignore - Ignore this information.
- Report - Report the situation.
- Abort - Terminate importing.

*Default = Report*

## ModelUpdate

The ModelUpdate property specifies which constructs to add to the model. The possible values are as follows:

- All - Add all recognized constructs to the model.
- Class - Add only recognized classes to the model.

*Default = Class*

## ModelUpdatingFailed

The ModelUpdatingFailed property specifies how to handle a failed import. The possible values are as follows:

- Ignore - Ignore this information.
- Report - Report the situation.
- Abort - Terminate importing.

*Default = Report*

## OutputFile

The OutputFile property specifies the name of the log file to which status and error messages are written

during the import process. The Log and Process options in the Reverse Engineering Options window determine which conditions are reported. The same messages are simultaneously written to the output window and the log file.

*Default = ReverseEngineering.log*

## OutputWindow

The OutputWindow property specifies which reverse engineering messages are written to the output window. This can help speed up performance. The possible values are as follows:

- None - No messages are written to the output window.
- File - The names of processed files are written to the output window.
- Error - Only errors are written to the output window.
- All - All messages are written to the output window.

*Default = File*

## ParsingError

The ParsingError property specifies how to report progress errors. The possible values are as follows:

- Ignore - Ignore this information.
- Report - Report the situation.
- Abort - Terminate importing.

*Default = Report*

## TimeStampPerFile

The TimeStampPerFile property is a Boolean value that allows you to specify that during reverse engineering Rational Rhapsody should include timestamps in the log files to indicate when each file was reverse engineered. If the value is set to Checked, timestamps is included.

*Default = Cleared*

## Update

The Update metaclass contains a property that controls whether imported packages and classes are merged or overwritten.

## CreateObjectModelDiagrams

When you reverse engineer code with Rational Rhapsody, you have the option of specifying that Rational Rhapsody should automatically generate object model diagrams based on the code imported. The Model Updating tab of the Reverse Engineering Options window contains a check box labeled "Populate Object Model Diagram" that can be selected to activate this option.

This feature is controlled by the `CreateObjectModelDiagrams` property.

Note that when you select this option in the Reverse Engineering Options window, it modifies the value of this property for the currently-active Configuration.

*Default = Checked*

## **IgnoreOperationBody**

You can use the property `IgnoreOperationBody` to specify that operations should be imported into a model during reverse engineering without importing the bodies of the operations. For constructors, the initializer list as well will not be imported.

This property also affects roundtripping. Any changes made to operation bodies (or to initializer lists of constructors) will not be brought into the model during roundtripping.

The property is set at the Configuration level.

*Default = False*

## **ObjectModelDiagramMaxElements**

The reverse engineering feature includes an option of having Rational Rhapsody create object model diagrams to reflect the classes that are imported into the model. The property `ObjectModelDiagramMaxElements` can be used to specify that even when this option is selected, an object model diagram should not be created if it would end up containing more than a certain number of graphic elements. If the number of elements is greater than the value of `ObjectModelDiagramMaxElements`, the diagram is not created.

*Default = 50*

## **Policy**

The `Policy` property specifies whether imported packages/classes should overwrite or be merged with existing ones.

*Default = Overwrite*

# RiCContainers

---

Rational Rhapsody uses containers to implement to-many relations between objects. These include relationships of one object to many, or many objects to many.

The RiCContainers subject contains the metaclasses that contain the properties that control the implementing of relations.

Each property in this subject includes the default value for each container type and relation type. For easier readability, the placeholder RelationType in these values represents all the other relation types that are not explicitly detailed. For example, the relation type User might have the default value of an empty string, whereas all the other relation types have the value of an empty MultiLine.

## BoundedOrdered

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed sequentially.

### Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter: `$cname->push_back($item)`

For maps (qualified relations), the following command inserts an item into the map based on a key: `$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))`

*Default = \$(CType)\_addHead(&(\$me\$cname), \$item)*

### Cast

The Cast property specifies the target.

*Default = (\$target\$reference)*

### CastRT

This property defines the return type casting in an iterator creation call. The user can empty it, as when generating MISRA compliant code.

*Default = (\$IterType)*



## Cleanup

The Cleanup property contains a pattern for the appropriate container destructor call.

*Default = \$(CType)\_Cleanup(&(\$me\$name))*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector: `new vector<$target*>`

*Default = \$me\$name = \$(CType)\_Create()*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$name: `vector<$target*> $name()`

*Default = \$(CType) \$name*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default =*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the `vector<$target*>` collection type determines the type of the variable `cl` on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class `Client` is the \$target of vector operations.

*Default = RiCList*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item: \$cname->find(\$item)

*Default = \$(CType)\_find(&(\$me\$cname), \$item)*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$me\$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that call the at() operation for the container to retrieve the item at the indexed position: \$cname->at(\$index)

*Default = \$(CType)\_getAt(&(\$me\$cname), \$index)*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:  
`$cname-operator[]($keyName)`

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = weak*

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = <oxf/RiCList.h>*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: `$cname()`

*Default = \$(CType)\_Init(&(\$me\$cname));*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: `pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL`

*Default = Empty string*

## InitSimple

The InitSimple property contains a pattern for the appropriate container initialization call.

*Default =*

## InitStatic

The InitStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: `vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()`

*Default = `gen_ptr pos; $IterType $iterator = $CastRT&($me$cname)`*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = `$IterCreate`*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = `$(CType)_get($iterator, pos)`*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item: \$iterator++

*Default = \$(CType)\_next(\$iterator, &pos)*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = \$IterReset*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterIncrement*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

*Default = \$IterReset*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection: \$iterator=\$cname->begin()

*Default = \$(CType)\_first(\$iterator, &pos)*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = Empty string*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = !\$ (CType)\_isDone(\$iterator, pos)*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<$target*>::const_iterator`,

You can change the iterator type to one of your own choice.

*Default = \$(CType) \**

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

Default for C++ = `for (int pos = 0; pos $multiplicity; ++pos)`

Default for C = `int pos; for (pos = 0; pos < $multiplicity; ++pos)`

Default for Java = `for (int pos = 0; pos $multiplicity; pos++)`

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$target\$reference*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the `find()` operation to point the iterator to the item to be removed and then call `erase()` to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname->erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair `p`, whose second element, `p.second`, is the item to be removed. The `find()` operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find($cname->begin(), $cname->end(),p); $cname->erase(pos)`

*Default = \$(CType)\_remove(&(\$me\$cname), \$item)*

## RemoveAll

The `RemoveAll` property specifies the command used to remove all items from the container. For example, the following command calls the `clear()` operation for the container to remove all items: `$cname->clear()`

*Default = \$(CType)\_removeAll(&(\$me\$cname))*

## RemoveKey

The `RemoveKey` property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the `erase()` operation for the container, passing it the `$keyName`, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

*Default =*

## Set

The `Set` property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, `$cname`, is a role name of the relation itself, because there is only one class involved: `$cname = $item`

*Default =*

## Type

The `Type` property specifies the type of the container as a pointer to the relation.

*Default =*

# BoundedUnordered

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed randomly.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter: `$cname->push_back($item)`

For maps (qualified relations), the following command inserts an item into the map based on a key: `$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))`

*Default = `$(CType)_add(&($me$cname), $item)`*

## Cast

The Cast property specifies the target.

*Default = `($target$reference)`*

## CastRT

This property defines the return type casting in an iterator creation call. The user can empty it, as when generating MISRA compliant code.

*Default = `($IterType)`*

## Cleanup

The Cleanup property contains a pattern for the appropriate container destructor call.

*Default = `$(CType)_Cleanup(&($me$cname))`*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector: `new vector<$target*>`



*Default = \$me\$name = \$(CType)\_Create(\$multiplicity)*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname: `vector<$target*> $cname()`

*Default = \$(CType) \$cname*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default =*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the `vector<$target*>` collection type determines the type of the variable `cl` on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class `Client` is the \$target of vector operations.

*Default = RiCCollection*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the `Find()` operation for container \$cname to locate the \$item: `$cname->find($item)`

*Default = \$(CType)\_find(&(\$me\$name), \$item)*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: `$me$name`

The variable `$me` is replaced with the object context variable as specified by the Me property. The variable `$name` is replaced with the name of the container, which is the role name for the relation.

*Default = `$me$name`*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The `ContainerTypes::RelationType::GetAt` property specifies a template for the body of the operation. For example, the following command generates code that call the `at()` operation for the container to retrieve the item at the indexed position: `$name->at($index)`

*Default = `$(CType)_getAt(&($me$name), $index)`*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the `end()` operation for the container to locate the last item in the collection: `$name-end()`

This property and `GetEndGenerate` were created to adhere to the standard library convention for "finding" where iteration should end. `GetEnd` is generated where `Get` is generated. The method name is defined by using the `CG::Relation::GetEnd` and `CG::Relation::GetEndGenerate` properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator `[]`, which has been overloaded according to the STL definition for maps: `$name-operator[]($keyName)`

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = weak*

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = <oxf/RiCCollection.h>*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: \$cname()

*Default = \$(CType)\_Init(&(\$me\$cname), \$multiplicity);*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: pos=0; pos<\$multiplicity; pos++; \$cname[pos]=NULL

*Default = Empty string*

## InitSimple

The InitSimple property contains a pattern for the appropriate container initialization call.

*Default =*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: `vector<$target*>::const_iterator $iterator; $iterator=$sname>begin()`

*Default = unsigned pos; \$IterType \$iterator = \$CastRT&(\$me\$sname)*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = \$(CType)\_get(\$iterator, pos)*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item: `$iterator++`

*Default = \$(CType)\_next(\$iterator, &pos)*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = \$IterIncrement*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterIncrement*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default = \$IterReset*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection: `$iterator=$cname->begin()`

*Default = \$(CType)\_first(\$iterator, &pos)*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = Empty string*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as `IterGetCurrent()`).

*Default = !\$(CType)\_isDone(\$iterator, pos)*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<`, as defined in the STL. `vector<$target*>::const_iterator`

You can change the iterator type to one of your own choice.

*Default = \$(CType) \**

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

Default for C++ = for (int pos = 0; pos \$multiplicity; ++pos)

Default for C = int pos; for (pos = 0; pos < \$multiplicity; ++pos)

Default for Java = for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$target\$reference*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(), $item); $cname>erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair<$keyType, $target*> p; p.second=$item; map<$keyType, $target*>::iterator pos=find($cname->begin(), $cname->end(), p); $cname->erase(pos)`

*Default = \$(CType)\_remove(&(\$me\$cname), \$item)*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items: \$cname->clear()

*Default* = \$(CType)\_removeAll(&(\$me\$cname))

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: \$cname-erase(\$keyName)

*Default* =

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved: \$cname = \$item

*Default* =

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default* =

## EmbeddedFixed

Defines properties for implementing embedded fixed relations.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter: \$cname->push\_back(\$item)

For maps (qualified relations), the following command inserts an item into the map based on a key:  
`$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))`

*Default = Empty MultiLine*

## Cast

The Cast property specifies the target.

*Default = (\$target\$reference)*

## CastRT

This property defines the return type casting in an iterator creation call. The user can empty it, as when generating MISRA compliant code.

*Default = (\$IterType)*

## Cleanup

The Cleanup property contains a pattern for the appropriate container destructor call.

*Default = \$(CType)\_Cleanup(&(\$me\$cname))*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector: `new vector<$target*>`

*Default = Empty string*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname: `vector<$target*> $cname()`

*Default = \$(CType)*

## CreateStatic



The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default =*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<Target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = \$(constant)\$target \$cname[\$multiplicity]*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item: \$cname->find(\$item)

*Default = \$(CType)\_find(&(\$me\$cname), \$item)*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$RelationTargetType \$cname[\$multiplicity]*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = &(\$me\$cname)*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that call the at() operation for the container to retrieve the item at the indexed position: \$cname->at(\$index)

*Default = &((\$me\$cname)[\$index])*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps: \$cname-operator[](\$keyName)

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = weak*

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation

implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = <oxf/RiCList.h>*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: \$cname()

*Default = Empty string*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: pos=0; pos<\$multiplicity; pos++; \$cname[pos]=NULL

*Default = Empty string*

## InitSimple

The InitSimple property contains a pattern for the appropriate container initialization call.

*Default = \$Loop { \$target\_ctor(&((\$me\$cname)[pos])); }*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: vector<\$target\*>::const\_iterator \$iterator; \$iterator=\$cname>begin()

*Default = \$IterType \$iterator = 0;*

## **IterCreateByValue**

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## **IterGetCurrent**

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = &((\$me\$name)[\$iterator])*

## **IterIncrement**

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item: `$iterator++`

*Default = \$iterator++*

## **IterIncrementForCleanUp**

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

*Default = \$IterIncrement*

## **IterIncrementForInit**

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterIncrement*

## **IterInit**

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default = Empty string*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection: \$iterator=\$cname->begin()

*Default = \$iterator = 0*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = Empty string*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end() With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = \$iterator < \$multiplicity*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<, as defined in the STL. vector<\$target\*>::const\_iterator

You can change the iterator type to one of your own choice.

*Default = int*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

Default for C++ = for (int pos = 0; pos < \$multiplicity; ++pos)

Default for C = int pos; for (pos = 0; pos < \$multiplicity; ++pos)

Default for Java = for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$target\$reference*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname>erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find($cname->begin(), $cname->end(),p); $cname->erase(pos)`

*Default = Empty MultiLine*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items: `$cname->clear()`

*Default = \$Loop { \$target\_Cleanup(&((\$me\$cname)[pos])); }*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved: \$cname = \$item

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

# EmbeddedScalar

Defines properties for implementing embedded scalar (one-to-one) relations.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter: \$cname->push\_back(\$item)

For maps (qualified relations), the following command inserts an item into the map based on a key: \$cname->insert(map<\$keyType,\$target\*>::value\_type( \$keyName,\$item))

*Default =*

## Cast

The Cast property specifies the target.

*Default = (\$target\*)*

## CastRT

This property defines the return type casting in an iterator creation call. The user can empty it, as when

generating MISRA compliant code.

*Default =*

## Cleanup

The Cleanup property contains a pattern for the appropriate container destructor call.

*Default =*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector: `new vector<$target*>`

*Default =*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname: `vector<$target*> $cname()`

*Default =*

## CreateStatic

The CreateStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to Reference.

*Default =*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the `vector<$target*>` collection type determines the type of the variable `cl` on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class `Client` is the \$target of vector operations.

*Default = \$(constant)\$target*



## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item: \$cname->find(\$item)

*Default =*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType\$(reference) \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = &(\$me\$cname)*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that call the at() operation for the container to retrieve the item at the indexed position: \$cname->at(\$index)

*Default =*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by

using the `CG::Relation::GetEnd` and `CG::Relation::GetEndGenerate` properties.

*Default =*

## GetKey

The `GetKey` property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator `[]`, which has been overloaded according to the STL definition for maps:  
`$cname-operator[]($keyName)`

*Default =*

## IncludeDirective

The `IncludeDirective` property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The `#include` directives are added to the header file.
- weak - The `#include` directives are added to the source file with forward declarations in the header file.

*Default = weak*

## IncludeFiles

The `IncludeFiles` property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the `Containment` property is set to `Reference`, a forward declaration of the container is added to the class specification file, and the `#include` is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = Empty string*

## Init

The `Init` property specifies the command used to initialize the container. For example, the following command calls the constructor for the container `$cname` to initialize it: `$cname()`

*Default =*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: pos=0; pos<\$multiplicity; pos++; \$cname[pos]=NULL

*Default =*

## InitSimple

The InitSimple property contains a pattern for the appropriate container initialization call.

*Default = \$target\_ctor(&(\$me\$(cname)))*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: vector<\$target\*>::const\_iterator \$iterator; \$iterator=\$cname>begin()

*Default = gen\_ptr pos; \$IterType \$iterator =*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default =*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: \*\$iterator

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default =*

## **IterIncrement**

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item: \$iterator++

*Default =*

## **IterIncrementForCleanUp**

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default =*

## **IterIncrementForInit**

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default =*

## **IterInit**

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

*Default =*

## **IterReset**

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection: \$iterator=\$cname->begin()

*Default =*

## **IterReturnType**

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = Empty string*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $name->end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default =*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<$target*>::const_iterator`

You can change the iterator type to one of your own choice.

*Default =*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

Default for C++ = `for (int pos = 0; pos $multiplicity; ++pos)`

Default for C = `int pos; for (pos = 0; pos < $multiplicity; ++pos)`

Default for Java = `for (int pos = 0; pos $multiplicity; pos++)`

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default = \$me\$name*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$CType\**

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname>erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find($cname->begin(), $cname->end(),p); $cname->erase(pos)`

*Default =*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items: `$cname->clear()`

*Default =*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved: `$cname = $item`

*Default = memcpy((void\*)&(\$me\$cname),(void\*)\$item, sizeof(\$target))*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## Fixed

Defines properties for implementing relations of fixed size.

### Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter: `$cname->push_back($item)`

For maps (qualified relations), the following command inserts an item into the map based on a key: `$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))`

*Default = \$(CType)\_add(&(\$me\$cname), \$item)*

### Cast

The Cast property specifies the target.

*Default = (\$target\$reference)*

### CastRT

This property defines the return type casting in an iterator creation call. The user can empty it, as when generating MISRA compliant code.

*Default = (\$IterType)*

### Cleanup

The Cleanup property contains a pattern for the appropriate container destructor call.

*Default = \$(CType)\_Cleanup(&(\$me\$cname))*

### Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and

returns a pointer to the vector: `new vector<$target*>`

*Default = `$me$name = $(CType)_Create($multiplicity)`*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$name: `vector<$target*> $name()`

*Default = `$(CType) $name`*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default =*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the `vector<$target*>` collection type determines the type of the variable `cl` on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class `Client` is the \$target of vector operations.

*Default = `RiCCollection`*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the `Find()` operation for container \$name to locate the \$item: `$name->find($item)`

*Default = `$(CType)_find(&($me$name), $item)`*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = `$CType $name`*



## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: `$me$name`

The variable `$me` is replaced with the object context variable as specified by the Me property. The variable `$name` is replaced with the name of the container, which is the role name for the relation.

*Default = `$me$name`*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The `ContainerTypes::RelationType::GetAt` property specifies a template for the body of the operation. For example, the following command generates code that call the `at()` operation for the container to retrieve the item at the indexed position: `$name->at($index)`

*Default = `$(CType)_getAt(&($me$name), $index)`*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the `end()` operation for the container to locate the last item in the collection: `$name-end()`

This property and `GetEndGenerate` were created to adhere to the standard library convention for "finding" where iteration should end. `GetEnd` is generated where `Get` is generated. The method name is defined by using the `CG::Relation::GetEnd` and `CG::Relation::GetEndGenerate` properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator `[]`, which has been overloaded according to the STL definition for maps: `$name-operator[]($keyName)`

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = weak*

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = <oxf/RiCCollection.h>*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: \$cname()

*Default = \$(CType)\_Init(&(\$me\$cname), \$multiplicity);*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: pos=0; pos<\$multiplicity; pos++; \$cname[pos]=NULL

*Default = \$(CType)\_setFixedSize(&(\$me\$cname), RiCTRUE)*

## InitSimple

The InitSimple property contains a pattern for the appropriate container initialization call.

*Default =*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: `vector<$target*>::const_iterator $iterator; $iterator=$sname>begin()`

*Default = unsigned pos; \$IterType \$iterator = \$CastRT&(\$me\$sname)*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = \$(CType)\_get(\$iterator, pos)*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item: `$iterator++`

*Default = \$(CType)\_next(\$iterator, &pos)*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = \$IterIncrement*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterIncrement*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default = \$IterReset*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection: `$iterator=$cname->begin()`

*Default = \$(CType)\_first(\$iterator, &pos)*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = Empty string*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as `IterGetCurrent()`).

*Default = !\$(CType)\_isDone(\$iterator, pos)*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<`, as defined in the STL. `vector<$target*>::const_iterator`

You can change the iterator type to one of your own choice.

*Default = \$(CType) \**

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

Default for C++ = for (int pos = 0; pos \$multiplicity; ++pos)

Default for C = int pos; for (pos = 0; pos < \$multiplicity; ++pos)

Default for Java = for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$target\$reference*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(), $item); $cname>erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find($cname->begin(), $cname->end(),p); $cname->erase(pos)`

*Default = \$(CType)\_remove(&(\$me\$cname), \$item)*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

```
$cname->clear()
```

*Default* = `$(CType)_removeAll(&($me$cname))`

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed:

```
$cname-erase($keyName)
```

*Default* =

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved: \$cname = \$item

*Default* =

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default* =

## General

Defines properties that enable you to set the directives and include files for the container.

## ContainerDirectives

The ContainerDirectives property (under OMContainers::General) specifies the preprocessor directives that are necessary when compiling code that uses a particular container library.

No additional directives are required when you use OMContainers.

*Default = Empty MultiLine*

## ContainerIncludes

The ContainerIncludes property (under OMContainers::General) specifies header files that must be included when you use a particular container library.

For example, when you use STLContainers, the following string causes one #include directive per container to be added to generated files (such as #include string): string, algorithm, vector, list, map, iterator Whether the #include directives are added to source or header files depends on the value of the IncludeDirective property.

*Default = Empty string*

## Qualified

Defines properties for implementing qualified relations, which are accessed by way of a key.

### Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter: \$cname->push\_back(\$item)

For maps (qualified relations), the following command inserts an item into the map based on a key: \$cname->insert(map<\$keyType,\$target\*>::value\_type( \$keyName,\$item))

*Default = \$(CType)\_add(&(\$me\$cname), (gen\_ptr)\$keyName, \$item)*

### Cast

The Cast property specifies the target.

*Default = (\$target\$reference)*

### CastRT

This property defines the return type casting in an iterator creation call. The user can empty it, as when generating MISRA compliant code.

*Default = (\$IterType)*

## Cleanup

The Cleanup property contains a pattern for the appropriate container destructor call.

*Default = \$(CType)\_Cleanup(&(\$me\$name))*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector: `new vector<$target*>`

*Default = \$me\$name = \$(CType)\_Create(NULL)*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$name: `vector<$target*> $name()`

*Default = \$(CType) \$name*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default =*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the `vector<$target*>` collection type determines the type of the variable `cl` on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class `Client` is the \$target of vector operations.

*Default = RiCMap*

## Find



The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item: \$cname->find(\$item)

*Default = \$(CType)\_find(&(\$me\$cname), \$item)*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$me\$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that call the at() operation for the container to retrieve the item at the indexed position: \$cname->at(\$index)

*Default = \$(CType)\_getAt(&(\$me\$cname), \$index)*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:  
`$cname-operator[]($keyName)`

*Default = \$(CType)\_getKey(&(\$me\$cname), (gen\_ptr)\$keyName)*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = weak*

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = <oxf/RiCMap.h>*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: `$cname()`

*Default = \$(CType)\_Init(&(\$me\$cname), NULL);*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: `pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL`

*Default = Empty string*

## InitSimple

The InitSimple property contains a pattern for the appropriate container initialization call.

*Default =*

## InitStatic

The InitStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: `vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()`

*Default = `gen_ptr pos; $IterType $iterator = $CastRT&($me$cname)`*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = `$IterCreate`*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = `$(CType)_get($iterator, pos)`*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item: \$iterator++

*Default = \$(CType)\_next(\$iterator, &pos)*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = \$IterIncrement*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterIncrement*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

*Default = \$IterReset*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection: \$iterator=\$cname->begin()

*Default = \$(CType)\_first(\$iterator, &pos)*

## IterReturnTypes

The IterReturnTypes property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = Empty string*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = `!$(CType)_isDone($iterator, pos)`*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<$target*>::const_iterator`,

You can change the iterator type to one of your own choice.

*Default = `$(CType) *`*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

Default for C++ = `for (int pos = 0; pos $multiplicity; ++pos)`

Default for C = `int pos; for (pos = 0; pos < $multiplicity; ++pos)`

Default for Java = `for (int pos = 0; pos $multiplicity; pos++)`

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = `$(constant)$target$reference`*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the `find()` operation to point the iterator to the item to item to be removed and then call `erase()` to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname->erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair `p`, whose second element, `p.second`, is the item to be removed. The `find()` operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find($cname->begin(), $cname->end(),p); $cname->erase(pos)`

*Default = \$(CType)\_remove(&(\$me\$cname), \$item)*

## RemoveAll

The `RemoveAll` property specifies the command used to remove all items from the container. For example, the following command calls the `clear()` operation for the container to remove all items: `$cname->clear()`

*Default = \$(CType)\_removeAll(&(\$me\$cname))*

## RemoveKey

The `RemoveKey` property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the `erase()` operation for the container, passing it the `$keyName`, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

*Default = \$(CType)\_removeKey(&(\$me\$cname), (gen\_ptr)\$keyName)*

## Set

The `Set` property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, `$cname`, is a role name of the relation itself, because there is only one class involved: `$cname = $item`

*Default =*

## Type

The `Type` property specifies the type of the container as a pointer to the relation.

*Default =*

# Scalar

Defines properties for implementing scalar relations.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter: `$cname->push_back($item)`

For maps (qualified relations), the following command inserts an item into the map based on a key: `$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))`

*Default =*

## Cast

The Cast property specifies the target.

*Default = (\$target\$reference)*

## CastRT

This property defines the return type casting in an iterator creation call. The user can empty it, as when generating MISRA compliant code.

*Default =*

## Cleanup

The Cleanup property contains a pattern for the appropriate container destructor call.

*Default =*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector: `new vector<$target*>`

*Default =*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname: `vector<$target*> $cname()`

*Default =*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default =*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the `vector<$target*>` collection type determines the type of the variable `cl` on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class `Client` is the \$target of vector operations.

*Default = \$RelationTargetType*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the `Find()` operation for container \$cname to locate the \$item: `$cname->find($item)`

*Default =*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*



## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$name

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$me\$name*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that call the at() operation for the container to retrieve the item at the indexed position: \$cname->at(\$index)

*Default =*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps: \$cname-operator[](\$keyName)

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = weak*

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default =*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: \$cname()

*Default =*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: pos=0; pos<\$multiplicity; pos++; \$cname[pos]=NULL

*Default =*

## InitSimple

The InitSimple property contains a pattern for the appropriate container initialization call.

*Default =*

## **InitStatic**

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## **IterCreate**

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: `vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()`

*Default =*

## **IterCreateByValue**

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default =*

## **IterGetCurrent**

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default =*

## **IterIncrement**

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item: `$iterator++`

*Default =*

## **IterIncrementForCleanUp**

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

*Default =*

## **IterIncrementForInit**

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default =*

## **IterInit**

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## **IterReset**

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection: `$iterator=$cname->begin()`

*Default =*

## **IterReturnType**

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = Empty string*

## **IterTest**

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as `IterGetCurrent()`).

*Default =*

## **IterType**

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<>`, as defined in the STL. `vector<$target*>::const_iterator`

You can change the iterator type to one of your own choice.

*Default =*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

Default for C++ = for (int pos = 0; pos \$multiplicity; ++pos)

Default for C = int pos; for (pos = 0; pos < \$multiplicity; ++pos)

Default for Java = for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$target\$reference*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname>erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find($cname->begin(), $cname->end(),p); $cname->erase(pos)`

*Default =*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

```
$cname->clear()
```

*Default =*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed:

```
$cname-erase($keyName)
```

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved: \$cname = \$item

*Default = \$me\$cname = \$item*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## StaticArray

Defines properties for implementing static arrays.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter: \$cname->push\_back(\$item)

For maps (qualified relations), the following command inserts an item into the map based on a key:  
`$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))`

*Default = \$Loop { if (!\$me\$cname[pos]) { \$me\$cname[pos] = \$item; break; } }*

## Cast

The Cast property specifies the target.

*Default = (\$target\$reference)*

## CastRT

This property defines the return type casting in an iterator creation call. The user can empty it, as when generating MISRA compliant code.

*Default =*

## Cleanup

The Cleanup property contains a pattern for the appropriate container destructor call.

*Default = Empty MultiLine*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector: `new vector<$target*>`

*Default = Empty string*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname: `vector<$target*> $cname()`

*Default = \$(CType)*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default =*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = \$RelationTargetType\$cname[\$multiplicity]*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item: \$cname->find(\$item)

*Default =*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$me\$cname*

## GetAt



The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that call the at() operation for the container to retrieve the item at the indexed position: \$cname->at(\$index)

*Default = \$me\$cname[\$index]*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps: \$cname-operator[](\$keyName)

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = weak*

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation

implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default =*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: \$cname()

*Default = Empty string*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: pos=0; pos<\$multiplicity; pos++; \$cname[pos]=NULL

*Default = \$Loop { \$me\$cname[pos] = NULL; }*

## InitSimple

The InitSimple property contains a pattern for the appropriate container initialization call.

*Default =*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: vector<\$target\*>::const\_iterator \$iterator; \$iterator=\$cname>begin()

*Default = gen\_ptr pos; \$IterType \$iterator = \$IterType \$iterator = 0;*

## **IterCreateByValue**

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## **IterGetCurrent**

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = \$me\$name[\$iterator]*

## **IterIncrement**

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item: `$iterator++`

*Default = \$iterator++*

## **IterIncrementForCleanUp**

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = \$IterIncrement*

## **IterIncrementForInit**

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterIncrement*

## **IterInit**

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default = Empty string*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection: \$iterator=\$cname->begin()

*Default = \$iterator = 0*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = Empty string*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end() With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = (\$iterator < \$multiplicity) && ((\$me\$cname)[\$iterator])*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<, as defined in the STL. vector<\$target\*>::const\_iterator

You can change the iterator type to one of your own choice.

*Default = int*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

Default for C++ = for (int pos = 0; pos < \$multiplicity; ++pos)

Default for C = int pos; for (pos = 0; pos < \$multiplicity; ++pos)

Default for Java = for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$target\$reference*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname>erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find($cname->begin(), $cname->end(),p); $cname->erase(pos)`

*Default = \$Loop { if (\$me\$cname[pos] == \$item) { \$me\$cname[pos] = NULL; } }*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items: `$cname->clear()`

*Default = \$Loop { \$me\$cname[pos] = NULL; }*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved: \$cname = \$item

*Default =*

## SetAt

The SetAt property specifies how code is generated for the body of the mutator for a scalar container.

*Default = \$cname[\$index] = \$item*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

# UnboundedOrdered

Defines properties for implementing relations whose multiplicity is unbounded (\*) and that should be accessed sequentially.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter: \$cname->push\_back(\$item)

For maps (qualified relations), the following command inserts an item into the map based on a key: \$cname->insert(map<\$keyType,\$target\*>::value\_type( \$keyName,\$item))

*Default = \$(CType)\_addHead(&(\$me\$cname), \$item)*

## Cast

The Cast property specifies the target.

*Default = (\$target\$reference)*

## CastRT

This property defines the return type casting in an iterator creation call. The user can empty it, as when generating MISRA compliant code.

*Default = (\$IterType)*

## Cleanup

The Cleanup property contains a pattern for the appropriate container destructor call.

*Default = \$(CType)\_Cleanup(&(\$me\$name))*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector: `new vector<$target*>`

*Default = \$me\$name = \$(CType)\_Create()*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname: `vector<$target*> $cname()`

*Default = \$(CType) \$cname*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default =*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = RiCList*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item: \$cname->find(\$item)

*Default = \$(CType)\_find(&(\$me\$cname), \$item)*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$me\$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that call the at() operation for the container to retrieve the item at the indexed position: \$cname->at(\$index)

*Default = \$(CType)\_getAt(&(\$me\$cname), \$index)*



## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$name-end()

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps: \$name-operator[](\$keyName)

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = weak*

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = <oxf/RiCList.h>*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: \$cname()

*Default = \$(CType)\_Init(&(\$me\$cname));*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: pos=0; pos<\$multiplicity; pos++; \$cname[pos]=NULL

*Default = Empty string*

## InitSimple

The InitSimple property contains a pattern for the appropriate container initialization call.

*Default =*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: vector<\$target\*>::const\_iterator \$iterator; \$iterator=\$cname>begin()

*Default = gen\_ptr pos; \$IterType \$iterator = \$CastRT&(\$me\$cname)*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = `$(CType)_get($iterator, pos)`*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the `$iterator` ahead one item: `$iterator++`

*Default = `$(CType)_next($iterator, &pos)`*

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

*Default = `$IterReset`*

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = `$IterIncrement`*

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default = `$IterReset`*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection: `$iterator=$cname->begin()`

*Default = \$(CType)\_first(\$iterator, &pos)*

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = Empty string*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = !\$(CType)\_isDone(\$iterator, pos)*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<$target*>::const_iterator`,

You can change the iterator type to one of your own choice.

*Default = \$(CType) \**

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

Default for C++ = `for (int pos = 0; pos $multiplicity; ++pos)`

Default for C = `int pos; for (pos = 0; pos < $multiplicity; ++pos)`

Default for Java = `for (int pos = 0; pos $multiplicity; pos++)`

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$target\$reference*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(),$item);$cname->erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find($cname->begin(), $cname->end(),p); $cname->erase(pos)`

*Default = \$(CType)\_remove(&(\$me\$cname), \$item)*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items: `$cname->clear()`

*Default = \$(CType)\_removeAll(&(\$me\$cname))*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved: `$cname = $item`

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

## UnboundedUnordered

Defines properties for implementing relations whose multiplicity is unbounded (\*) and that should be accessed randomly.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter: `$cname->push_back($item)`

For maps (qualified relations), the following command inserts an item into the map based on a key: `$cname->insert(map<$keyType,$target*>::value_type( $keyName,$item))`

*Default = \$(CType)\_addHead(&(\$me\$cname), \$item)*

## Cast

The Cast property specifies the target.

*Default = (\$target\$reference)*

## CastRT

This property defines the return type casting in an iterator creation call. The user can empty it, as when generating MISRA compliant code.

*Default = (\$IterType)*

## Cleanup

The Cleanup property contains a pattern for the appropriate container destructor call.

*Default = \$(CType)\_Cleanup(&(\$me\$name))*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector: new vector<\$target\*>

*Default = \$me\$name = \$(CType)\_Create(\$multiplicity)*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$name: vector<\$target\*> \$name()

*Default = \$(CType) \$name*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default =*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector<\$target\*> collection type determines the type of the variable cl on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class Client is the \$target of vector operations.

*Default = RiCCollection*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$name to locate the \$item: \$name->find(\$item)

*Default = \$(CType)\_find(&(\$me\$name), \$item)*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = \$CType \$cname*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = \$me\$cname*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that call the at() operation for the container to retrieve the item at the indexed position: \$cname->at(\$index)

*Default = \$(CType)\_getAt(&(\$me\$cname), \$index)*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from



a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:  
`$cname-operator[]($keyName)`

*Default =*

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = weak*

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = <oxf/RiCCollection.h>*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: `$cname()`

*Default = \$(CType)\_Init(&(\$me\$cname), \$multiplicity);*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: `pos=0; pos<$multiplicity; pos++; $cname[pos]=NULL`

*Default = Empty string*

## InitSimple

The InitSimple property contains a pattern for the appropriate container initialization call.

*Default =*

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: `vector<$target*>::const_iterator $iterator; $iterator=$cname>begin()`

*Default = unsigned pos; \$IterType \$iterator = \$CastRT&(\$me\$cname)*

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = \$IterCreate*

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = \$(CType)\_get(\$iterator, pos)*

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item: `$iterator++`

*Default = \$(CType)\_next(\$iterator, &pos)*

## **IterIncrementForCleanUp**

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

*Default = \$IterIncrement*

## **IterIncrementForInit**

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = \$IterIncrement*

## **IterInit**

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

*Default = \$IterReset*

## **IterReset**

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection: \$iterator=\$cname->begin()

*Default = \$(CType)\_first(\$iterator, &pos)*

## **IterReturnType**

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = Empty string*

## **IterTest**

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname->end() With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

*Default = !\$(CType)\_isDone(\$iterator, pos)*

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<\$,target\*>::const\_iterator

You can change the iterator type to one of your own choice.

*Default = \$(CType) \**

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

Default for C++ = for (int pos = 0; pos \$multiplicity; ++pos)

Default for C = int pos; for (pos = 0; pos < \$multiplicity; ++pos)

Default for Java = for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = \$(constant)\$target\$reference*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to be removed and then call erase() to remove it. vector<\$,target\*>::iterator pos=find(\$cname>begin(), \$cname>end(),\$item);\$cname>erase(pos) This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair `p`, whose second element, `p.second`, is the item to be removed. The `find()` operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair<$keyType,$target*> p; p.second=$item; map<$keyType,$target*>::iterator pos=find($cname->begin(), $cname->end(),p); $cname->erase(pos)`

*Default = \$(CType)\_remove(&(\$me\$cname), \$item)*

## RemoveAll

The `RemoveAll` property specifies the command used to remove all items from the container. For example, the following command calls the `clear()` operation for the container to remove all items: `$cname->clear()`

*Default = \$(CType)\_removeAll(&(\$me\$cname))*

## RemoveKey

The `RemoveKey` property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the `erase()` operation for the container, passing it the `$keyName`, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

*Default =*

## Set

The `Set` property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, `$cname`, is a role name of the relation itself, because there is only one class involved: `$cname = $item`

*Default =*

## Type

The `Type` property specifies the type of the container as a pointer to the relation.

*Default =*

## User

Defines properties for user-defined implementations of relations.

You can create your own implementations for relations by defining a new set of properties under the `User`

metaclass. Once these are defined, you can give them permanent status by manually saving them in the factory.prp file under any other name, for example MyFaves. To complete their installation, you must add the new name as an enumerated value to the CG::Relation::Implementation property.

For example, you can change the definition of the Implementation property as follows:

```
Subject CG Metaclass Relation Property Implementation Enum "Default,Scalar,Fixed,BoundedOrdered,
BoundedUnordered,UnboundedOrdered, UnboundedUnordered,Qualified,MyFaves, User" "Default" end
end
```

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter: \$cname->push\_back(\$item)

For maps (qualified relations), the following command inserts an item into the map based on a key: \$cname->insert(map<\$keyType,\$target\*>::value\_type( \$keyName,\$item))

*Default = Empty string*

## Cast

The Cast property specifies the target.

*Default =*

## CastRT

This property defines the return type casting in an iterator creation call. The user can empty it, as when generating MISRA compliant code.

*Default = Empty string*

## Cleanup

The Cleanup property contains a pattern for the appropriate container destructor call.

*Default = Empty string*

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and

returns a pointer to the vector: `new vector<$target*>`

*Default = Empty string*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname: `vector<$target*> $cname()`

*Default = Empty string*

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

*Default =*

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the `vector<$target*>` collection type determines the type of the variable `cl` on the left side of the assignment:

```
vector<Client*>* cl = new vector<Client*>(2);
```

In this case, the class `Client` is the \$target of vector operations.

*Default = Empty string*

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the `Find()` operation for container \$cname to locate the \$item: `$cname->find($item)`

*Default = Empty string*

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about composite types.

*Default = Empty string*

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$name

The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

*Default = Empty string*

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that call the at() operation for the container to retrieve the item at the indexed position: \$cname->at(\$index)

*Default = Empty string*

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end()

This property and GetEndGenerate were created to adhere to the standard library convention for "finding" where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

*Default =*

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps: \$cname-operator[](\$keyName)

*Default =*



## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file.

*Default = weak*

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

*Default = Empty string*

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: \$cname()

*Default = Empty string*

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: pos=0; pos<\$multiplicity; pos++; \$cname[pos]=NULL

*Default = Empty string*

## InitSimple

The InitSimple property contains a pattern for the appropriate container initialization call.

*Default =*

## **InitStatic**

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

*Default =*

## **IterCreate**

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: `vector<$target*>::const_iterator $iterator; $iterator=$sname>begin()`

*Default = Empty string*

## **IterCreateByValue**

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

*Default = Empty string*

## **IterGetCurrent**

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator`

This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

*Default = Empty string*

## **IterIncrement**

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item: `$iterator++`

*Default = Empty string*

## **IterIncrementForCleanUp**

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

*Default = Empty string*

## **IterIncrementForInit**

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

*Default = Empty string*

## **IterInit**

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default = Empty string*

## **IterReset**

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection: `$iterator=$cname->begin()`

*Default = Empty string*

## **IterReturnType**

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*Default = Empty string*

## **IterTest**

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname->end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as `IterGetCurrent()`).

*Default = Empty string*

## **IterType**

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<`, as defined in the STL. `vector<$target*>::const_iterator`

You can change the iterator type to one of your own choice.

*Default = Empty string*

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

Default for C++ = for (int pos = 0; pos \$multiplicity; ++pos)

Default for C = int pos; for (pos = 0; pos < \$multiplicity; ++pos)

Default for Java = for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

*Default =*

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

*Default = Empty string*

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector<$target*>::iterator pos=find($cname>begin(), $cname>end(), $item); $cname>erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair<$keyType, $target*> p; p.second=$item; map<$keyType, $target*>::iterator pos=find($cname->begin(), $cname->end(), p); $cname->erase(pos)`

*Default = Empty string*

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:  
`$cname->clear()`

*Default = Empty string*

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed:  
`$cname-erase($keyName)`

*Default =*

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved: `$cname = $item`

*Default =*

## Type

The Type property specifies the type of the container as a pointer to the relation.

*Default =*

# RTInterface

---

The RTInterface subject contains metaclasses that contain properties that determine how Rational Rhapsody interacts with requirements traceability tools.

## DOORS

The DOORS metaclass contains properties that enable you to define the installation directory, Rational DOORS project name, location of Rational DOORS license file, and whether to run Rational DOORS in batch mode.

### CheckForLastModifyDiagrams

The CheckForLastModifyDiagrams property is a Boolean value that specifies whether to check when the diagrams were last modified.

*(Default = Checked)*

### ExportReadOnlyUnits

The ExportReadOnlyUnits property is a Boolean value that specifies whether to export read-only units to Rational DOORS.

*(Default = Checked)*

### InstallationDir

*The InstallationDir property specifies the path to the Rational DOORS installation. This property is optional, because Rational Rhapsody reads this information from the Windows registry. (Default = empty string)*

### LinkModuleName

The LinkModuleName property specifies the name of the set module for the links in Rational DOORS.

*(Default = Rhapsody\_links)*

### LmLicenseFile

The LmLicenseFile property specifies the location of your Rational DOORS license. This property is optional, because Rational Rhapsody reads this information from the Windows registry.

*(Default = empty string)*

## ModuleNameFromProject

The ModuleNameFromProject property is a Boolean value that specifies whether to get the name of the module from the Rational Rhapsody project.

*(Default = Cleared)*

## ProjectName

The ProjectName property specifies the name of the Rational DOORS project entered in the Rational DOORS Interface window. Exporting and checking of data are disabled until you enter a project name.

*(Default = empty string)*

## RunInBatchMode

The RunInBatchMode property specifies whether to run Rational DOORS in batch mode rather than interactive mode. This property corresponds to the option of the same name in the Rational DOORS Interface window. Note that you must use interactive mode if you want to navigate to Rational DOORS from the Rational Rhapsody browser.

*(Default = Cleared)*

## ExportOptions

The ExportOptions metaclass contains properties that determine which Rational Rhapsody items are exported to the requirement traceability tool (DOORS) and how they are mapped to Rational DOORS formal modules. Most of these options are set in the Export Options window of the Rational DOORS interface.

## ActivityDiagrams

The ActivityDiagrams property is a Boolean value that specifies whether to export activity diagrams to Rational DOORS.

*(Default = Checked)*

## ActivityStates

The ActivityStates property is a Boolean value that specifies whether to export activity states to Rational DOORS.

*(Default = Checked)*

## **ActivityTransitions**

The ActivityTransitions property is a Boolean value that specifies whether to export activity transitions to Rational DOORS.

*(Default = Checked)*

## **Actors**

The Actors property is a Boolean value that specifies whether to export actors to Rational DOORS.

*(Default = Checked)*

## **Associations**

The Associations property is a Boolean value that specifies whether to export associations to Rational DOORS.

*(Default = Checked)*

## **Attributes**

The Attributes property is a Boolean value that specifies whether to export attributes to Rational DOORS.

*(Default = Checked)*

## **Classes**

The Classes property is a Boolean value that specifies whether to export classes to Rational DOORS.

*(Default = Checked)*

## **CommunicationDiagrams**

The CommunicationDiagrams property is a Boolean value that specifies whether to export communication diagrams to Rational DOORS.

*(Default = Checked)*

## **Comments**



The Comments property is a Boolean value that specifies whether to export comments to Rational DOORS.

*(Default = Checked)*

## **ComponentDiagrams**

The ComponentDiagrams property is a Boolean value that specifies whether to export component diagrams to Rational DOORS.

*(Default = Checked)*

## **Components**

The Components property is a Boolean value that specifies whether to export components to Rational DOORS.

*(Default = Checked)*

## **Configurations**

The Configurations property is a Boolean value that specifies whether to export configurations to Rational DOORS.

*(Default = Checked)*

## **Constraints**

The Constraints property is a Boolean value that specifies whether to export constraints to Rational DOORS.

*(Default = Checked)*

## **ControlledFiles**

The ControlledFiles property indicates whether or not external files, such as project specifications files produced in Word or Excel, are accepted as sources for requirements in the Rational Rhapsody project.

*(Default = Checked)*

## **CreateModulePerPackage**

The CreateModulePerPackage property is a Boolean value that specifies whether to create a separate formal module in Rational DOORS to correspond to each package in the Rational Rhapsody project.

If this property is set to Checked, one Rational DOORS formal module is created for each package selected in the Rational Rhapsody browser tree in the Rational DOORS Interface window. Otherwise, a single formal module named RHAPSODY\_MODULE is created in Rational DOORS to which all design elements are exported.

*(Default = Checked)*

## **Dependencies**

The Dependencies property is a Boolean value that specifies whether to export dependencies to Rational DOORS.

*(Default = Checked)*

## **Events**

The Events property is a Boolean value that specifies whether to export events to Rational DOORS.

*(Default = Checked)*

## **ExportAllScope**

The ExportAllScope property is a Boolean value that specifies whether to export all elements to Rational DOORS. This property corresponds to the Export All check box in the Rational DOORS Interface window.

*(Default = Checked)*

## **ExportLabels**

The ExportLabels property is a Boolean value that specifies whether to export labels to Rational DOORS.

*(Default = Cleared)*

## **ExportPictures**

The ExportPictures property is a Boolean value that specifies whether to export pictures to Rational DOORS.

*(Default = Cleared)*

## **Files**

The Files property is a Boolean value that specifies whether to export files to Rational DOORS.

*(Default = Checked)*

## **Flows**

The Flows property is a Boolean value that specifies whether to export information flows to Rational DOORS.

*(Default = Checked)*

## **Folders**

The Folders property is a Boolean value that specifies whether to export folders to Rational DOORS.

*(Default = Checked)*

## **GlobalFunctions**

The GlobalFunctions property is a Boolean value that specifies whether to export global functions to Rational DOORS.

*(Default = Checked)*

## **GlobalInstances**

The GlobalInstances property is a Boolean value that specifies whether to export global instances to Rational DOORS.

*(Default = Checked)*

## **GlobalVariables**

The GlobalVariables property is a Boolean values that specifies whether to export global variables to Rational DOORS.

*(Default = Checked)*

## **HyperLinks**

The HyperLinks property is a Boolean value that specifies whether to export hyperlinks to Rational DOORS.

*(Default = Checked)*

## ItemFlows

The ItemFlows property is a Boolean value that specifies whether to export ItemFlows to Rational DOORS.

*Default = Checked*

## Links

The Links property is a Boolean value that specifies whether to export links to Rational DOORS.

*(Default = Checked)*

## Nodes

The Nodes property is a Boolean value that specifies whether to export nodes to Rational DOORS.

*(Default = Checked)*

## ObjectModelDiagrams

The ObjectModelDiagrams property is a Boolean value that specifies whether to export object model diagrams to Rational DOORS.

*(Default = Checked)*

## Operations

The Operations property is a Boolean value that specifies whether to export operations to Rational DOORS.

*(Default = Checked)*

## Packages

The Packages property is a Boolean value that specifies whether to export packages to Rational DOORS.

*(Default = Checked)*

## Ports

The Ports property is a Boolean value that specifies whether to export ports to Rational DOORS.

*(Default = Checked)*

## PurgeOnDelete

The PurgeOnDelete property is a Boolean value that specifies whether to use hard deletion in Rational DOORS. With hard delete, the element and its link is deleted from the Rational DOORS database. With soft delete, the element is marked as deleted, but remains in the database so it can be recovered; the link is deleted. Note the following:

- When there is an extra element in Rational DOORS that does not exist in Rational Rhapsody, the system displays a message asking whether you want to delete it.
- If you soft delete an element and later create an element with the same name, a new shadow element is created in Rational DOORS, and the old one is not used.
- If you switch from soft delete to hard delete, the soft-deleted elements remain in Rational DOORS.

*(Default = Checked)*

## Relations

The Relations property is a Boolean value that specifies whether to export relations to Rational DOORS.

*(Default = Checked)*

## Requirements

The Requirements property is a Boolean value that specifies whether to export requirements to Rational DOORS.

*(Default = Checked)*

## ScopeToExport

*The ScopeToExport property is a list of selected packages and diagrams that is exported to Rational DOORS if ExportAllScope is Cleared. (Default = empty string)*

## SequenceDiagrams

The SequenceDiagrams property is a Boolean value that specifies whether to export sequence diagrams to Rational DOORS.

*(Default = Checked)*

## StateCharts

The StateCharts property is a Boolean value that specifies whether to export statecharts to Rational

DOORS.

*(Default = Checked)*

## **States**

The States property is a Boolean value that specifies whether to export states to Rational DOORS.

*(Default = Checked)*

## **Stereotypes**

The Stereotypes property is a Boolean value that specifies whether to export stereotypes to Rational DOORS.

*(Default = Checked)*

## **StructureDiagrams**

The StructureDiagrams property is a Boolean value that specifies whether to export structure diagrams to Rational DOORS.

*(Default = Checked)*

## **Swimlanes**

The Swimlanes property is a Boolean value that specifies whether to export swimlanes to Rational DOORS.

*(Default = Checked)*

## **Tags**

The Tags property is a Boolean value that specifies whether to export tags to Rational DOORS.

*(Default = Checked)*

## **Transitions**

The Transitions property is a Boolean value that specifies whether to export transitions to Rational DOORS.

*(Default = Checked)*

## Types

The Types property is a Boolean value that specifies whether to export types to Rational DOORS.

*(Default = Checked)*

## UseCaseDiagrams

The UseCaseDiagrams property is a Boolean value that specifies whether to export use case diagrams to Rational DOORS.

*(Default = Checked)*

## UseCases

The UseCases property is a Boolean value that specifies whether to export use cases to Rational DOORS.

*(Default = Checked)*

# SequenceDiagram

---

The SequenceDiagram subject contains metaclasses that contain properties that determine the appearance and behavior of sequence diagrams.

## Animation

Contains properties related to animated sequence diagrams.

### DisplayTimeStamp

If you would like animated sequence diagrams to include timestamp information that indicates when specific messages were sent, set the value of the property DisplayTimeStamp to True.

*Default = False*

## CancelledTimeout

Contains properties relating to canceled timeouts.

### DefaultName

When you add a new model element, it is assigned a default name based on the type of element, for example, class\_0 or class\_1. The same type of generic approach is used when you create elements based on a new term. So if your model contains a "new term" stereotype called SystemConfiguration, when you add new elements of this type, they will have names such as systemconfiguration\_0 or systemconfiguration\_1.

For "new terms", you have an option to change this default naming scheme. You can provide a value for the property DefaultName (at the level of the relevant stereotype) to provide a different string that should be used as the basis for the default name when you create an element of that type. So, you can call your "new term" stereotype SystemConfiguration but then assign a value such as SC to the DefaultName property for that stereotype. Then, each new element of that type that you create will have a default name such as sc\_0 or sc\_1.

*Default = CanTm*

## Comment



The Comment metaclass contains properties relating to the display of Comment elements in sequence diagrams.

## CommentNotation

The CommentNotation property determines how Comment elements are displayed. This property can be set to one of two styles:

- Note\_Style
- Box\_Style

If CommentNotation is set to Note\_Style, then the appearance of the element is determined by the value selected for the property Comment::ShowForm: Note, Plain, or PushPin.

If CommentNotation is set to Box\_Style, then the element resembles a class-box, and you can control the display of compartments for the element.

Default = Note\_Style

## Compartments

The Compartments property determines which of the available compartments are displayed by default for the various types of elements. The value for this property is a comma-delimited string containing the names of the compartments that should be visible.

Since the available compartments vary from element to element, it is recommended that you do not try to set the value of this property by using the Properties window or directly in the .prp file. Rather, you should use an element's Display Options to set which compartments are visible, and then use the Make Default option to apply these settings at the diagram or project level for new elements of this type.

*Default = Blank*

## QuickNavigationCategories

Graphic elements in diagrams can include a list of shortcuts that are accessible from the title bar of the element. You can use the property QuickNavigationCategories to specify the categories of items that should be included in this shortcut list and the order in which the categories should appear in the list. The value of the property should be a comma-separated list of categories. The categories that can be included are:

- Activities
- DiagramReferences
- Diagrams
- Hyperlinks
- InternalBlockDiagrams
- MainBehavior
- MainDiagram

- References
- StateCharts

If you want greater control over the content of the navigation list (such as showing/hiding specific elements within a given category, or changing the order of elements within a specific category), use the Compartments list, which can be found under "Display Options".

*Default = Hyperlinks*

## ShowAnnotationContents

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the text displayed on the element in the diagram determined by the value that was set for the property ShowAnnotationContents. The property can take one of the following values:

- Name - the name of the element
- Label - the label assigned to the element
- Description - the content of the Description field
- Specification (for Requirements only) - the content of the Specification field

*Default = Description*

## ShowForm

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the appearance of the element is determined by the value that was set for the property ShowForm.

The possible values for this property vary for the different elements, as does the default value used. The values used are:

- Plain - No color background behind text
- Note - Color background behind text
- Pushpin - Color background plus pin icon

*Default = Note*

## ShowName

The ShowName property determines the text to display next to a graphic element in a diagram. For most elements, you can provide a name and a label. Your descriptive label can be useful in cases where the name itself might not be sufficient due to various constraints, such as the inability to use spaces if the name of the element is to appear in the code.

The possible values for this property vary for the different elements, as does the default value used. The possible values are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example,

package\_1::package\_1b::class\_0

- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is displayed with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

Note that for some elements, such as destruction elements in sequence diagrams and timing diagrams, the property is a boolean property that hides/shows the name of the element.

*Default = Name\_only*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## Condition\_Mark

The Condition\_Mark metaclass contains properties that can be used to show a state of an instance or a condition the instance chooses.

## AlignConditionMarksLeft

The AlignConditionMarksLeft property is a Boolean value that controls the left alignment of condition mark text. If set to Checked, all newly created condition marks have text aligned left.

- Condition marks from projects created (last saved) before Rational Rhapsody 6.0 is aligned left.
- Alignment of existing text cannot be changed by using the diagram editor

*(Default = Cleared)*

# Constraint

The Constraint metaclass contains properties relating to the display of Constraint elements in sequence diagrams.

## Compartments

The Compartments property determines which of the available compartments are displayed by default for the various types of elements. The value for this property is a comma-delimited string containing the names of the compartments that should be visible.

Since the available compartments vary from element to element, it is recommended that you do not try to set the value of this property by using the Properties window or directly in the .prp file. Rather, you should use an element's Display Options to set which compartments are visible, and then use the Make Default option to apply these settings at the diagram or project level for new elements of this type.

*Default = Specification*

## ConstraintNotation

The ConstraintNotation property determines how Constraint elements are displayed. This property can be set to one of two styles:

- Note\_Style
- Box\_Style

If ConstraintNotation is set to Note\_Style, then the appearance of the element is determined by the value selected for the property Constraint::ShowForm: Note, Plain, or PushPin.

If ConstraintNotation is set to Box\_Style, then the element resembles a class-box, and you can control the display of compartments for the element.

*Default = Note\_Style*

## QuickNavigationCategories

Graphic elements in diagrams can include a list of shortcuts that are accessible from the title bar of the element. You can use the property QuickNavigationCategories to specify the categories of items that should be included in this shortcut list and the order in which the categories should appear in the list. The value of the property should be a comma-separated list of categories. The categories that can be included are:

- Activities
- DiagramReferences
- Diagrams

- Hyperlinks
- InternalBlockDiagrams
- MainBehavior
- MainDiagram
- References
- StateCharts

If you want greater control over the content of the navigation list (such as showing/hiding specific elements within a given category, or changing the order of elements within a specific category), use the Compartments list, which can be found under "Display Options".

*Default = Hyperlinks*

## ShowAnnotationContents

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the text displayed on the element in the diagram determined by the value that was set for the property ShowAnnotationContents. The property can take one of the following values:

- Name - the name of the element
- Label - the label assigned to the element
- Description - the content of the Description field
- Specification (for Requirements only) - the content of the Specification field

*Default = Description*

## ShowForm

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the appearance of the element is determined by the value that was set for the property ShowForm.

The possible values for this property vary for the different elements, as does the default value used. The values used are:

- Plain - No color background behind text
- Note - Color background behind text
- Pushpin - Color background plus pin icon

*Default = Note*

## ShowName

The ShowName property determines the text to display next to a graphic element in a diagram. For most elements, you can provide a name and a label. Your descriptive label can be useful in cases where the name itself might not be sufficient due to various constraints, such as the inability to use spaces if the name of the element is to appear in the code.

The possible values for this property vary for the different elements, as does the default value used. The possible values are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is displayed with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

Note that for some elements, such as destruction elements in sequence diagrams and timing diagrams, the property is a boolean property that hides/shows the name of the element.

*Default = Name\_only*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## CreateMessage

Contains properties relating to CreateMessage elements.

## DefaultName

When you add a new model element, it is assigned a default name based on the type of element, for example, class\_0 or class\_1. The same type of generic approach is used when you create elements based on a new term. So if your model contains a "new term" stereotype called SystemConfiguration, when you add new elements of this type, they will have names such as systemconfiguration\_0 or systemconfiguration\_1.

For "new terms", you have an option to change this default naming scheme. You can provide a value for the property `DefaultName` (at the level of the relevant stereotype) to provide a different string that should be used as the basis for the default name when you create an element of that type. So, you can call your "new term" stereotype `SystemConfiguration` but then assign a value such as `SC` to the `DefaultName` property for that stereotype. Then, each new element of that type that you create will have a default name such as `sc_0` or `sc_1`.

*Default = createmessage\_\$Index*

## DataFlow

Contains properties relating to DataFlow elements.

### DefaultName

When you add a new model element, it is assigned a default name based on the type of element, for example, `class_0` or `class_1`. The same type of generic approach is used when you create elements based on a new term. So if your model contains a "new term" stereotype called `SystemConfiguration`, when you add new elements of this type, they will have names such as `systemconfiguration_0` or `systemconfiguration_1`.

For "new terms", you have an option to change this default naming scheme. You can provide a value for the property `DefaultName` (at the level of the relevant stereotype) to provide a different string that should be used as the basis for the default name when you create an element of that type. So, you can call your "new term" stereotype `SystemConfiguration` but then assign a value such as `SC` to the `DefaultName` property for that stereotype. Then, each new element of that type that you create will have a default name such as `sc_0` or `sc_1`.

*Default = dataflow\_\$Index*

## Depends

The Depends metaclass contains properties relating to the display of Dependency lines in sequence diagrams.

### line\_style

The `line_style` property specifies the default line style for a graphical item.

The possible values are:

- `straight_arrows` - straight line.
- `rectilinear_arrows` - rectilinear line with right-angled corners placed at appropriate locations, depending

on the starting and ending points of the line

- `spline_arrows` - curved line without corners
- `rounded_rectilinear_arrows` - similar to ordinary rectilinear lines but use rounded bends rather than straight ninety-degree angles

*Default = `spline_arrows`*

*Default = `straight_arrows`*

## ShowName

The ShowName property determines the text to display next to a graphic element in a diagram. For most elements, you can provide a name and a label. Your descriptive label can be useful in cases where the name itself might not be sufficient due to various constraints, such as the inability to use spaces if the name of the element is to appear in the code.

The possible values for this property vary for the different elements, as does the default value used. The possible values are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, `package_1::package_1b::class_0`
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is displayed with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

Note that for some elements, such as destruction elements in sequence diagrams and timing diagrams, the property is a boolean property that hides/shows the name of the element.

*Default = `None`*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = `Label`*



# DestroyMessage

Contains properties relating to DestroyMessage elements.

## DefaultName

When you add a new model element, it is assigned a default name based on the type of element, for example, class\_0 or class\_1. The same type of generic approach is used when you create elements based on a new term. So if your model contains a "new term" stereotype called SystemConfiguration, when you add new elements of this type, they will have names such as systemconfiguration\_0 or systemconfiguration\_1.

For "new terms", you have an option to change this default naming scheme. You can provide a value for the property DefaultName (at the level of the relevant stereotype) to provide a different string that should be used as the basis for the default name when you create an element of that type. So, you can call your "new term" stereotype SystemConfiguration but then assign a value such as SC to the DefaultName property for that stereotype. Then, each new element of that type that you create will have a default name such as sc\_0 or sc\_1.

*Default = destroymessage\_\$Index*

# DestructionEvent

Contains properties relating to destruction event elements in the diagram.

## ShowName

The ShowName property determines the text to display next to a graphic element in a diagram. For most elements, you can provide a name and a label. Your descriptive label can be useful in cases where the name itself might not be sufficient due to various constraints, such as the inability to use spaces if the name of the element is to appear in the code.

The possible values for this property vary for the different elements, as does the default value used. The possible values are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed

- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is displayed with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

Note that for some elements, such as destruction elements in sequence diagrams and timing diagrams, the property is a boolean property that hides/shows the name of the element.

*Default = True*

## DiagramFrame

The DiagramFrame metaclass contains properties that control the appearance of a diagram frame.

### DefaultSize

The DefaultSize property specifies default values for diagram frame dimensions.

*Default = 20, 20, 590, 500*

### ShowName

The ShowName property determines the text to display next to a graphic element in a diagram. For most elements, you can provide a name and a label. Your descriptive label can be useful in cases where the name itself might not be sufficient due to various constraints, such as the inability to use spaces if the name of the element is to appear in the code.

The possible values for this property vary for the different elements, as does the default value used. The possible values are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is displayed with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

Note that for some elements, such as destruction elements in sequence diagrams and timing diagrams, the

property is a boolean property that hides/shows the name of the element.

*Default = Name*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## EventMessage

Contains properties relating to EventMessage elements.

### DefaultName

When you add a new model element, it is assigned a default name based on the type of element, for example, class\_0 or class\_1. The same type of generic approach is used when you create elements based on a new term. So if your model contains a "new term" stereotype called SystemConfiguration, when you add new elements of this type, they will have names such as systemconfiguration\_0 or systemconfiguration\_1.

For "new terms", you have an option to change this default naming scheme. You can provide a value for the property DefaultName (at the level of the relevant stereotype) to provide a different string that should be used as the basis for the default name when you create an element of that type. So, you can call your "new term" stereotype SystemConfiguration but then assign a value such as SC to the DefaultName property for that stereotype. Then, each new element of that type that you create will have a default name such as sc\_0 or sc\_1.

*Default = eventmessage\_ \$Index*

## FoundMessage

Contains properties relating to FoundMessage elements.

### DefaultName

When you add a new model element, it is assigned a default name based on the type of element, for example, class\_0 or class\_1. The same type of generic approach is used when you create elements based on a new term. So if your model contains a "new term" stereotype called SystemConfiguration, when you add new elements of this type, they will have names such as systemconfiguration\_0 or systemconfiguration\_1.

For "new terms", you have an option to change this default naming scheme. You can provide a value for the property DefaultName (at the level of the relevant stereotype) to provide a different string that should be used as the basis for the default name when you create an element of that type. So, you can call your "new term" stereotype SystemConfiguration but then assign a value such as SC to the DefaultName property for that stereotype. Then, each new element of that type that you create will have a default name such as sc\_0 or sc\_1.

*Default = foundmessage\_\$Index*

## General

The General metaclass contains properties that control the appearance and behavior of elements in sequence diagrams.

### AutoCreateExecutionOccurrence

The AutoCreateExecutionOccurrence property determines whether execution occurrences are created automatically when a message is created. See the Rational Rhapsody help for more information on execution occurrences.

*(Default = Cleared)*

### AutoLaunchAnimation

The AutoLaunchAnimation property can be set for a sequence diagram so that Rational Rhapsody automatically opens the diagram when the application is run in animation mode.

The following values are available:

- Never - The diagram is never opened automatically.
- Always - The diagram is always opened automatically.
- If\_Open - The diagram is opened automatically only if the sequence diagram is already open.

*Default = If\_Open*

### ClassCentricMode

*The ClassCentricMode property specifies whether you can create sequence diagrams with instances and messages that are not realized by model elements. When this property is set to True, you can create a*

*class by typing Class Name>, which in turn changes the label on the instance line to :Class Name>. (Default = Cleared if its in Analysis mode, Checked if its in Design mode)*

## CleanupRealized

*The CleanupRealized property specifies whether to delete the realized messages and classifier roles from the sequence diagram when you delete classifiers, operations, or events. (Default = Cleared if its in Analysis mode, Checked if its in Design mode)*

## ConfirmCreation

The ConfirmCreation property specifies whether Rational Rhapsody should confirm the creation of the corresponding operation. When you change the name of a message and this property is set to True, a message displays asking whether you want to create the operation.

When this property is set to Cleared, there is no confirmation window - the operation is created automatically. This property is relevant when the RealizeMessages property is set to Checked (usually for design mode of sequence diagrams).

*(Default = Checked)*

## DefaultLifelineType

*The DefaultLifelineType property determines whether new instance lines are of type class or file. (Default = Class)*

## HorizontalMessageType

The HorizontalMessageType property determines the default type of new horizontal messages: There are 4 possible values:

- Default - Same as PrimitiveOperation
- PrimitiveOperation - An operation whose body you write yourself. Rational Rhapsody automatically generates bodies for all other types of operations.
- TriggeredOperation - A cross between an operation and an event. It is started by another object to trigger a state transition and its body is executed in response to the transition taken. Because it is a synchronous event, the sending object waits for the execution of the triggered operation.
- Event - An instantaneous occurrence that can trigger a state transition in a class.

The default is Default.

## MaxNumberOfAnimMessages

The MaxNumberOfAnimMessages property specifies the maximum number of animation messages to display at any time. The OnReachedMaxAnimMessages property determines how Rational Rhapsody behaves once this number has been reached.

*(Default = 1000)*

## MinimumMessageSpacing

When you draw messages in a sequence diagram, Rhapsody maintains a certain minimum distance between the messages. As a result, when you add a new message, you may see existing messages move in order to maintain this distance. You can change the minimum distance between messages by modifying the value of the property MinimumMessageSpacing. The value of the property represents the number of pixels between messages.

*Default = 20*

## OnReachedMaxAnimMessages

The OnReachedMaxAnimMessages property determines how Rational Rhapsody should behave when the maximum number of messages has been reached. The property can take the following values:

- Stop - Rational Rhapsody stops displaying animated messages in the diagram after the maximum number has been reached.
- KeepLast - After the maximum number of messages specified has been reached, Rational Rhapsody erases the first messages displayed. The product continues erasing displayed messages in this manner so that the number of messages displayed on the diagram at any one time does not exceed the maximum specified..

*(Default = KeepLast)*

## RealizeMessages

*The RealizeMessages property specifies whether to realize messages in sequence diagrams (use constructive mode). (Default = Checked)*

## SelfMessageType

The HorizontalMessageType property determines the default type of new “self” messages (messages sent from an item to itself): There are 4 possible values:

- Default - Same as PrimitiveOperation
- PrimitiveOperation - An operation whose body you write yourself. Rational Rhapsody automatically generates bodies for all other types of operations.
- TriggeredOperation - A cross between an operation and an event. It is started by another object to trigger a state transition and its body is executed in response to the transition taken. Because it is a synchronous event, the sending object waits for the execution of the triggered operation.
- Event - An instantaneous occurrence that can trigger a state transition in a class.

*(Default = Default)*

## ShowAnimCreateArrow

The ShowAnimCreateArrow property determines whether or not create arrows are displayed in an animated sequence diagram.

*Default = Checked*

## ShowAnimDestroyArrow

The ShowAnimDestroyArrow property determines whether or not destroy arrows are displayed in an animated sequence diagram.

*Default = Checked*

## ShowAnimStateMark

The ShowAnimStateMark property is used to control the display of states on animated sequence diagrams.

By default, during animation, states entered are displayed as condition marks on instance lines.

If you prefer not to have states displayed on your animated sequence diagrams, set the value of this property to False.

*Default = Checked*

## ShowAnimTimeoutArrow

The ShowAnimTimeoutArrow property determines whether or not timeout arrows are displayed in an animated sequence diagram.

*Default = Checked*

## ShowAnimCancelTimeoutArrow

The ShowAnimCancelTimeoutArrow property determines whether or not canceled timeout arrows are displayed in an animated sequence diagram.

*Default = Checked*

## ShowAnimDataFlowArrow

The ShowAnimDataFlowArrow property determines whether or not data flow arrows are displayed in an animated sequence diagram to indicate the flow of data between flow ports.

*Default = Checked*

## ShowArguments

The ShowArguments property specifies whether message arguments should be displayed in sequence diagrams, and how they should be displayed. The possible values are:

- None - Message arguments should not be displayed.
- Names - Message arguments should be displayed, but without their types.
- NamesAndTypes - Message arguments should be displayed together with their types.

*Default = Names*

## ShowDynamicAnimInstanceName

By default, on animated sequence diagrams, Rational Rhapsody does not update the caption for an instance line if the name of the instance is changed dynamically by the application.

If you would like Rational Rhapsody to update instance names on the diagram if they are changed, set the value of the ShowDynamicAnimInstanceName property to True.

Note that if your diagram includes the special notation that allows auto-creation of animated instances, the value of this property has no effect. Instance names are always updated.

*Default = Cleared*

## ShowSequenceNumbers

*The ShowSequenceNumbers property specifies whether to sequence numbers in sequence diagrams.  
(Default = Cleared)*

## SlantMessageType

The SlantMessageType property determines the default type of new "slanted" messages (messages sent from one item to another). The possible values are:

- Default - Same as PrimitiveOperation.
- PrimitiveOperation - An operation whose body you write yourself. Rational Rhapsody automatically generates bodies for all other types of operations.
- TriggeredOperation - A cross between an operation and an event. It is started by another object to trigger a state transition and its body is executed in response to the transition taken. Because it is a synchronous event, the sending object waits for the execution of the triggered operation.
- Event - An instantaneous occurrence that can trigger a state transition in a class.

*Default = Default (same as PrimitiveOperation)*

## SysMLDiagramKindTitle



When a diagram is created in SysML projects, the diagram frame contains a header for the diagram that includes information such as the name of the package that contains the diagram. The property `SysMLDiagramKindTitle` is used to specify a prefix for the diagram frame header that reflects the type of diagram. For example, when you create an internal block diagram, the header in the diagram frame starts with the prefix "ibd".

Note that the structure of the entire header is controlled by the property `General::Graphics::DiagramHeader`.

*Default = sd*

## InstanceLine

Contains properties that affect the display of new instance lines in sequence diagrams.

### BottomMargin

The `BottomMargin` property defines the margin size between the lowest element in the sequence diagram and the end of the instance line.

### ShowStereotype

The `ShowStereotype` property determines whether or not classifier role stereotypes is opened when you add new instance lines to the diagram.

*Default = False*

## InteractionOperator

The `InteractionOperator` metaclass contains a property that controls the appearance of interaction operator guards.

### ShowOperandsGuards

The `ShowOperandsGuards` property controls the appearance of guards. When you draw an `InteractionOperator`, guard might be set: it displays at the top of the `InteractionOperator` under the name [condition].

Setting this property to `Checked` displays guards. Setting this property to `Cleared` hides any guards.

*(Default = Checked)*

## LostMessage

Contains properties relating to LostMessage elements.

### DefaultName

When you add a new model element, it is assigned a default name based on the type of element, for example, class\_0 or class\_1. The same type of generic approach is used when you create elements based on a new term. So if your model contains a "new term" stereotype called SystemConfiguration, when you add new elements of this type, they will have names such as systemconfiguration\_0 or systemconfiguration\_1.

For "new terms", you have an option to change this default naming scheme. You can provide a value for the property DefaultName (at the level of the relevant stereotype) to provide a different string that should be used as the basis for the default name when you create an element of that type. So, you can call your "new term" stereotype SystemConfiguration but then assign a value such as SC to the DefaultName property for that stereotype. Then, each new element of that type that you create will have a default name such as sc\_0 or sc\_1.

*Default = lostmessage\_ \$Index*

## Message

Contains properties that affect the display of new messages in sequence diagrams.

### DefaultName

When you add a new model element, it is assigned a default name based on the type of element, for example, class\_0 or class\_1. The same type of generic approach is used when you create elements based on a new term. So if your model contains a "new term" stereotype called SystemConfiguration, when you add new elements of this type, they will have names such as systemconfiguration\_0 or systemconfiguration\_1.

For "new terms", you have an option to change this default naming scheme. You can provide a value for the property DefaultName (at the level of the relevant stereotype) to provide a different string that should be used as the basis for the default name when you create an element of that type. So, you can call your "new term" stereotype SystemConfiguration but then assign a value such as SC to the DefaultName property for that stereotype. Then, each new element of that type that you create will have a default name such as sc\_0 or sc\_1.

*Default = message\_ \$Index*

## ShowStereotype

The ShowStereotype property determines whether or not message stereotypes is opened when you add new messages to the diagram.

*Default = Cleared*

## ShowStereotypeOfOperation

The ShowStereotypeOfOperation property is a Boolean property which, when enabled, specifies that a message displays its realized operation stereotype.

*Default = Cleared*

## SupportFreeText

Prior to release 8.1.4, it was possible to add free text to messages in sequence diagrams, following the name of the operation, for example "printReport() - this prints the report". Beginning in 8.1.4, you can no longer include text that is not part of the operation name or operation arguments. In order to maintain the pre-8.1.4 behavior for older models, the property SupportFreeText was added to the backward compatibility settings for 8.1.4 with a value of True.

## Note

Contains properties relating to the display of Note elements.

## ShowForm

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the appearance of the element is determined by the value that was set for the property ShowForm.

The possible values for this property vary for the different elements, as does the default value used. The values used are:

- Plain - No color background behind text
- Note - Color background behind text
- Pushpin - Color background plus pin icon

*Default = Note*

# ReplyMessage

Contains properties relating to ReplyMessage Elements.

## DefaultName

When you add a new model element, it is assigned a default name based on the type of element, for example, class\_0 or class\_1. The same type of generic approach is used when you create elements based on a new term. So if your model contains a "new term" stereotype called SystemConfiguration, when you add new elements of this type, they will have names such as systemconfiguration\_0 or systemconfiguration\_1.

For "new terms", you have an option to change this default naming scheme. You can provide a value for the property DefaultName (at the level of the relevant stereotype) to provide a different string that should be used as the basis for the default name when you create an element of that type. So, you can call your "new term" stereotype SystemConfiguration but then assign a value such as SC to the DefaultName property for that stereotype. Then, each new element of that type that you create will have a default name such as sc\_0 or sc\_1.

*Default = replymessage\_\$Index*

# Requirement

The Requirement metaclass contains properties relating to the display of Requirement elements in sequence diagrams.

## Compartments

The Compartments property determines which of the available compartments are displayed by default for the various types of elements. The value for this property is a comma-delimited string containing the names of the compartments that should be visible.

Since the available compartments vary from element to element, it is recommended that you do not try to set the value of this property by using the Properties window or directly in the .prp file. Rather, you should use an element's Display Options to set which compartments are visible, and then use the Make Default option to apply these settings at the diagram or project level for new elements of this type.

*Default = ID, Specification*

## RequirementNotation

The RequirementNotation property determines how Requirement elements are displayed. This property

can be set to one of two styles:

- Note\_Style
- Box\_Style

If RequirementNotation is set to Note\_Style, then the appearance of the element is determined by the value selected for the property Requirement::ShowForm: Note, Plain, or PushPin.

If RequirementNotation is set to Box\_Style, then the element resembles a class-box, and you can control the display of compartments for the element.

*Default = Note\_Style*

## QuickNavigationCategories

Graphic elements in diagrams can include a list of shortcuts that are accessible from the title bar of the element. You can use the property QuickNavigationCategories to specify the categories of items that should be included in this shortcut list and the order in which the categories should appear in the list. The value of the property should be a comma-separated list of categories. The categories that can be included are:

- Activities
- DiagramReferences
- Diagrams
- Hyperlinks
- InternalBlockDiagrams
- MainBehavior
- MainDiagram
- References
- StateCharts

If you want greater control over the content of the navigation list (such as showing/hiding specific elements within a given category, or changing the order of elements within a specific category), use the Compartments list, which can be found under "Display Options".

*Default = Hyperlinks*

## ShowAnnotationContents

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the text displayed on the element in the diagram determined by the value that was set for the property ShowAnnotationContents. The property can take one of the following values:

- Name - the name of the element
- Label - the label assigned to the element
- Description - the content of the Description field
- Specification (for Requirements only) - the content of the Specification field

*Default = Description*

## ShowForm

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the appearance of the element is determined by the value that was set for the property ShowForm.

The possible values for this property vary for the different elements, as does the default value used. The values used are:

- Plain - No color background behind text
- Note - Color background behind text
- Pushpin - Color background plus pin icon

*Default = Note*

## ShowName

The ShowName property determines the text to display next to a graphic element in a diagram. For most elements, you can provide a name and a label. Your descriptive label can be useful in cases where the name itself might not be sufficient due to various constraints, such as the inability to use spaces if the name of the element is to appear in the code.

The possible values for this property vary for the different elements, as does the default value used. The possible values are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is displayed with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

Note that for some elements, such as destruction elements in sequence diagrams and timing diagrams, the property is a boolean property that hides/shows the name of the element.

*Default = Name\_only*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## SpecificationAnnotationFormat

When a requirement is displayed on a diagram, you can choose to have the element show one of the following: its name, its label, its description, the specification text for the requirement.

If you choose to display the specification text, you can use the property SpecificationAnnotationFormat to customize the text that is displayed.

The following keywords can be included in the value of the property: \$Specification (to include the specification text) and \$ID (to include the ID that was specified in the Features dialog for the requirement).

Note that this property affects the displayed text only when a requirement uses the "Note" notation style. When using the "Box" notation style, only the specification text is displayed, regardless of the value of this property.

*Default = \$Specification*

## TimeInterval

Contains properties relating to TimeInterval elements.

### DefaultName

When you add a new model element, it is assigned a default name based on the type of element, for example, class\_0 or class\_1. The same type of generic approach is used when you create elements based on a new term. So if your model contains a "new term" stereotype called SystemConfiguration, when you add new elements of this type, they will have names such as systemconfiguration\_0 or systemconfiguration\_1.

For "new terms", you have an option to change this default naming scheme. You can provide a value for the property DefaultName (at the level of the relevant stereotype) to provide a different string that should be used as the basis for the default name when you create an element of that type. So, you can call your "new term" stereotype SystemConfiguration but then assign a value such as SC to the DefaultName property for that stereotype. Then, each new element of that type that you create will have a default name such as sc\_0 or sc\_1.

*Default = <(\$Index) sec>*

## SequenceDiagram

The SequenceDiagram metaclass contains a property that controls the fill color of graphic elements in sequence diagrams.

### Fillcolor

The Fillcolor property specifies the default fill color for the object.

*Default = 218,218,218*

## Timeout

Contains properties relating to timeouts.

### DefaultName

When you add a new model element, it is assigned a default name based on the type of element, for example, class\_0 or class\_1. The same type of generic approach is used when you create elements based on a new term. So if your model contains a "new term" stereotype called SystemConfiguration, when you add new elements of this type, they will have names such as systemconfiguration\_0 or systemconfiguration\_1.

For "new terms", you have an option to change this default naming scheme. You can provide a value for the property DefaultName (at the level of the relevant stereotype) to provide a different string that should be used as the basis for the default name when you create an element of that type. So, you can call your "new term" stereotype SystemConfiguration but then assign a value such as SC to the DefaultName property for that stereotype. Then, each new element of that type that you create will have a default name such as sc\_0 or sc\_1.

*Default = CanTm*



# SPARK

---

The SPARK subject contains metaclasses that contain properties that enable you to control the generation of SPARK annotations from Rational Rhapsody Developer for Ada models so they can be analyzed by the SPARK Examiner. See the Rational Rhapsody Developer for Ada documentation for information.

## Class

The Class metaclass contains properties that control the examination level for the class.

### ExaminerLevelBody

The ExaminerLevelBody property specifies the examination level for the class. The possible values are as follows:

- None - Do not examine the file.
- Data - Perform data-flow analysis on the file.
- Information - Perform information-flow analysis on the file.

*Default = Data*

### ExaminerLevelSpec

The ExaminerLevelSpec property specifies the examination level for the class specification. The possible values are as follows:

- None - Do not examine the file.
- Data - Perform data-flow analysis on the file.
- Information - Perform information-flow analysis on the file.

*Default = Data*

## Package

The Package metaclass contains properties that control the examination level for the package.

### ExaminerLevelBody

The ExaminerLevelBody property specifies the examination level for the class. The possible values are as follows:

- None - Do not examine the file.
- Data - Perform data-flow analysis on the file.
- Information - Perform information-flow analysis on the file.

*Default = Data*

## **ExaminerLevelSpec**

The ExaminerLevelSpec property specifies the examination level for the class specification. The possible values are as follows:

- None—Do not examine the file.
- Data - Perform data-flow analysis on the file.
- Information - Perform information-flow analysis on the file.

*Default = Data*

# StatechartDiagram

---

The StatechartDiagram contains metaclasses that contain properties are used to control the appearance of elements in statechart diagrams.

## AcceptEventAction

The AcceptEventAction metaclass contains properties that affect the appearance of accept event actions in statechart diagrams.

### ShowNotation

For Accept Event Action elements in statecharts, the property ShowNotation determines what text will be displayed inside the element - the name of the element, the label assigned to the element, or the event associated with the element.

Note that when you change the value of the property, the new behavior applies only to new Accept Event Action elements that you add to a diagram. For existing elements, the text displayed is not changed. To change the display for existing elements, you can open the Display Options dialog for the element and select one of the options.

*Default = Event*

### ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = None*

## AcceptTimeEvent

Contains properties relating to the appearance of AcceptTimeEvent elements.

### ShowName

The ShowName property determines the text to display next to a graphic element in a diagram. For most

elements, you can provide a name and a label. Your descriptive label can be useful in cases where the name itself might not be sufficient due to various constraints, such as the inability to use spaces if the name of the element is to appear in the code.

The possible values for this property vary for the different elements, as does the default value used. The possible values are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is displayed with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

Note that for some elements, such as destruction elements in sequence diagrams and timing diagrams, the property is a boolean property that hides/shows the name of the element.

*Default = Duration*

## AutoPopulate

The AutoPopulate metaclass contains properties that can be used to control the appearance of diagrams that are drawn automatically by Rational Rhapsody.

## ArrowDirection

The ArrowDirection property is used when Rational Rhapsody automatically generates a diagram, and it determines whether the flow of connectors in the diagram runs from top to bottom or bottom to top.

There are two situations where Rational Rhapsody automatically generates diagrams:

- If you have selected the Populate Diagrams option for Reverse Engineering (for those diagrams where this feature is supported).
- If you double-click a diagram in the browser that was generated by using the Rational Rhapsody API.

*Default = Bottom-Top*

# ButtonArray

The ButtonArray metaclass contains properties that determine the appearance and behavior of button array controls on statecharts.

## ButtonFont

The ButtonFont property lets you select the font to use for the text on the face of a push button control.

To change the value of the property, click the "..." button in the box next to the property value to open the Font window. The value of the property affects both buttons already on the statechart and new buttons added to the statechart. (The display of buttons already on the statechart changes only after you refresh the diagram.)

*Default = Arial 10 NoBold NoItalic*

## Direction

The Direction property determines whether the button array controls are used to input data, display data, or both. The possible values are:

- In - The button arrays are only used to input data for the attribute to which it is bound.
- Out - The button arrays are only used to display data for the attribute to which it is bound.
- InOut - The button arrays are used to input data and display data.

*Default = InOut*

## ShowName

The ShowName property determines whether or not a caption is displayed for button array elements, and if so, what text should be displayed. The possible values are:

- BindedElementFullPath - The full path of the attribute that is bound to the button array.
- BindedElement - The name of the attribute that is bound to the button array.
- Name - The name of the button array element.
- None - No text is displayed.

*Default = Name*

# Comment

The Comment metaclass contains a property that controls the appearance of comments in statecharts.

## CommentNotation

The CommentNotation property determines how Comment elements are displayed. This property can be set to one of two styles:

- Note\_Style
- Box\_Style

If CommentNotation is set to Note\_Style, then the appearance of the element is determined by the value selected for the property Comment::ShowForm: Note, Plain, or PushPin.

If CommentNotation is set to Box\_Style, then the element resembles a class-box, and you can control the display of compartments for the element.

*Default = Note\_Style*

## Compartments

The Compartments property determines which of the available compartments are displayed by default for the various types of elements.

The value for this property is a comma-delimited string containing the names of the compartments that should be visible.

Since the available compartments vary from element to element, it is recommended you not set the value of this property by using the Properties window or directly in the .prp file. Use the Display Options of the element to set which compartments are visible, and then use the Make Default option to apply these settings at the diagram or project level for new elements of this type.

*Default = Empty MultiLine*

## QuickNavigationCategories

Graphic elements in diagrams can include a list of shortcuts that are accessible from the title bar of the element. You can use the property QuickNavigationCategories to specify the categories of items that should be included in this shortcut list and the order in which the categories should appear in the list. The value of the property should be a comma-separated list of categories. The categories that can be included are:

- Activities
- DiagramReferences
- Diagrams
- Hyperlinks
- InternalBlockDiagrams
- MainBehavior

- MainDiagram
- References
- StateCharts

If you want greater control over the content of the navigation list (such as showing/hiding specific elements within a given category, or changing the order of elements within a specific category), use the Compartments list, which can be found under "Display Options".

*Default = Hyperlinks*

## ShowAnnotationContents

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the text displayed on the element in the diagram determined by the value that was set for the property ShowAnnotationContents. The property can take one of the following values:

- Name - the name of the element
- Label - the label assigned to the element
- Description - the content of the Description field
- Specification (for Requirements only) - the content of the Specification field

*Default = Description*

## ShowForm

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the appearance of the element is determined by the value that was set for the property ShowForm.

The possible values for this property vary for the different elements, as does the default value used. The values used are:

- Plain - No color background behind text
- Note - Color background behind text
- Pushpin - Color background plus pin icon

*Default = Note*

## ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram.

For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code.

The possible values are:

- Description - The content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is opened with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

*Default = Relative*

## ShowStereotype

The ShowStereotype property determines if, and how, a stereotype of the element is opened in a diagram. The possible values are:

- Label - The stereotype of the element is opened as a text label.
- Bitmap - The bitmap image that is associated with the stereotype of the element is opened.
- None - The stereotype of the element is not displayed.

*Default = Label*

## Complete

The metaclass Complete contains properties that determine whether or not Rational Rhapsody automatically draws the relations that exist between an element added to a diagram and elements already on the diagram.

### Complete\_Relation

The Complete\_Relation property is used to specify that when an element is added to a diagram, Rational Rhapsody should automatically draw the relations that exist between the element and elements already on the diagram.

*Default = Cleared*

## CompState



The CompState metaclass contains a property that controls the appearance of components in statecharts.

## ShowCompName

The ShowCompName property specifies whether to show the component names in statecharts.

*Default = Cleared*

## ShowStereotype

The ShowStereotype property determines if, and how, a stereotype of the element is opened in a diagram. The possible values are:

- Label - The stereotype of the element is opened as a text label.
- None - The stereotype of the element is not displayed.

*Default = None*

# Constraint

The Constraint metaclass contains properties that specifies the constraints for statecharts.

## Compartments

The Compartments property determines which of the available compartments are displayed by default for the various types of elements.

The value for this property is a comma-delimited string containing the names of the compartments that should be visible.

Since the available compartments vary from element to element, it is recommended you not set the value of this property by using the Properties window or directly in the .prp file. Use the Display Options of the element to set which compartments are visible, and then use the Make Default option to apply these settings at the diagram or project level for new elements of this type.

*Default = Specification*

## ConstraintNotation

The ConstraintNotation property determines how Constraint elements are displayed. This property can be set to one of two styles:

- Note\_Style
- Box\_Style

If ConstraintNotation is set to Note\_Style, then the appearance of the element is determined by the value selected for the property Constraint::ShowForm: Note, Plain, or PushPin.

If ConstraintNotation is set to Box\_Style, then the element resembles a class-box, and you can control the display of compartments for the element.

*Default = Note\_Style*

## QuickNavigationCategories

Graphic elements in diagrams can include a list of shortcuts that are accessible from the title bar of the element. You can use the property QuickNavigationCategories to specify the categories of items that should be included in this shortcut list and the order in which the categories should appear in the list. The value of the property should be a comma-separated list of categories. The categories that can be included are:

- Activities
- DiagramReferences
- Diagrams
- Hyperlinks
- InternalBlockDiagrams
- MainBehavior
- MainDiagram
- References
- StateCharts

If you want greater control over the content of the navigation list (such as showing/hiding specific elements within a given category, or changing the order of elements within a specific category), use the Compartments list, which can be found under "Display Options".

*Default = Hyperlinks*

## ShowAnnotationContents

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the text displayed on the element in the diagram determined by the value that was set for the property ShowAnnotationContents. The property can take one of the following values:

- Name - the name of the element
- Label - the label assigned to the element
- Description - the content of the Description field
- Specification (for Requirements only) - the content of the Specification field

*Default = Description*

## ShowForm

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the appearance of the element is determined by the value that was set for the property ShowForm.

The possible values for this property vary for the different elements, as does the default value used. The values used are:

- Plain - No color background behind text
- Note - Color background behind text
- Pushpin - Color background plus pin icon

*Default = Note*

## ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram.

For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code.

The possible values are:

- Description - The content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is opened with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

*Default = Relative*

## ShowStereotype

The ShowStereotype property determines if, and how, a stereotype of the element is opened in a diagram. The possible values are:

- Label - The stereotype of the element is opened as a text label.

- Bitmap - The bitmap image that is associated with the stereotype of the element is opened.
- None - The stereotype of the element is not displayed.

*Default = Label*

## DefaultTransition

The DefaultTransition metaclass has properties that control the appearance of default transition.

### line\_style

The line\_style property specifies the default line style for a graphical item.

The possible values are:

- straight\_arrows - straight line.
- rectilinear\_arrows - rectilinear line with right-angled corners placed at appropriate locations, depending on the starting and ending points of the line
- spline\_arrows - curved line without corners
- rounded\_rectilinear\_arrows - similar to ordinary rectilinear lines but use rounded bends rather than straight ninety-degree angles

*Default = spline\_arrows*

*Default = spline\_arrows*

### ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram.

For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code.

The possible values are:

- Description - The content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed

- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is opened with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

*Default = Name*

## ShowStereotype

The ShowStereotype property determines if, and how, a stereotype of the element is opened in a diagram. The possible values are:

- Label - The stereotype of the element is opened as a text label.
- Bitmap - The bitmap image that is associated with the stereotype of the element is opened.
- None - The stereotype of the element is not displayed.

*Default = None*

## Depends

The Depends metaclass has properties that control the appearance of dependency relation lines in statecharts.

### line\_style

The line\_style property specifies the default line style for a graphical item.

The possible values are:

- straight\_arrows - straight line.
- rectilinear\_arrows - rectilinear line with right-angled corners placed at appropriate locations, depending on the starting and ending points of the line
- spline\_arrows - curved line without corners
- rounded\_rectilinear\_arrows - similar to ordinary rectilinear lines but use rounded bends rather than straight ninety-degree angles

*Default = spline\_arrows*

*Default = straight\_arrows*

### ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram.

For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code.

The possible values are:

- Description - The content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is opened with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

*Default = None*

## ShowStereotype

The ShowStereotype property determines if, and how, a stereotype of the element is opened in a diagram. The possible values are:

- Label - The stereotype of the element is opened as a text label.
- Bitmap - The bitmap image that is associated with the stereotype of the element is opened.
- None - The stereotype of the element is not displayed.

*Default = Label*

## DiagramFrame

The DiagramFrame metaclass contains properties that control the appearance of a diagram frame.

### DefaultSize

The DefaultSize property specifies default values for diagram frame dimensions.

*Default = 20, 20, 590, 500*

### ShowName

The ShowName property determines the text to display next to a graphic element in a diagram. For most elements, you can provide a name and a label. Your descriptive label can be useful in cases where the name itself might not be sufficient due to various constraints, such as the inability to use spaces if the name of the element is to appear in the code.

The possible values for this property vary for the different elements, as does the default value used. The possible values are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is displayed with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

Note that for some elements, such as destruction elements in sequence diagrams and timing diagrams, the property is a boolean property that hides/shows the name of the element.

*Default = Name*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## DigitalDisplay

The DigitalDisplay metaclass contains properties that determine the appearance and behavior of digital display controls on statecharts.

## ShowName

The ShowName property determines whether or not a caption is displayed for digital display elements, and if so, what text should be displayed. The possible values are:

- BindedElementFullPath - The full path of the attribute that is bound to the digital display.
- BindedElement - The name of the attribute that is bound to the digital display.
- Name - The name of the digital display element.
- None - No text is displayed.

*Default = Name*

## Gauge

The Gauge metaclass contains properties that determine the appearance and behavior of gauge controls on statecharts.

### ShowName

The ShowName property determines whether or not a caption is displayed for gauge elements, and if so, what text should be displayed. The possible values are:

- BindedElementFullPath - The full path of the attribute that is bound to the gauge.
- BindedElement - The name of the attribute that is bound to the gauge.
- Name - The name of the gauge element.
- None - No text is displayed.

*Default = Name*

## General

The General metaclass contains properties that specify general behavior of the statechart, such as whether to confirm deletion of objects.

### DeleteConfirmation

The DeleteConfirmation property specifies whether confirmation is required before deleting a graphical element from the model. Note that this property does not apply to statechart elements, which have a separate DeleteConfirmation property. The possible values are as follows:

- Always - Rational Rhapsody displays a confirmation window each time you try to delete an item from the model.
- Never - Confirmation is not required to delete an element.



- WhenNeeded - Rational Rhapsody displays a message asking for confirmation if there are references to the element (or for some other reason).

*Default = Never*

## ShowTransitionNumbers

For transitions in statecharts, you have the option of having a number displayed alongside each transition in the diagram. You can toggle the display of these numbers by modifying the value of the property ShowTransitionNumbers.

This is particularly useful if you have also enabled the generation of comments for requirements in statechart code (by setting the value of the property IncludeRequirementsAsComments to True). When both features are enabled, the comment generated for the requirement met by the transition will include the same number that is displayed for the transition in the diagram.

Note that the numbers displayed are used for mapping purposes only. There is no significance to the specific number that is used for any transition.

For more details, see the topic "Including requirements as comments in statechart and flowchart code" in the Rhapsody help.

*Default = False*

## SysMLDiagramKindTitle

When a diagram is created in SysML projects, the diagram frame contains a header for the diagram that includes information such as the name of the package that contains the diagram. The property SysMLDiagramKindTitle is used to specify a prefix for the diagram frame header that reflects the type of diagram. For example, when you create an internal block diagram, the header in the diagram frame starts with the prefix "ibd".

Note that the structure of the entire header is controlled by the property General::Graphics::DiagramHeader.

## HistoryConnector

Contains properties relating to the display of history connectors in statecharts.

## UseUMLNotation

In release 8.1 of Rhapsody, the notation for deep history connectors was modified to increase compliance with the UML standard. The property UseUMLNotation can be used to toggle the use of this notation. To preserve the previous appearance of deep history connectors for pre-8.1 models, the value of this property is set to False in the 8.1 backward compatibility settings.

*Default = True*

## Knob

The Knob metaclass contains properties that determine the appearance and behavior of knob controls on statecharts.

### Direction

The Direction property determines whether the knob controls are used to input data, display data, or both. The possible values are:

- In - The knobs are only used to input data for the attribute to which it is bound.
- Out - The knobs are only used to display data for the attribute to which it is bound.
- InOut - The knobs are used to input data and display data.

*Default = InOut*

### ShowName

The ShowName property determines whether or not a caption is displayed for knob elements, and if so, what text should be displayed. The possible values are:

- BindedElementFullPath - The full path of the attribute that is bound to the knob.
- BindedElement - The name of the attribute that is bound to the knob.
- Name - The name of the knob element.
- None - No text is displayed.

*Default = Name*

## Led

The LED metaclass contains properties that determine the appearance and behavior of LED controls on statecharts.

### ShowName

The ShowName property determines whether or not a caption is displayed for LED elements, and if so, what text should be displayed. The possible values are:

- BindedElementFullPath - The full path of the attribute that is bound to the LED.

- **BindedElement** - The name of the attribute that is bound to the LED.
- **Name** - The name of the LED element.
- **None** - No text is displayed.

*Default = Name*

## LevelIndicator

The LevelIndicator metaclass contains properties that determine the appearance and behavior of level indicator controls on statecharts.

### ShowName

The ShowName property determines whether or not a caption is displayed for level indicator elements, and if so, what text should be displayed. The possible values are:

- **BindedElementFullPath** - The full path of the attribute that is bound to the level indicator.
- **Name** - The name of the level indicator element.
- **None** - No text is displayed.

*Default = Name*

## MatrixDisplay

The MatrixDisplay metaclass contains properties that determine the appearance and behavior of matrix display controls on statecharts.

### ShowName

The ShowName property determines whether or not a caption is displayed for matrix display elements, and if so, what text should be displayed. The possible values are:

- **BindedElementFullPath** - The full path of the attribute that is bound to the matrix display.
- **BindedElement** - The name of the attribute that is bound to the matrix display.
- **Name** - The name of the matrix display element.
- **None** - No text is displayed.

*Default = Name*

## Meter

The Meter metaclass contains properties that determine the appearance and behavior of meter controls on statecharts.

### ShowName

The ShowName property determines whether or not a caption is displayed for meter elements, and if so, what text should be displayed. The possible values are:

- BindedElementFullPath - The full path of the attribute that is bound to the meter.
- BindedElement - The name of the attribute that is bound to the meter.
- Name - The name of the meter element.
- None - No text is displayed.

*Default = Name*

## Note

The Note metaclass contains properties that specify the display of notes in statecharts.

### ShowForm

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the appearance of the element is determined by the value that was set for the property ShowForm.

The possible values for this property vary for the different elements, as does the default value used. The values used are:

- Plain - No color background behind text
- Note - Color background behind text
- Pushpin - Color background plus pin icon

*Default = Note*

## OnOffSwitch

The OnOffSwitch metaclass contains properties that determine the appearance and behavior of on/off switch controls on statecharts.

## Direction

The Direction property determines whether the on/off switch controls are used to input data, display data, or both. The possible values are:

- In - The on/off switches are only used to input data for the attribute to which it is bound.
- Out - The on/off switches are only used to display data for the attribute to which it is bound.
- InOut - The on/off switches are used to input data and display data.

*Default = InOut*

## ShowName

The ShowName property determines whether or not a caption is displayed for on/off switch elements, and if so, what text should be displayed. The possible values are:

- BindedElementFullPath - The full path of the attribute that is bound to the on/off switch.
- BindedElement - The name of the attribute that is bound to the on/off switch.
- Name - The name of the on/off switch element.
- None - No text is displayed.

*Default = Name*

## PushButton

The PushButton metaclass contains properties that determine the appearance and behavior of push button controls on statecharts.

### ButtonFont

The ButtonFont property lets you select the font to use for the text on the face of a push button control.

To change the value of the property, click the "..." button in the box next to the property value to open the Font window. The value of the property affects both buttons already on the statechart and new buttons added to the statechart. (The display of buttons already on the statechart changes only after you refresh the diagram.)

*Default = Arial 10 NoBold NoItalic*

### ShowName

The ShowName property determines whether or not a caption is displayed for push button elements, and if so, what text should be displayed. The possible values are:

- **BindedElementFullPath** - The full path of the attribute that is bound to the push button.
- **BindedElement** - The name of the attribute that is bound to the push button.
- **Name** - The name of the push button element.
- **None** - No text is displayed.

*Default = Name*

## Requirement

The Requirement metaclass contains properties that specify requirements in statecharts.

### Compartments

The Compartments property determines which of the available compartments are displayed by default for the various types of elements.

The value for this property is a comma-delimited string containing the names of the compartments that should be visible.

Since the available compartments vary from element to element, it is recommended you not set the value of this property by using the Properties window or directly in the .prp file. Use the Display Options of the element to set which compartments are visible, and then use the Make Default option to apply these settings at the diagram or project level for new elements of this type.

*Default = ID, Specification*

### QuickNavigationCategories

Graphic elements in diagrams can include a list of shortcuts that are accessible from the title bar of the element. You can use the property QuickNavigationCategories to specify the categories of items that should be included in this shortcut list and the order in which the categories should appear in the list. The value of the property should be a comma-separated list of categories. The categories that can be included are:

- Activities
- DiagramReferences
- Diagrams
- Hyperlinks
- InternalBlockDiagrams
- MainBehavior
- MainDiagram
- References
- StateCharts

If you want greater control over the content of the navigation list (such as showing/hiding specific elements within a given category, or changing the order of elements within a specific category), use the Compartments list, which can be found under "Display Options".

*Default = Hyperlinks*

## RequirementNotation

The RequirementNotation property determines how Requirement elements are displayed. This property can be set to one of two styles:

- Note\_Style
- Box\_Style

If RequirementNotation is set to Note\_Style, then the appearance of the element is determined by the value selected for the property Requirement::ShowForm: Note, Plain, or PushPin.

If RequirementNotation is set to Box\_Style, then the element resembles a class-box, and you can control the display of compartments for the element.

*Default = Note\_Style*

## ShowAnnotationContents

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the text displayed on the element in the diagram determined by the value that was set for the property ShowAnnotationContents. The property can take one of the following values:

- Name - the name of the element
- Label - the label assigned to the element
- Description - the content of the Description field
- Specification (for Requirements only) - the content of the Specification field

*Default = Description*

## ShowForm

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the appearance of the element is determined by the value that was set for the property ShowForm.

The possible values for this property vary for the different elements, as does the default value used. The values used are:

- Plain - No color background behind text
- Note - Color background behind text
- Pushpin - Color background plus pin icon

*Default = Note*

## ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram.

For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code.

The possible values are:

- Description - The content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is opened with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

*Default = Relative*

## ShowStereotype

The ShowStereotype property determines if, and how, a stereotype of the element is opened in a diagram. The possible values are:

- Label - The stereotype of the element is opened as a text label.
- Bitmap - The bitmap image that is associated with the stereotype of the element is opened.
- None - The stereotype of the element is not displayed.

*Default = Label*

## SpecificationAnnotationFormat

When a requirement is displayed on a diagram, you can choose to have the element show one of the following: its name, its label, its description, the specification text for the requirement.

If you choose to display the specification text, you can use the property SpecificationAnnotationFormat to customize the text that is displayed.

The following keywords can be included in the value of the property: \$Specification (to include the



specification text) and \$ID (to include the ID that was specified in the Features dialog for the requirement).

Note that this property affects the displayed text only when a requirement uses the "Note" notation style. When using the "Box" notation style, only the specification text is displayed, regardless of the value of this property.

*Default = \$Specification*

## SendAction

The SendAction metaclass contains properties that relate to Send Action elements in statecharts.

### ShowNotation

The ShowNotation property determines what caption is opened for new Send Action elements that are added to a statechart. The property can take any of the following values:

- Name - the name of the Send Action element
- Label - the label of the Send Action element
- FullNotation - the event that is to be sent and the object that is to receive the event (target)
- Event - the event that is to be sent

This property can be set at the diagram level or higher.

Note that when you change the value of this property, the display of any new Send Action elements are affected, but the display of Send Action elements already on the diagram remains as is. (The display of existing elements on the diagram can be controlled by selecting Display Options from the View menu.)

*Default = FullNotation*

### ShowStereotype

The ShowStereotype property determines if, and how, a stereotype of the element is opened in a diagram. The possible values are:

- Label - The stereotype of the element is opened as a text label.
- Bitmap - The bitmap image that is associated with the stereotype of the element is opened.
- None - The stereotype of the element is not displayed.

*Default = None*

## Slider

The Slider metaclass contains properties that determine the appearance and behavior of slider controls on statecharts.

### Direction

The Direction property determines whether slider controls are used to input data, display data, or both. The possible values are:

- In - The sliders are only used to input data for the attribute to which it is bound.
- Out - The sliders are only used to display data for the attribute to which it is bound.
- InOut - The sliders are used to input data and display data.

*Default = InOut*

### ShowName

The ShowName property determines whether or not a caption is displayed for slider elements, and if so, what text should be displayed. The possible values are:

- BindedElementFullPath - The full path of the attribute that is bound to the slider.
- BindedElement - The name of the attribute that is bound to the slider.
- Name - The name of the slider element.
- None - No text is displayed.

*Default = Name*

## State

The State metaclass contains properties that control the appearance of state boxes.

### Compartments

The Compartments property determines which of the available compartments are displayed by default for the various types of elements.

The value for this property is a comma-delimited string containing the names of the compartments that should be visible.

Since the available compartments vary from element to element, it is recommended you not set the value

of this property by using the Properties window or directly in the .prp file. Use the Display Options of the element to set which compartments are visible, and then use the Make Default option to apply these settings at the diagram or project level for new elements of this type.

*Default = Empty MultiLine*

## InheritedStateColor

If a class is derived from a class that has a statechart, a statechart is created for the derived class. This new statechart includes the elements inherited from the statechart of the base class, and you can add additional elements to the statechart of the derived class.

In order to help you distinguish between the inherited elements and the other elements in the statechart, a different color is used for the outline of the inherited elements. You can control the color used for inherited elements by modifying the value of the property InheritedStateColor. You can also control the color of the text on inherited elements by modifying the value of the property InheritedStateFontColor.

Note that if you modify the values of these properties, the new values will affect only the display of inherited statecharts that are created subsequently, but not the display of existing inherited statecharts.

*Default = RGB (191, 191, 191)*

## InheritedStateFontColor

If a class is derived from a class that has a statechart, a statechart is created for the derived class. This new statechart includes the elements inherited from the statechart of the base class, and you can add additional elements to the statechart of the derived class.

In order to help you distinguish between the inherited elements and the other elements in the statechart, a different color is used for the outline of the inherited elements. You can control the color used for inherited elements by modifying the value of the property InheritedStateColor. You can also control the color of the text on inherited elements by modifying the value of the property InheritedStateFontColor.

Note that if you modify the values of these properties, the new values will affect only the display of inherited statecharts that are created subsequently, but not the display of existing inherited statecharts.

*Default = RGB (128, 128, 128)*

## ShowDescription

The ShowDescription property specifies whether or not the descriptions for the states in the statechart should be displayed.

*Default = Cleared*

## ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram.

For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code.

The possible values are:

- Description - The content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is opened with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

*Default = Name\_only*

## ShowReactions

The ShowReactions property specifies whether reactions are displayed in the corresponding states.

*Default = Cleared*

## ShowStereotype

The ShowStereotype property determines if, and how, a stereotype of the element is opened in a diagram. The possible values are:

- Label - The stereotype of the element is opened as a text label.
- Bitmap - The bitmap image that is associated with the stereotype of the element is opened.
- None - The stereotype of the element is not displayed.

*Default = Label*

# StateDiagram

The StateDiagram metaclass contains properties that affect the display of statecharts.

## DefaultView

The DefaultView property can be used to determine the default view for statecharts - diagram view or tabular view. This property can be set at the level of individual statecharts or higher.

Note that if the property is set at the package level or higher, it affects the display of all statecharts in the package, not just new statecharts created after the value of the property was changed.

*Default = Diagram view*

## Fillcolor

The Fillcolor property specifies the default fill color for the object.

*Default = 218,218,218*

## TextBox

The TextBox metaclass contains properties that determine the appearance and behavior of text box controls on statecharts.

## Direction

The Direction property determines whether text box controls are used to input data, display data, or both. The possible values are:

- In - The text boxes are only used to input data for the attribute to which it is bound.
- Out - The text boxes are only used to display data for the attribute to which it is bound.
- InOut - The text boxes are used to input data and display data.

*Default = InOut*

## ShowName

The ShowName property determines whether or not a caption is displayed for text box elements, and if so, what text should be displayed. The possible values are:

- BindedElementFullPath - The full path of the attribute that is bound to the text box.
- BindedElement - The name of the attribute that is bound to the text box.
- Name - The name of the text box element.
- None - No text is displayed.

*Default = Name*

# Transition

The Transition metaclass contains a property that controls the appearance of transitions in statecharts.

## ArrowHead

The property ArrowHead can be used to select the style of arrowhead you would like to use for transitions in diagrams such as activity diagrams and statecharts - an open arrowhead or closed arrowhead.

Note that when you change the value of the property, existing transitions will be displayed as they were previously until you refresh the diagram (F5).

*Default = OpenArrow*

## line\_style

The line\_style property specifies the default line style for a graphical item.

The possible values are:

- straight\_arrows - straight line.
- rectilinear\_arrows - rectilinear line with right-angled corners placed at appropriate locations, depending on the starting and ending points of the line
- spline\_arrows - curved line without corners
- rounded\_rectilinear\_arrows - similar to ordinary rectilinear lines but use rounded bends rather than straight ninety-degree angles

*Default = spline\_arrows*

*Default = spline\_arrows*

## ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram.

For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code.

The possible values are:

- Description - The content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example,

package\_1::package\_1b::class\_0

- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is opened with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

*Default = Name*

## ShowStereotype

The ShowStereotype property determines if, and how, a stereotype of the element is opened in a diagram. The possible values are:

- Label - The stereotype of the element is opened as a text label.
- Bitmap - The bitmap image that is associated with the stereotype of the element is opened.
- None - The stereotype of the element is not displayed.

*Default = None*

# STLContainers

---

Rational Rhapsody uses containers to implement to-many relations between objects. These include relationships of one object to many, or many objects to many.

The STLContainers subject contains metaclasses that contain properties that control the implementing of relations.

Each property in this subject includes the default value for each container type and relation type. For easier readability, the placeholder RelationType in these values represents all the other relation types that are not explicitly detailed. For example, the relation type User might have the default value of an empty string, whereas all the other relation types have the value of an empty MultiLine.

## BoundedOrdered

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed sequentially.

### Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter: \$cname-push\_back(\$item)

For maps (qualified relations), the following command inserts an item into the map based on a key: \$cname-insert(map\$keyType,\$target\*::value\_type( \$keyName,\$item))

The default is \$cname->push\_back(\$item).

### Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector: new vector\$target\*

The default is \$cname = \$CreateStatic.

### CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it



the name stored in \$cname: `vector$target* $cname()`

The default is `$CType $cname`.

## CreateStatic

The CreateStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to Reference.

The default is `new $CType`.

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the `vector$target*` collection type determines the type of the variable `cl` on the left side of the assignment:

```
vectorClient*>* cl = new vectorClient*>(2);
```

In this case, the class `Client` is the `$target` of vector operations.

The default is `std::vector <$RelationTargetType>`.

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about Composite Types.

The default is `$CType $cname`.

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: `$me$cname` The variable `$me` is replaced with the object context variable as specified by the `Me` property. The variable `$cname` is replaced with the name of the container, which is the role name for the relation.

The default is `$cname`.

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The `ContainerTypes::RelationType::GetAt` property specifies a template for the body of the operation. For example, the following command generates code that call the `at()` operation for the container to retrieve the item at the indexed position: `$cname-at($index)`

The default is `$cname->operator[]($index)`.

## GetEnd

The `GetEnd` property specifies the command used to retrieve the last item in the container. For example, the following command calls the `end()` operation for the container to locate the last item in the collection: `$cname-end()` This property and `GetEndGenerate` were created to adhere to the standard library convention for “finding” where iteration should end. `GetEnd` is generated where `Get` is generated. The method name is defined by using the `CG::Relation::GetEnd` and `CG::Relation::GetEndGenerate` properties.

The default is `$cname->end()`.

## GetKey

The `GetKey` property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator `[]`, which has been overloaded according to the STL definition for maps: `$cname-operator[]($keyName)`

## IncludeDirective

The `IncludeDirective` property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The `#include` directives are added to the header file. (Default)
- weak - The `#include` directives are added to the source file with forward declarations in the header file.

## IncludeFiles

The `IncludeFiles` property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference.

If the `Containment` property is set to `Reference`, a forward declaration of the container is added to the class specification file, and the `#include` is added to the class implementation file. A new set of properties that

describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

The default is `<vector >, <iterator >`.

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: `$cname()` The default is `$cname()`.

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: `pos=0; pos$multiplicity; pos++; $cname[pos]=NULL`

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: `vector$target*::const_iterator $iterator; $iterator=$cname-begin()`

The default is as follows: `$IterType $iterator; $IterReset`

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

The default is `$IterCreate`.

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator` (Default) This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item: `$iterator++` (Default)

## IterIncrementForCleanUp

The `IterIncrementForCleanUp` property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

The default is `$iterator = $cname->begin()`.

## IterIncrementForInit

The `IterIncrementForInit` property specifies the code that increments the iterator, for certain initialization cases.

The default is `$iterator = $cname->begin()`.

## IterReset

The `IterReset` property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection: `$iterator=$cname-begin()` (Default)

## IterReturntype

The `IterReturntype` property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

The default is `$IterType`.

## IterTest

The `IterTest` property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname-end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, `IterTest` retrieves the current item in the collection (the same as `IterGetCurrent`).

The default is `$iterator != $cname->end()`.

## IterType

The `IterType` property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<>`, as defined in the STL. `vector$target*::const_iterator` You can change the iterator type to one of your own choice.

The default is `$CType::const_iterator`.

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

The default value for C++ is as follows: for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$name. The default value for all other subjects is \$name.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

The default is \$(constant)\$target\$reference.

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. vector\$target\*::iterator pos=find(\$cname-begin(), \$cname-end(),\$item);\$cname-erase(pos) This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. pair\$keyType,\$target\* p; p.second=\$item; map\$keyType,\$target\*::iterator pos=find(\$cname-begin(), \$cname-end(),p); \$cname-erase(pos)

The default is as follows: \$CType::iterator pos = std::find(\$cname->begin(), \$cname->end(),\$item); if (pos != \$cname->end()) { \$cname->erase(pos); }

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items: \$cname-clear() The default is \$cname->clear().

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: \$cname-erase(\$keyName)

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved: \$cname = \$item

## Type

The Type property specifies the type of the container as a pointer to the relation.

# BoundedUnordered

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed sequentially.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter: \$cname-push\_back(\$item)

For maps (qualified relations), the following command inserts an item into the map based on a key: \$cname-insert(map\$keyType,\$target\*::value\_type( \$keyName,\$item))

The default is \$cname->push\_back(\$item).

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector: new vector\$target\* The default is \$cname = \$CreateStatic.

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname: `vector$target* $cname()` The default is `$CType $cname`.

## CreateStatic

The CreateStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to Reference.

The default is `new $CType`.

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the `vector$target*` collection type determines the type of the variable `cl` on the left side of the assignment:

```
vectorClient*>* cl = new vectorClient*>(2);
```

In this case, the class `Client` is the \$target of vector operations.

The default is `std::list<$RelationTargetType>`.

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the `Find()` operation for container \$cname to locate the \$item: `$cname-find($item)`

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about Composite Types.

The default is `$CType $cname`.

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: `$me$cname` The variable `$me` is replaced with the object context variable as specified by the `Me` property. The variable `$cname` is replaced with the name of the container, which is the role name for the relation.

The default is `$cname`.

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that call the at() operation for the container to retrieve the item at the indexed position: \$cname-at(\$index)

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end() This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

The default is \$cname->end().

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps: \$cname-operator[](\$keyName)

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file. (Default)
- weak - The #include directives are added to the source file with forward declarations in the header file.

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.



Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

The default is <list>,<iterator>.

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: \$cname() (Default)

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: pos=0; pos\$multiplicity; pos++; \$cname[pos]=NULL

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: vector\$target\*::const\_iterator \$iterator; \$iterator=\$cname-begin()

The default is as follows: \$IterType \$iterator; \$IterReset

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

The default is \$IterCreate.

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: \*\$iterator This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

The default is `*$iterator`.

## IterIncrement

The `IterIncrement` property specifies the code that increments the iterator. For example, the following command moves the `$iterator` ahead one item: `$iterator++` (Default)

## IterIncrementForCleanUp

The `IterIncrementForCleanUp` property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

The default is `$IterReset`.

## IterIncrementForInit

The `IterIncrementForInit` property specifies the code that increments the iterator, for certain initialization cases.

The default is `$IterReset`.

## IterInit

The `IterInit` property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

## IterReset

The `IterReset` property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection: `$iterator=$cname->begin()`

The default is `$iterator = $cname->begin()`.

## IterReturnType

The `IterReturnType` property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

*(Default = `$IterType`)*

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $sname-end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as `IterGetCurrent`).

The default is `$iterator != $sname->end()`.

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<`, as defined in the STL. `vector$target*::const_iterator` You can change the iterator type to one of your own choice.

The default is `$CType::const_iterator`.

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the `OMContainers.StaticArray` metaclass) can use any other property in the same metaclass. Therefore, other properties that use `$Loop` expand the specified Loop property body.

The default value for C++ is as follows: `for (int pos = 0; pos $multiplicity; ++pos)` The default value for C is as follows: `int pos; for (pos = 0; pos $multiplicity; ++pos)` The default value for Java is as follows: `for (int pos = 0; pos $multiplicity; pos++)`

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for `RiCContainers::EmbeddedScalar` is `$me$sname`. The default value for all other subjects is `$sname`.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

The default is `$(constant)$target$reference`.

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the `find()` operation to point the iterator to the item to item to

be removed and then call `erase()` to remove it. `vector$target*::iterator pos=find($cname-begin(), $cname-end(),$item);$cname-erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair `p`, whose second element, `p.second`, is the item to be removed. The `find()` operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair$keyType,$target* p; p.second=$item; map$keyType,$target*::iterator pos=find($cname-begin(), $cname-end(),p); $cname-erase(pos)`

The default is as follows: `$CType::iterator pos = std::find($cname->begin(), $cname->end(),$item); if (pos != $cname->end()) { $cname->erase(pos); }`

## RemoveAll

The `RemoveAll` property specifies the command used to remove all items from the container. For example, the following command calls the `clear()` operation for the container to remove all items: `$cname-clear()`

The default is `$cname->clear()`.

## RemoveKey

The `RemoveKey` property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the `erase()` operation for the container, passing it the `$keyName`, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

## Set

The `Set` property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, `$cname`, is a role name of the relation itself, because there is only one class involved: `$cname = $item`

## Type

The `Type` property specifies the type of the container as a pointer to the relation.

## EmbeddedFixed

The `EmbeddedFixed` metaclass defines properties for implementing embedded fixed relations.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter: `$cname-push_back($item)`

For maps (qualified relations), the following command inserts an item into the map based on a key: `$cname-insert(map$keyType,$target*::value_type( $keyName,$item))`

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector: `new vector$target*`

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`: `vector$target* $cname()`

The default is `$CType`.

## CreateStatic

The CreateStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to `Reference`.

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the `vector$target*` collection type determines the type of the variable `cl` on the left side of the assignment:

```
vectorClient*>* cl = new vectorClient*>(2);
```

In this case, the class `Client` is the `$target` of vector operations.

The default is `$(constant)$target $cname[$multiplicity]`.

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the `Find()` operation for container `$cname` to locate the `$item`: `$cname-find($item)`

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about Composite Types.

The default is `$RelationTargetType $cname[$multiplicity]`.

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: `$me$cname` The variable `$me` is replaced with the object context variable as specified by the `Me` property. The variable `$cname` is replaced with the name of the container, which is the role name for the relation.

The default is `&$cname`.

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The `ContainerTypes::RelationType::GetAt` property specifies a template for the body of the operation. For example, the following command generates code that call the `at()` operation for the container to retrieve the item at the indexed position: `$cname-at($index)`

The default is `($RelationTargetType) &$cname[$index]`.

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the `end()` operation for the container to locate the last item in the collection: `$cname-end()` This property and `GetEndGenerate` were created to adhere to the standard library convention for “finding” where iteration should end. `GetEnd` is generated where `Get` is generated. The method name is defined by using the `CG::Relation::GetEnd` and `CG::Relation::GetEndGenerate` properties.

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator `[]`, which has been overloaded according to the STL definition for maps:

\$cname-operator[](\$keyName)

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file. (Default)

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: \$cname()

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: pos=0; pos\$multiplicity; pos++; \$cname[pos]=NULL

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the

container. For example, the following command instantiates the iterator class and points it to the first item in the container: `vector$target*::const_iterator $iterator; $iterator=$cname-begin()`

The default is `$IterType $iterator = 0;`.

## IterCreateByValue

The `IterCreateByValue` property is the same as `IterCreate`, but instantiates the iterator by value.

The default is `$IterCreate`.

## IterGetCurrent

The `IterGetCurrent` property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator` This value is the same as that for `IterTest` only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for `IterGetCurrent` returns a pointer to the current item in the collection.

The default is as follows: `((($RelationTargetType)&$cname[$iterator])`

## IterIncrement

The `IterIncrement` property specifies the code that increments the iterator. For example, the following command moves the `$iterator` ahead one item: `$iterator++` (Default)

## IterIncrementForCleanUp

The `IterIncrementForCleanUp` property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

The default is `$IterIncrement`.

## IterIncrementForInit

The `IterIncrementForInit` property specifies the code that increments the iterator, for certain initialization cases.

The default is `$IterIncrement`.

## IterInit

The `IterInit` property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.



*Default = Blank*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection: \$iterator=\$cname-begin()

The default is \$iterator = 0.

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

The default is \$IterType.

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname-end() With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

The default is \$iterator < \$multiplicity.

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<, as defined in the STL. vector\$target\*::const\_iterator You can change the iterator type to one of your own choice.

The default is int.

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

The default value for C++ is as follows: for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for `RiCContainers::EmbeddedScalar` is `$me$name`. The default value for all other subjects is `$name`.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

The default is `$(constant)$target$reference`.

## Remove

The Remove property specifies the command used to remove an item from a relation. For example, the following commands call the `find()` operation to point the iterator to the item to be removed and then call `erase()` to remove it. `vector$target*::iterator pos=find($cname-begin(), $cname-end(),$item);$cname-erase(pos)` This operation applies only to “to-many” (non-scalar) containers. For maps (qualified relations), the following commands create a pair `p`, whose second element, `p.second`, is the item to be removed. The `find()` operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair$keyType,$target* p; p.second=$item; map$keyType,$target*::iterator pos=find($cname-begin(), $cname-end(),p); $cname-erase(pos)`

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the `clear()` operation for the container to remove all items: `$cname-clear()`

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the `erase()` operation for the container, passing it the `$keyName`, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, `$cname`, is a role name of the relation itself, because there is only one class involved: `$cname = $item`

## Type

The Type property specifies the type of the container as a pointer to the relation.

## EmbeddedScalar

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed sequentially.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter: `$cname-push_back($item)`

For maps (qualified relations), the following command inserts an item into the map based on a key: `$cname-insert(map$keyType,$target*::value_type( $keyName,$item))`

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector: `new vector$target*`

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`: `vector$target* $cname()`

## CreateStatic

The CreateStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to Reference.

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example,

when instantiating a vector that contains references to two clients, the `vector$target*` collection type determines the type of the variable `cl` on the left side of the assignment:

```
vectorClient*>* cl = new vectorClient*>(2);
```

In this case, the class `Client` is the `$target` of vector operations.

The default is `$(constant)$target`.

## Find

The `Find` property specifies the command used to locate an item in a container. For example, the following command calls the `Find()` operation for container `$cname` to locate the `$item`: `$cname-find($item)`

## FullTypeDefinition

The `FullTypeDefinition` property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about Composite Types.

The default is `$CType$reference $cname`.

## Get

The `Get` property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: `$me$cname` The variable `$me` is replaced with the object context variable as specified by the `Me` property. The variable `$cname` is replaced with the name of the container, which is the role name for the relation.

The default is `&$cname`.

## GetAt

The `GetAt` property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The `ContainerTypes::RelationType::GetAt` property specifies a template for the body of the operation. For example, the following command generates code that call the `at()` operation for the container to retrieve the item at the indexed position: `$cname-at($index)`

## GetEnd

The `GetEnd` property specifies the command used to retrieve the last item in the container. For example, the following command calls the `end()` operation for the container to locate the last item in the collection:

\$cname-end() This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:  
\$cname-operator[](\$keyName)

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file. (Default)

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

The default is \$(constRT)\$target\*.

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default is \$cname.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

The default is \$(constant)\$target\*.

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to be removed and then call erase() to remove it. `vector$target*::iterator pos=find($cname-begin(), $cname-end(), $item); $cname-erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair$keyType, $target* p; p.second=$item; map$keyType, $target*::iterator pos=find($cname-begin(), $cname-end(), p); $cname-erase(pos)`

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items: `$cname-clear()`

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved: `$cname = $item`

## Type

The Type property specifies the type of the container as a pointer to the relation.

## Fixed

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed sequentially.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter: `$cname-push_back($item)`

For maps (qualified relations), the following command inserts an item into the map based on a key: `$cname-insert(map$keyType,$target*::value_type( $keyName,$item))`

The default is `$cname->insert($cname->begin(), $item)`.

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector: `new vector$target*`

The default is `$cname = $CreateStatic`.

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`: `vector$target* $cname()`

The default is `$CType $cname`.

## CreateStatic

The CreateStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to Reference.

The default is `new $CType`.

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the `vector$target*` collection type determines the type of the variable `cl` on the left side of the assignment:

```
vectorClient*>* cl = new vectorClient*>(2);
```

In this case, the class `Client` is the `$target` of vector operations.

The default is `std::vector<$RelationTargetType>`.

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item: `$cname-find($item)`

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about Composite Types.

The default is `$CType $cname`.

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: `$me$cname` The variable `$me` is replaced with the object context variable as specified by the Me property. The variable `$cname` is replaced with the name of the container, which is the role name for the relation.

The default is `$cname`.

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The `ContainerTypes::RelationType::GetAt` property specifies a template for the body of the operation. For example, the following command generates code that call the `at()` operation for the container to retrieve the item at the indexed position: `$cname-at($index)`

The default is `$cname->operator[]($index)`.

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the `end()` operation for the container to locate the last item in the collection: `$cname-end()` This property and `GetEndGenerate` were created to adhere to the standard library convention for “finding” where iteration should end. `GetEnd` is generated where `Get` is generated. The method name is defined by using the `CG::Relation::GetEnd` and `CG::Relation::GetEndGenerate` properties.



The default is `$cname->end()`.

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:  
`$cname-operator[]($keyName)`

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file. (Default)
- weak - The #include directives are added to the source file with forward declarations in the header file.

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

The default is `< vector>,< iterator>`.

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: `$cname()` (Default)

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: `pos=0; pos$multiplicity; pos++; $cname[pos]=NULL`

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector$target*::const_iterator $iterator; $iterator=$cname-begin()
```

The default is as follows: \$IterType \$iterator; \$IterReset

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

The default is \$IterCreate.

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: \*\$iterator This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

The default is \*\$iterator.

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item: \$iterator++ (Default)

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

The default is \$IterReset.

## IterIncrementForInit

The `IterIncrementForInit` property specifies the code that increments the iterator, for certain initialization cases.

The default is `$IterReset`.

## IterInit

The `IterInit` property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

## IterReset

The `IterReset` property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection: `$iterator=$cname-begin()` The default is `$iterator = $cname->begin()`.

## IterReturnType

The `IterReturnType` property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

The default is `$IterType`.

## IterTest

The `IterTest` property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname-end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, `IterTest` retrieves the current item in the collection (the same as `IterGetCurrent`).

The default is `$iterator != $cname->end()`.

## IterType

The `IterType` property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<`, as defined in the STL. `vector$target*::const_iterator` You can change the iterator type to one of your own choice.

The default is `$CType::const_iterator`.

## Loop

The `Loop` property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the `OMContainers.StaticArray` metaclass) can use any other property in

the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

The default value for C++ is as follows: for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$name. The default value for all other subjects is \$name.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

The default is \$(constant)\$target\$reference.

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. vector\$target\*::iterator pos=find(\$cname-begin(), \$cname-end(),\$item);\$cname-erase(pos) This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. pair\$keyType,\$target\* p; p.second=\$item; map\$keyType,\$target\*::iterator pos=find(\$cname-begin(), \$cname-end(),p); \$cname-erase(pos)

The default is as follows:

```
$CType::iterator pos = std::find($cname->begin(), $cname->end(),$item); if (pos != $cname->end()) {  
$cname->erase(pos); }
```

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items: \$cname-clear() (Default)

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: \$cname-erase(\$keyName)

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved: \$cname = \$item

## Type

The Type property specifies the type of the container as a pointer to the relation.

## General

The General metaclass contains properties that enable you to set the directives and include files for the container.

### ContainerDirectives

The ContainerDirectives property specifies the preprocessor directives that are necessary when compiling code that uses a particular container library.

No additional directives are required when you use OMContainers.

The default is as follows:

```
#ifdef _MSC_VER // disable Microsoft compiler warning (debug information truncated) #pragma  
warning(disable: 4786) #endif
```

### ContainerIncludes

The ContainerIncludes property specifies header files that must be included when you use a particular container library.

For example, when you use STLContainers, the following string causes one #include directive per container to be added to generated files (such as #include string): string, algorithm, vector, list, map, iterator Whether the #include directives are added to source or header files depends on the value of the IncludeDirective property.

The default is string,algorithm.

# Qualified

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed sequentially.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter: `$cname-push_back($item)`

For maps (qualified relations), the following command inserts an item into the map based on a key: `$cname-insert(map$keyType,$target*::value_type( $keyName,$item))`

The default is as follows:

```
$cname->insert($CType::value_type($keyName, $item))
```

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector: `new vector$target*`

The default is `$cname = $CreateStatic`.

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`: `vector$target* $cname()`

The default is `$CType $cname`.

## CreateStatic

The CreateStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to `Reference`.

The default is `new $CType`.

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector\$target\* collection type determines the type of the variable cl on the left side of the assignment:

```
vectorClient*>* cl = new vectorClient*>(2);
```

In this case, the class Client is the \$target of vector operations.

The default is std::map<\$keyType, \$RelationTargetType>.

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item: \$cname-find(\$item)

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about Composite Types.

The default is \$CType \$cname.

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

The default is \$cname.

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that call the at() operation for the container to retrieve the item at the indexed position: \$cname-at(\$index)

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end() This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

The default is \$cname->end().

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps: \$cname-operator[](\$keyName)

The default is as follows:

```
($cname->find($keyName) != $cname->end() ? (*$cname->find($keyName)).second : NULL)
```

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file. (Default)
- weak - The #include directives are added to the source file with forward declarations in the header file.

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.



The default is as follows: <map>,<iterator>,<oxf/OMValueCompare.h>

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: \$cname() (Default)

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: pos=0; pos\$multiplicity; pos++; \$cname[pos]=NULL

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: vector\$target\*::const\_iterator \$iterator; \$iterator=\$cname-begin()

The default is as follows:

```
$IterType $iterator; $IterReset
```

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

The default is \$IterCreate.

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: \*\$iterator This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

The default is (\*\$iterator).second.

## IterIncrement

The `IterIncrement` property specifies the code that increments the iterator. For example, the following command moves the `$iterator` ahead one item: `$iterator++` (Default)

## **IterIncrementForCleanUp**

The `IterIncrementForCleanUp` property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

The default is `$IterReset`.

## **IterIncrementForInit**

The `IterIncrementForInit` property specifies the code that increments the iterator, for certain initialization cases.

The default is `$IterReset`.

## **IterInit**

The `IterInit` property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

## **IterReset**

The `IterReset` property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection: `$iterator=$cname-begin()` The default is `$iterator = $cname->begin()`.

## **IterReturnType**

The `IterReturnType` property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

The default is `$IterType`.

## **IterTest**

The `IterTest` property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname-end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, `IterTest` retrieves the current item in the collection (the same as `IterGetCurrent`).

The default is `$iterator != $cname->end()`.

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<`, as defined in the STL. `vector$target*::const_iterator` You can change the iterator type to one of your own choice.

The default is `$CType::const_iterator`.

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the `OMContainers.StaticArray` metaclass) can use any other property in the same metaclass. Therefore, other properties that use `$Loop` expand the specified Loop property body.

The default value for C++ is as follows: `for (int pos = 0; pos $multiplicity; ++pos)` The default value for C is as follows: `int pos; for (pos = 0; pos $multiplicity; ++pos)` The default value for Java is as follows: `for (int pos = 0; pos $multiplicity; pos++)`

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for `RiCContainers::EmbeddedScalar` is `$me$name`. The default value for all other subjects is `$name`.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

The default is `$(constant)$target$reference`.

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the `find()` operation to point the iterator to the item to item to be removed and then call `erase()` to remove it. `vector$target*::iterator pos=find($cname-begin(), $cname-end(),$item);$cname-erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair `p`, whose second element, `p.second`, is the item to be removed. The `find()` operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair$keyType,$target* p; p.second=$item; map$keyType,$target*::iterator pos=find($cname-begin(), $cname-end(),p); $cname-erase(pos)`

The default is as follows:

```
$CType::iterator pos = std::find_if($cname->begin(), $cname->end(), OMValueCompare<const $keyType,$RelationTargetType>($item)); if (pos != $cname->end()) { $cname->erase(pos); }
```

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items:

```
$cname-clear()
```

The default is `$cname->clear()`.

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed:

```
$cname-erase($keyName)
```

The default is `$cname->erase($keyName)`.

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved: `$cname = $item`

## Type

The Type property specifies the type of the container as a pointer to the relation.

## Scalar

The Scalar metaclass defines properties for implementing scalar relations.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter: `$cname-push_back($item)`

For maps (qualified relations), the following command inserts an item into the map based on a key: `$cname-insert(map$keyType,$target*::value_type( $keyName,$item))`

## Create

The `Create` property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector: `new vector$target*`

## CreateByValue

The `CreateByValue` property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`: `vector$target* $cname()`

## CreateStatic

The `CreateStatic` property defines the initialization of a relation when `CG::Relation::Containment` is set to `Reference`.

## CType

The `CType` property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the `vector$target*` collection type determines the type of the variable `cl` on the left side of the assignment:

```
vectorClient*>* cl = new vectorClient*>(2);
```

In this case, the class `Client` is the `$target` of vector operations.

The default is `$(constant)$target$reference`.

## Find

The `Find` property specifies the command used to locate an item in a container. For example, the following command calls the `Find()` operation for container `$cname` to locate the `$item`: `$cname-find($item)`

## FullTypeDefinition

The `FullTypeDefinition` property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about Composite Types.

The default is `$RelationTargetType $cname`.

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: `$me$cname` The variable `$me` is replaced with the object context variable as specified by the Me property. The variable `$cname` is replaced with the name of the container, which is the role name for the relation.

The default is `$cname`.

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The `ContainerTypes::RelationType::GetAt` property specifies a template for the body of the operation. For example, the following command generates code that call the `at()` operation for the container to retrieve the item at the indexed position: `$cname-at($index)`

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the `end()` operation for the container to locate the last item in the collection: `$cname-end()` This property and `GetEndGenerate` were created to adhere to the standard library convention for “finding” where iteration should end. `GetEnd` is generated where `Get` is generated. The method name is defined by using the `CG::Relation::GetEnd` and `CG::Relation::GetEndGenerate` properties.

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator `[]`, which has been overloaded according to the STL definition for maps: `$cname-operator[]($keyName)`

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file. (Default)

## **IncludeFiles**

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file.

A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

## **Init**

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: \$cname()

## **InitInCtorBody**

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: pos=0; pos\$multiplicity; pos++; \$cname[pos]=NULL

## **InitStatic**

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

## **IterCreate**

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: vector\$target\*::const\_iterator \$iterator; \$iterator=\$cname-begin()

## **IterCreateByValue**

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator` This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item: `$iterator++` (Default)

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection: `$iterator=$cname-begin()`

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

The default is `$(constRT)$target$reference`.

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname-end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<`, as defined in the STL. `vector$target*::const_iterator` You can change the iterator type to one of your own choice.



## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

The default value for C++ is as follows: `for (int pos = 0; pos $multiplicity; ++pos)` The default value for C is as follows: `int pos; for (pos = 0; pos $multiplicity; ++pos)` The default value for Java is as follows: `for (int pos = 0; pos $multiplicity; pos++)`

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$name. The default value for all other subjects is \$name.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

The default is \$CType.

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector$target*::iterator pos=find($cname-begin(), $cname-end(),$item);$cname-erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair$keyType,$target* p; p.second=$item; map$keyType,$target*::iterator pos=find($cname-begin(), $cname-end(),p); $cname-erase(pos)`

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items: `$cname-clear()`

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: \$cname-erase(\$keyName)

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved: \$cname = \$item The default is \$cname = \$item.

## Type

The Type property specifies the type of the container as a pointer to the relation.

## StaticArray

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed sequentially.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter: \$cname-push\_back(\$item)

For maps (qualified relations), the following command inserts an item into the map based on a key: \$cname-insert(map\$keyType,\$target\*::value\_type( \$keyName,\$item))

The default is as follows:

```
$Loop { if (!$cname[pos]) { $cname[pos] = $item; break; } }
```

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector: new vector\$target\*

## CreateByValue

The `CreateByValue` property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`: `vector$target* $cname()`

The default is `$CType`.

## CreateStatic

The `CreateStatic` property defines the initialization of a relation when `CG::Relation::Containment` is set to `Reference`.

## CType

The `CType` property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the `vector$target*` collection type determines the type of the variable `cl` on the left side of the assignment:

```
vectorClient*>* cl = new vectorClient*>(2);
```

In this case, the class `Client` is the `$target` of vector operations.

The default is `$RelationTargetType $cname[$multiplicity]`.

## Find

The `Find` property specifies the command used to locate an item in a container. For example, the following command calls the `Find()` operation for container `$cname` to locate the `$item`: `$cname-find($item)`

## FullTypeDefinition

The `FullTypeDefinition` property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about Composite Types.

The default is `$CType`.

## Get

The `Get` property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: `$me$cname` The variable `$me` is replaced with the object context variable as specified by the `Me` property. The variable `$cname` is replaced with the name of the container, which is the role name for the relation.

The default is \$cname.

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that call the at() operation for the container to retrieve the item at the indexed position:

```
$cname-at($index)
```

The default is \$cname[\$index].

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end() This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:  
\$cname-operator[](\$keyName)

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file.
- weak - The #include directives are added to the source file with forward declarations in the header file. (Default)

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file.

A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: \$cname()

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: pos=0; pos\$multiplicity; pos++; \$cname[pos]=NULL

The default is as follows:

```
$Loop { $cname[pos] = NULL; }
```

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container:

```
vector$target*::const_iterator $iterator; $iterator=$cname-begin()
```

The default is \$IterType \$iterator = 0;.

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

The default is \$IterCreate.

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: \*\$iterator This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

The default is \$cname[\$iterator].

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item: \$iterator++ (Default)

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

The default is \$IterIncrement.

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

The default is \$IterIncrement.

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation getRelation>() generated operation.

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection: \$iterator=\$cname-begin() The default is \$iterator = 0.

## IterReturnType

The `IterReturnType` property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

The default is `$IterType`.

## IterTest

The `IterTest` property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname-end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, `IterTest` retrieves the current item in the collection (the same as `IterGetCurrent`).

The default is `$iterator < $multiplicity`.

## IterType

The `IterType` property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<`, as defined in the STL. `vector$target*::const_iterator` You can change the iterator type to one of your own choice.

The default is `int`.

## Loop

The `Loop` property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the `OMContainers.StaticArray` metaclass) can use any other property in the same metaclass. Therefore, other properties that use `$Loop` expand the specified `Loop` property body.

The default value for C++ is as follows: `for (int pos = 0; pos $multiplicity; ++pos)` The default value for C is as follows: `int pos; for (pos = 0; pos $multiplicity; ++pos)` The default value for Java is as follows: `for (int pos = 0; pos $multiplicity; pos++)`

## Member

The `Member` property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for `RiCContainers::EmbeddedScalar` is `$me$cname`. The default value for all other subjects is `$cname`.

## RelationTargetType

The `RelationTargetType` property specifies the return type for relation getters, as part of the relation implementation properties.

The default is `$(constant)$target$reference`.

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the `find()` operation to point the iterator to the item to be removed and then call `erase()` to remove it. `vector$target*::iterator pos=find($cname-begin(), $cname-end(),$item);$cname-erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair `p`, whose second element, `p.second`, is the item to be removed. The `find()` operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair$keyType,$target* p; p.second=$item; map$keyType,$target*::iterator pos=find($cname-begin(), $cname-end(),p); $cname-erase(pos)`

The default is as follows:

```
$Loop { if ($cname[pos] == $item) { $cname[pos] = NULL; } }
```

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the `clear()` operation for the container to remove all items: `$cname-clear()`

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the `erase()` operation for the container, passing it the `$keyName`, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, `$cname`, is a role name of the relation itself, because there is only one class involved: `$cname = $item`

## SetAt

The SetAt property specifies how code is generated for the body of the mutator for a scalar container.

The default is `$cname[$index] = $item`.



## Type

The Type property specifies the type of the container as a pointer to the relation.

## UnboundedOrdered

Defines properties for implementing relations whose multiplicity is bounded and that are to be accessed sequentially.

### Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter: `$cname-push_back($item)`

For maps (qualified relations), the following command inserts an item into the map based on a key: `$cname-insert(map$keyType,$target*::value_type( $keyName,$item))`

The default is `$cname->push_back($item)`.

### Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector: `new vector$target* $cname` The default is `$cname = $CreateStatic`.

### CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`: `vector$target* $cname()` The default is `$CType $cname`

### CreateStatic

The CreateStatic property defines the initialization of a relation when `CG::Relation::Containment` is set to Reference.

The default is `new $CType`.

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector\$target\* collection type determines the type of the variable cl on the left side of the assignment:

```
vectorClient*>* cl = new vectorClient*>(2);
```

In this case, the class Client is the \$target of vector operations.

The default is std::vector<\$RelationTargetType>.

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item: \$cname-find(\$item)

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about Composite Types.

The default is \$CType \$cname.

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

The default is \$cname.

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that call the at() operation for the container to retrieve the item at the indexed position: \$cname-at(\$index) The default is \$cname->operator[](\$index).

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end() This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

The default is \$cname->end().

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps: \$cname-operator[](\$keyName)

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file. (Default)
- weak - The #include directives are added to the source file with forward declarations in the header file.

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

The default is <vector>,<iterator>.

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: \$cname() (Default)

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: pos=0; pos\$multiplicity; pos++; \$cname[pos]=NULL

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: vector\$target\*::const\_iterator \$iterator; \$iterator=\$cname-begin() The default is as follows: \$IterType \$iterator; \$IterReset

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

The default is \$IterCreate

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: \*\$iterator (Default) This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item: \$iterator++ (Default)

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the cleanUpRelations() method.

The default is `$IterReset`.

## IterIncrementForInit

The `IterIncrementForInit` property specifies the code that increments the iterator, for certain initialization cases.

The default is `$IterReset`.

## IterInit

The `IterInit` property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

## IterReset

The `IterReset` property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection: `$iterator=$cname-begin()` The default is `$iterator = $cname->begin()`.

## IterReturnType

The `IterReturnType` property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

The default is `$IterType`.

## IterTest

The `IterTest` property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: `$iterator != $cname-end()` With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, `IterTest` retrieves the current item in the collection (the same as `IterGetCurrent`).

The default is `$iterator != $cname->end()`.

## IterType

The `IterType` property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type `vector<`, as defined in the STL. `vector$target*::const_iterator` You can change the iterator type to one of your own choice.

The default is `$CType::const_iterator`

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

The default value for C++ is as follows: `for (int pos = 0; pos $multiplicity; ++pos)` The default value for C is as follows: `int pos; for (pos = 0; pos $multiplicity; ++pos)` The default value for Java is as follows: `for (int pos = 0; pos $multiplicity; pos++)`

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is `$me$name`. The default value for all other subjects is `$name`.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

The default is `$(constant)$target$reference`.

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the `find()` operation to point the iterator to the item to be removed and then call `erase()` to remove it. `vector$target*::iterator pos=find($name-begin(), $name-end(),$item);$name-erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair `p`, whose second element, `p.second`, is the item to be removed. The `find()` operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair$keyType,$target* p; p.second=$item; map$keyType,$target*::iterator pos=find($name-begin(), $name-end(),p); $name-erase(pos)` The default is as follows:

```
$CType::iterator pos = std::find($name->begin(), $name->end(),$item); if (pos != $name->end()) {  
$name->erase(pos); }
```

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the `clear()` operation for the container to remove all items:

`$cname-clear()` The default is `$cname->clear()`.

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the `erase()` operation for the container, passing it the `$keyName`, which maps into a dictionary used to locate the item to be removed:  
`$cname-erase($keyName)`

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, `$cname`, is a role name of the relation itself, because there is only one class involved: `$cname = $item`

## UnboundedUnordered

The UnboundedUnordered metaclass defines properties for implementing relations whose multiplicity is unbounded (\*) and that should be accessed randomly.

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the `push_back()` member function of the container class specified by `$cname`, and passes the item to be added as a formal parameter: `$cname-push_back($item)`

For maps (qualified relations), the following command inserts an item into the map based on a key:  
`$cname-insert(map$keyType,$target*::value_type( $keyName,$item))`

The default is `$cname->push_back($item)`.

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by `$target` and returns a pointer to the vector: `new vector$target*`

The default is `$cname = $CreateStatic`.

## CreateByValue

The `CreateByValue` property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by `$target` and assigns it the name stored in `$cname`: `vector$target* $cname()` The default is `$CType $cname`.

## CreateStatic

The `CreateStatic` property defines the initialization of a relation when `CG::Relation::Containment` is set to `Reference`.

The default is `new $CType`.

## CType

The `CType` property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the `vector$target*` collection type determines the type of the variable `cl` on the left side of the assignment:

```
vectorClient*>* cl = new vectorClient*>(2);
```

In this case, the class `Client` is the `$target` of vector operations.

The default is `std::list>$RelationTargetType>`.

## Find

The `Find` property specifies the command used to locate an item in a container. For example, the following command calls the `Find()` operation for container `$cname` to locate the `$item`: `$cname-find($item)`

## FullTypeDefinition

The `FullTypeDefinition` property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about Composite Types.

The default is `$CType $cname`.

## Get

The `Get` property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: `$me$cname` The variable `$me` is replaced with the object context variable as specified by the `Me` property. The variable `$cname` is replaced with the name of the container, which is the role name for the relation.



The default is \$cname.

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that call the at() operation for the container to retrieve the item at the indexed position:

```
$cname-at($index)
```

The default is \$cname->operator[](\$index).

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end() This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The method name is defined by using the CG::Relation::GetEnd and CG::Relation::GetEndGenerate properties.

The default is \$cname->end().

## GetKey

The GetKey property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator[], which has been overloaded according to the STL definition for maps:  
\$cname-operator[](\$keyName)

## IncludeDirective

The IncludeDirective property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- strong - The #include directives are added to the header file. (Default)
- weak - The #include directives are added to the source file with forward declarations in the header file.

## IncludeFiles

The IncludeFiles property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the Containment property is set to Reference, a forward declaration of the container is added to the class specification file, and the #include is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

The default is <list>,<iterator>.

## Init

The Init property specifies the command used to initialize the container. For example, the following command calls the constructor for the container \$cname to initialize it: \$cname() (Default)

## InitInCtorBody

The InitInCtorBody property specifies code to initialize containers for arrays inside the class constructor body. For example: pos=0; pos\$multiplicity; pos++; \$cname[pos]=NULL

## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: vector\$target\*::const\_iterator \$iterator; \$iterator=\$cname-begin()

The default is as follows:

```
$IterType $iterator; $IterReset
```

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

The default is \$IterCreate.

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator` This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

The default is `*$iterator`.

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the `$iterator` ahead one item: `$iterator++` (Default)

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

The default is `$IterReset`.

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

The default is `$IterReset`.

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the `begin()` operation for the iterator to point the iterator to the first item in the collection:

```
$iterator=$cname-begin()
```

The default is `$iterator = $cname->begin()`.

## IterReturnType

The IterReturnType property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

The default is \$IterType.

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname-end() With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

The default is \$iterator != \$cname->end().

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<, as defined in the STL. vector\$target\*::const\_iterator You can change the iterator type to one of your own choice.

The default is \$CType::const\_iterator.

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

The default value for C++ is as follows: for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$cname. The default value for all other subjects is \$cname.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

The default is \$(constant)\$target\$reference.

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector$target*::iterator pos=find($cname-begin(), $cname-end(),$item);$cname-erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair$keyType,$target* p; p.second=$item; map$keyType,$target*::iterator pos=find($cname-begin(), $cname-end(),p); $cname-erase(pos)` The default is as follows:

```
$CType::iterator pos = std::find($cname->begin(), $cname->end(),$item); if (pos != $cname->end()) {  
$cname->erase(pos); }
```

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items: `$cname-clear()` The default is `$cname->clear()`.

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved: `$cname = $item`

## Type

The Type property specifies the type of the container as a pointer to the relation.

## User

The User metaclass defines properties for user-defined implementations of relations.

You can create your own implementations for relations by defining a new set of properties under the User metaclass. Once these are defined, you can give them permanent status by manually saving them in the factory.prp file under any other name, for example MyFaves.

To complete their installation, you must add the new name as an enumerated value to the CG::Relation::Implementation property.

For example, you can change the definition of the Implementation property as follows: Subject CG Metaclass Relation Property Implementation Enum "Default,Scalar,Fixed,BoundedOrdered,BoundedUnordered,UnboundedOrdered, UnboundedUnordered,Qualified,MyFaves, User" "Default" end end

## Add

The Add property specifies the command used to add an item to a container.

For example, the following command calls the push\_back() member function of the container class specified by \$cname, and passes the item to be added as a formal parameter: \$cname-push\_back(\$item)

For maps (qualified relations), the following command inserts an item into the map based on a key: \$cname-insert(map\$keyType,\$target\*::value\_type( \$keyName,\$item))

## Create

The Create property specifies the command used to create a new container.

For example, the following command allocates space for a vector of role names represented by \$target and returns a pointer to the vector: new vector\$target\*

## CreateByValue

The CreateByValue property specifies the command used to create a new container by value.

For example, the following command instantiates a list of role names represented by \$target and assigns it the name stored in \$cname: vector\$target\* \$cname()

## CreateStatic

The CreateStatic property defines the initialization of a relation when CG::Relation::Containment is set to Reference.

## CType

The CType property specifies the collection type used to generate behaviors for relations. For example, when instantiating a vector that contains references to two clients, the vector\$target\* collection type determines the type of the variable cl on the left side of the assignment:

```
vectorClient*>* cl = new vectorClient*>(2);
```

In this case, the class Client is the \$target of vector operations.

## Find

The Find property specifies the command used to locate an item in a container. For example, the following command calls the Find() operation for container \$cname to locate the \$item: \$cname-find(\$item)

## FullTypeDefinition

The FullTypeDefinition property specifies the implementation template for a typedef composite type.

See the Rational Rhapsody help for information about Composite Types.

## Get

The Get property specifies a template for the code that is generated for the body of the accessor for a particular type of container.

For example, for a scalar relation generated in C, the body of the accessor is specified as: \$me\$cname The variable \$me is replaced with the object context variable as specified by the Me property. The variable \$cname is replaced with the name of the container, which is the role name for the relation.

## GetAt

The GetAt property specifies a template for the name of the operation generated to retrieve one of the targets of a to-many relation that uses an index.

The ContainerTypes::RelationType::GetAt property specifies a template for the body of the operation. For example, the following command generates code that call the at() operation for the container to retrieve the item at the indexed position: \$cname-at(\$index)

## GetEnd

The GetEnd property specifies the command used to retrieve the last item in the container. For example, the following command calls the end() operation for the container to locate the last item in the collection: \$cname-end() This property and GetEndGenerate were created to adhere to the standard library convention for “finding” where iteration should end. GetEnd is generated where Get is generated. The

method name is defined by using the `CG::Relation::GetEnd` and `CG::Relation::GetEndGenerate` properties.

## GetKey

The `GetKey` property specifies a template for the name of an operation generated to retrieve an item from a map (qualified relation) based on a key.

For example, the following command retrieves an item based on the key name by using the subscript operator `[]`, which has been overloaded according to the STL definition for maps:  
`$cname-operator[]($keyName)`

## IncludeDirective

The `IncludeDirective` property specifies how container header files are included in generated files. The property is read through a pointer to the concrete relation.

The possible values are as follows:

- `strong` - The `#include` directives are added to the header file. (Default)
- `weak` - The `#include` directives are added to the source file with forward declarations in the header file.

## IncludeFiles

The `IncludeFiles` property enables selective framework includes of templates based on a particular relation implementation.

If this property is defined, the specified include files are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the `Containment` property is set to `Reference`, a forward declaration of the container is added to the class specification file, and the `#include` is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

## Init

The `Init` property specifies the command used to initialize the container. For example, the following command calls the constructor for the container `$cname` to initialize it: `$cname()`

## InitInCtorBody

The `InitInCtorBody` property specifies code to initialize containers for arrays inside the class constructor body. For example: `pos=0; pos$multiplicity; pos++; $cname[pos]=NULL`



## InitStatic

The InitStatic property defines the initialization of a relation when CG::Relation::Containment is set to Value.

## IterCreate

The IterCreate property specifies the commands used to create an iterator for traversing the items in the container. For example, the following command instantiates the iterator class and points it to the first item in the container: `vector$target*::const_iterator $iterator; $iterator=$cname-begin()`

## IterCreateByValue

The IterCreateByValue property is the same as IterCreate, but instantiates the iterator by value.

## IterGetCurrent

The IterGetCurrent property specifies the command used to retrieve the current item in a container. For example, the following command returns a pointer to the current item in the collection: `*$iterator` This value is the same as that for IterTest only when you use the Rational Rhapsody framework OMContainers container set. When you use the STL container set, an operation substituted for IterGetCurrent returns a pointer to the current item in the collection.

## IterIncrement

The IterIncrement property specifies the code that increments the iterator. For example, the following command moves the \$iterator ahead one item: `$iterator++` (Default)

## IterIncrementForCleanUp

The IterIncrementForCleanUp property specifies code to increment the iterator when a relation is cleaned inside the `cleanUpRelations()` method.

## IterIncrementForInit

The IterIncrementForInit property specifies the code that increments the iterator, for certain initialization cases.

## IterInit

The IterInit property is used to generate code that initializes an iterator over a relation. The iterator is the return value of the relation `getRelation>()` generated operation.

*Default =*

## IterReset

The IterReset property specifies the command used to reset the iterator to the beginning. For example, the following command calls the begin() operation for the iterator to point the iterator to the first item in the collection: \$iterator=\$cname-begin()

## IterReturnTypes

The IterReturnTypes property specifies the return type for functions such as getters for the various containers that Rational Rhapsody uses.

## IterTest

The IterTest property specifies the command used to test whether the iterator is at the end of the set. For example, the following command returns a pointer to the last item in the collection: \$iterator != \$cname-end() With STLContainers, unlike OMContainers, it is possible to store a NULL value as the container. With OMContainers, IterTest retrieves the current item in the collection (the same as IterGetCurrent).

## IterType

The IterType property specifies the iterator type. For example, the following command denotes that the iterator in use is the parameterized type vector<, as defined in the STL. vector\$target\*::const\_iterator You can change the iterator type to one of your own choice.

## Loop

The Loop property reuses code that performs a loop. Beginning with Version 5.0, each property in a container metaclass (for example, the OMContainers.StaticArray metaclass) can use any other property in the same metaclass. Therefore, other properties that use \$Loop expand the specified Loop property body.

The default value for C++ is as follows: for (int pos = 0; pos \$multiplicity; ++pos) The default value for C is as follows: int pos; for (pos = 0; pos \$multiplicity; ++pos) The default value for Java is as follows: for (int pos = 0; pos \$multiplicity; pos++)

## Member

The Member property specifies the name of the embedded member in an embedded scalar (one-to-one) relation.

The default value for RiCContainers::EmbeddedScalar is \$me\$cname. The default value for all other subjects is \$cname.

## RelationTargetType

The RelationTargetType property specifies the return type for relation getters, as part of the relation implementation properties.

## Remove

The Remove property specifies the command used to remove an item from a relation.

For example, the following commands call the find() operation to point the iterator to the item to item to be removed and then call erase() to remove it. `vector$target*::iterator pos=find($cname-begin(), $cname-end(),$item);$cname-erase(pos)` This operation applies only to “to-many” (non-scalar) containers.

For maps (qualified relations), the following commands create a pair p, whose second element, p.second, is the item to be removed. The find() operation points the iterator to the position of the pair in the collection. Finally, the item at that position is erased. `pair$keyType,$target* p; p.second=$item; map$keyType,$target*::iterator pos=find($cname-begin(), $cname-end(),p); $cname-erase(pos)`

## RemoveAll

The RemoveAll property specifies the command used to remove all items from the container. For example, the following command calls the clear() operation for the container to remove all items: `$cname-clear()`

## RemoveKey

The RemoveKey property specifies the command that is generated to remove an item from a map (qualified relation) based on a key. For example, the following command calls the erase() operation for the container, passing it the \$keyName, which maps into a dictionary used to locate the item to be removed: `$cname-erase($keyName)`

## Set

The Set property specifies how code is generated for the body of the mutator for a scalar container.

For example, the following command says that the container name, \$cname, is a role name of the relation itself, because there is only one class involved: `$cname = $item`

## Type

The Type property specifies the type of the container as a pointer to the relation.

# TestConductor

---

The TestConductor subject contains metaclasses that contain properties that affect the TestConductor tool. This subject is available only if you have installed TestConductor.

## SDInstance

The SDInstance metaclass contains a number of properties that are used by TestConductor for internal data.

### ExecutionIterations

This property is used by TestConductor for internal data. It should not be changed by the user.

### ExecutionMode

This property is used by TestConductor for internal data. It should not be changed by the user.

### ExecutionOrder

This property is used by TestConductor for internal data. It should not be changed by the user.

### ParameterValues

This property is used by TestConductor for internal data. It should not be changed by the user.

## SequenceDiagram

The SequenceDiagram metaclass controls sequence diagram properties used by TestConductor.

### ActivationCondition

The ActivationCondition property specifies an activation condition. Activation conditions are used to specify the point in time during model execution when SD instances become activated. You can use activation conditions to model stubs or a predecessor order between several SD instances in a test definition. You can associate one activation condition with every SD. Activation conditions can specify a starting point of SD instance simulation, such as event sending or event receiving, which in turn can be a result of the behavior defined by another SD. TestConductor supports conditional expressions for events and conditions in the following form: `ObjectName->CondName(Parameters)` In this syntax:

- **ObjectName** is a parameterized or concrete name of a class instance or an environment variable that can be represented by the system border.
- **CondName** is a particular kind of event, state, or method action.
- **Parameters** is a state of a statechart, or the name of an event or method, and the receiver of this event or method, depending on the **CondName**.

Rational Rhapsody does not perform any static syntax checks on these conditions.

*Default = TRUE*

## Parameter

The **Parameter** property specifies the parameterized name used in a test definition. TestConductor supports test definitions based on SDs, whose instances either have concrete or parameterized names. A parameterized name is one that is not a valid (or concrete ) object name as usually used in Rational Rhapsody. You can also use anonymous class names, which do not have concrete names or parameters. In this case, the class name is internally expanded internally to the unique concrete object instance. During test execution, SDs are animated in relation to the default names. See the TestConductor Help for more information.

Default = Empty string

## Settings

The Settings metaclass contains several properties with options for TestConductor.

### AcknowledgeApplyChanges

If this property is checked, TestConductor prompts the user to acknowledge changes that have been made in the Edit SDInstances window. If cleared, changes that have been made in the window is accepted by TestConductor without the user being asked to acknowledge the changes.

This property is relevant only for animation based testing mode.

*Default = Checked*

### AddQuotesToCharAndStringArguments

The **AddQuotesToCharAndStringArguments** property controls the generation of driver and stub code for character and string types.

If the property is checked TestConductor automatically adds surrounding quotes to return values and arguments of character or string types when generating driver or stub code if the specified characters or strings are not already enclosed by quotes.

If the property is cleared TestConductor will not add surrounding quotes to return values and arguments of character or string types.

This property is a global option and should be modified only on the project.

*Default = Checked*

## **CreateTestArchitectureMode**

The CreateTestArchitectureMode property controls the behavior of the TestConductor function "Create TestArchitecture".

If the value of the property is set to Standard, then each time "Create TestArchitecture" is performed, TestConductor creates a component and a configuration for the newly created TestArchitecture by using the default settings for components and configurations.

If the value of the property is set to Advanced, then each time "Create TestArchitecture" is performed, TestConductor displays a window that allows you to select which of the existing components/configurations should be used as the basis for the property values of the new component/configuration that is to be created.

This property should not be modified on existing test architectures.

*Default = Standard*

## **CreateTestArchitectureTransparency**

The CreateTestArchitectureTransparency property controls the behavior of the TestConductor function "Create TestArchitecture".

If the value of the property is set to BlackBox, then each time "Create TestArchitecture" is performed, TestConductor creates a test architecture tailored for black box testing of the system under test using its public interface without validation of intrnal communication.

If the value of the property is set to GreyBox, then each time "Create TestArchitecture" is performed, TestConductor creates a test architecture tailored for grey box testing: The system under test is stimulated using its public interface but validation of internal communication is possible.

This property is relevant only for assertion based testing mode.

This property should not be modified on existing test architectures.

*Default = BlackBox*

## **CreateTestArchitectureUsingGlobalObjects**

The CreateTestArchitectureUsingGlobalObjects property controls the behavior of the TestConductor function "Create TestArchitecture".

If the property is checked, then each time "Create TestArchitecture" is performed, TestConductor creates a test architecture which allows generating driver or stub code in objects related to the system under test.

If the property is cleared, then each time "Create TestArchitecture" is performed, TestConductor creates a test architecture which does not support generating driver or stub code in objects related to the system under test.

This property is relevant only for assertion based testing mode.

This property should not be modified on existing test architectures.

*Default = Cleared*

## MapSDToTestArchitectureMode

The MapSDToTestArchitectureMode property controls the behavior of the test case wizard when a test case is created for an existing sequence diagram.

If the value of this property is set to "Strict", only those test architectures that contain at least one SUT instance of one of the classes of the life lines of the original sequence diagram are considered to be suitable for the new test case.

If the value of this property is set to "Weak", then also all test architectures for which the same message exchange is possible as in the original sequence diagram are considered to be suitable even if they do not contain a SUT instance of one of the classes of the life lines of the original sequence diagram.

*Default = Strict*

## OverwriteTestContextDiagram

The OverwriteTestContextDiagram property determines whether existing TestContextDiagrams are overwritten when performing an Update TestArchitecture on a TestContext. The property can take any of the following values:

- Never - each time Update TestArchitecture is performed, a new TestContextDiagram is added to the existing TestContextDiagrams, that is, existing TestContextDiagrams are never overwritten.
- askUser - each time Update TestArchitecture is performed, user is asked if an existing TestContextDiagram should be overwritten with the new one.
- Always - each time Update TestArchitecture is performed, existing TestContextDiagram is overwritten by the new one.

*Default = Never*

## ReplacementCreationMode

The ReplacementCreationMode property controls the behavior of the TestConductor function "Create TestArchitecture".

If the value of the property is set to **Wrapper**, then each time "Create TestArchitecture" is performed, TestConductor creates a test architecture with replacement test components which are wrapping the original behavior of the replaced class.

If the value of the property is set to **Stub**, then each time "Create TestArchitecture" is performed, TestConductor creates a test architecture with replacement test components which stubs only without any behavior of the replaced class.

This property is relevant only for assertion based testing mode.

*Default = Wrapper*

## ReportLocation

The ReportLocation property controls the location in the model where TestConductor creates test results.

Per default test results (test execution result or model, code or requirement coverage result) are created directly below the executed element (test package, test context or test case).

This property can be used to define a (test-)package where test results are created. The value of the property should contain a full path name of a (test-)package, using :: as delimiter. The package will be created if it does not exist.

These placeholders can be used: \$TESTPACKAGENAME, \$TESTCONTEXTNAME, \$TESTCASENAME, \$CONFIGURATIONNAME.

*Default = Empty string*

## TestCaseExecutionOrder

The TestCaseExecutionOrder property controls the execution order of TestCases when executing a TestContext. The possible values are:

- **BrowserOrder**- TestCases are executed in the same order as they are displayed in the browser
- **DeclarationOrder** - TestCases are executed in a user-defined order. The declaration order can be specified by right-clicking "TestCases" and selecting "Edit TestCases Order".

*Default = BrowserOrder*

## TestingMode

The TestingMode property controls the behavior of the TestConductor function "Create TestArchitecture".

If the value of the property is set to **AssertionBased**, then each time "Create TestArchitecture" is performed, TestConductor creates a test architecture tailored for assertion based testing.

If the value of the property is set to **AnimationBased**, then each time "Create TestArchitecture" is performed, TestConductor creates a test architecture tailored for animation based testing.



Assertion based testing mode is available only for C++ and C, animation based testing mode is available for Ada, C++, C, and Java.

This property should not be modified on existing test architectures.

*Default = AssertionBased (C++ or C); AnimationBased (Ada or Java)*

## TestCase

The TestCase metaclass contains properties that affect the behavior of TestConductor.

### AnimatedSUT

Depending on whether or not the SUT classes are animated, TestConductor uses different execution algorithms to control the execution of test cases. The property can take any of the following values:

- Automatic - TestConductor tries to deduce whether or not the SUT contains animation code, and chooses the appropriate execution algorithm.
- True - TestConductor chooses the appropriate algorithm on the assumption that the SUT classes contain animation code.
- False - TestConductor chooses the appropriate algorithm on the assumption that the SUT classes do not contain animation code.

This property is relevant only for animation based testing mode.

*Default = Automatic*

### ATGTestCase

This property is checked if the TestCase is generated by ATG, otherwise it is cleared.

### CallOperationsOnlyWhenCallstackEmpty

If this property is checked, TestConductor delays operation calls that refer to inputs of TestConductor so that these operation calls are made only when the call stack of the focus thread is empty.

If the property is cleared, all operation calls are made by TestConductor immediately even if the call stack of the focus thread is not empty.

This property is relevant only for animation based testing mode.

*Default = Cleared*

## ComputeCoverage

The ComputeCoverage property determines whether or not TestConductor automatically computes and reports the model coverage achieved when executing the test cases.

This property is relevant only for animation based testing mode. This property should not be used anymore, instead the corresponding tag ComputeModelCoverage of the code generation configuration should be used.

*Default = False*

## CoverageKind

If TestConductor::TestCase::ComputeCoverage is enabled, the CoverageKind property defines how the coverage will be measured. TestConductor supports four kinds of coverage measurement:

- SUT flat - Only coverage of the top level class of the SUT is measured. States, transitions, and operations of parts of the SUT are not considered. Coverage of model elements of test components is also not measured.
- SUT hierarchical - Coverage of the SUT is measured in a hierarchical manner. This means that also states, transitions, and operations of parts of the SUT are hierarchically regarded for coverage measure. Coverage of model elements of test components is not measured.
- TestContext flat - Coverage is measured in terms of all states, transitions, and operations defined at the first decomposition level of the test context, meaning that all states, transitions, and operations of the direct parts of the test context are considered.
- TestContext hierarchical - All states, transitions, and operations in the hierarchical structure of the test context are considered for coverage measurement.

This property is relevant only for animation based testing mode. This property should not be used anymore, instead the corresponding tag CoverageKind of the code generation configuration should be used.

*Default = SUT flat*

## CreateSDForFailedSDInstance

If this property is checked, then for each failed SDInstance of the TestCase, a color-coded sequence diagram showing the reason for failure is added to the model when TestCase execution has finished.

*Default = Cleared*

This property is relevant only for animation based testing mode.

## DriveMessagesToTestComponents

This property is used by TestConductor for internal data. It should not be changed by the user.

## EvaluateSDOperatorInArbiter

The EvaluateSDOperatorInArbiter property determines if the control expression of an SD operator should be evaluated in the context of the test arbiter or of the test context.

Prior to release 8.2, the control expression of an SD operator has been evaluated in arbiter of the test architecture. Since release 8.2, the control expression of an SD operator is evaluated in the test context, allowing simpler navigation expressions for example to attributes of the SUT in the control expression.

In order to preserve the previous behavior for pre-8.2 models, the property EvaluateSDOperatorInArbiter was added to the backward compatibility settings for C++ and , with a value of True.

## ExecuteTestWithTracer

If this property is checked, tracer outputs (trace #all all) are generated during TestCase execution.

This property is relevant only for animation based testing mode.

*Default = Cleared*

## ExecutionAnimationStartedTimeout

The ExecutionAnimationStartedTimeout property defines the time (in seconds) that TestConductor waits for the animated application to connect to Rhapsody. If the application does not connect to Rhapsody within the specified time, the test case execution is stopped.

This property is relevant only for animation based testing mode.

*Default = 20*

## ExecutionAnimationStoppedTimeout

The ExecutionAnimationStoppedTimeout property defines the time (in seconds) that TestConductor waits for the animated application to terminate after receiving the terminate command from TestConductor. If the application does not terminate within the specified time, TestConductor simply proceeds.

This property is relevant only for animation based testing mode.

*Default = 20*

## ExecutionFirstIdleTimeout

The ExecutionFirstIdleTimeout property defines the time (in seconds) that TestConductor waits for the animated application to become idle after giving the first “Go Idle” command. If the application does not become idle within the specified time, the test case execution is stopped.

This property is relevant only for animation based testing mode.

*Default = 20*

## **ExecutionIdleTimeout**

The ExecutionIdleTimeout property defines the amount of time (in seconds) that TestConductor will allow an application to show no activity before the test case is interrupted. If this property is set to zero, it means that no timeout is defined. The testing profile defines a global timeout which can be overwritten for every test package, test context, and test case.

This property is relevant only for animation based testing mode.

*Default = 600*

## **MultipleConditionCheck**

The MultipleConditionCheck property allows you to configure TestConductor to check the condition reached and following conditions, without system activity, until one condition mark evaluates to False. This is done by setting the value of this property to True.

This property is relevant only for animation based testing mode.

*Default = False*

## **ResetAppBeforeStartTest**

If this property is checked, TestConductor restarts the application each time a TestCase is executed. If the property is cleared, then if the application is already running, TestConductor executes the TestCase in the current execution state of the running application.

The property only affects sequence diagram-based TestCases. For code/flowchart/activity TestCases, TestConductor always restarts the application.

This property is relevant only for animation based testing mode.

*Default = Checked*

## **TerminateAppOnQuitTest**

If this property is checked, TestConductor terminates the application after TestCase execution has finished. If cleared, the application is not terminated after TestCase execution.

The property only affects sequence diagram-based TestCases. For code/flowchart/activity TestCases, TestConductor always terminates the application after TestCase execution has finished.

This property is relevant only for animation based testing mode.

*Default = Checked*

## Tolerances

This property is used by TestConductor for internal data. It should not be changed by the user.

## UseOM\_RETURN

The UseOM\_RETURN property is used to determine how a return value is to be checked.

The property should be set to True for operations that use the animation macro OM\_RETURN.

For operations that do not use the OM\_RETURN macro, the value of the property should be set to False. Note that in such cases, TestConductor can only check return values for operation calls that originate from TestComponents.

This property is relevant only for animation based testing mode.

*Default = False*

## WriteTestExecutionLogFile

If this property is checked, TestConductor creates an execution log file called "C:/tmp/rtc.log". During TestCase execution, TestConductor writes log messages to this file, which can be used for purposes such as debugging.

If the machine running TestConductor does not have a directory called "C:/tmp", no log file is created.

This property is relevant only for animation based testing mode.

*Default = Cleared*

## TestContext

The TestContext metaclass contains properties that affect the behavior of TestConductor.

## CreateSDTestCaseWithParts

The CreateSDTestCaseWithParts property controls the behavior of the TestConductor function "Create SD TestCase".

If the property is checked, then each time "Create SD TestCase" is performed, TestConductor creates a test scenario containing individual SUT life lines for SUT instances and their parts (one nesting level).

If the property is cleared, then each time "Create SD TestCase" is performed, TestConductor creates a test scenario containing SUT life lines only for direct SUT instances of the test architecture.

*Default = Cleared*

## **TestContextExecution\_PostTestCaseOperation**

The TestContextExecution\_PostTestCaseOperation property can be used to define an operation of the test context which will be called at the end of the execution of each test case.

*Default = Empty string*

## **TestContextExecution\_PreTestCaseOperation**

The TestContextExecution\_PreTestCaseOperation property can be used to define an operation of the test context which will be called at the beginning of the execution of each test case.

*Default = Empty string*

## **TestContextExecution\_RestartExecutable**

The TestContextExecution\_RestartExecutable property controls if the tested application is restarted for each test case when executing a test context or test package.

If the property is checked the tested application will be restarted for each test case when executing a test context or test package.

If the property is not checked, when executing a test context or test package each time a test case has finished the next test case will be activated without restarting the tested application.

*Default = Checked*

# TimingDiagram

---

Contains properties for Timing diagrams.

## CancelledTimeout

Contains properties relating to canceled timeouts.

### DefaultName

When you add a new model element, it is assigned a default name based on the type of element, for example, class\_0 or class\_1. The same type of generic approach is used when you create elements based on a new term. So if your model contains a "new term" stereotype called SystemConfiguration, when you add new elements of this type, they will have names such as systemconfiguration\_0 or systemconfiguration\_1.

For "new terms", you have an option to change this default naming scheme. You can provide a value for the property DefaultName (at the level of the relevant stereotype) to provide a different string that should be used as the basis for the default name when you create an element of that type. So, you can call your "new term" stereotype SystemConfiguration but then assign a value such as SC to the DefaultName property for that stereotype. Then, each new element of that type that you create will have a default name such as sc\_0 or sc\_1.

*Default = CanTm*

## Comment

The Comment metaclass contains properties relating to the display of Comment elements in timing diagrams.

### CommentNotation

The CommentNotation property determines how Comment elements are displayed. This property can be set to one of two styles:

- Note\_Style
- Box\_Style

If CommentNotation is set to Note\_Style, then the appearance of the element is determined by the value selected for the property Comment::ShowForm: Note, Plain, or PushPin.

If CommentNotation is set to Box\_Style, then the element resembles a class-box, and you can control the

display of compartments for the element.

Default = Note\_Style

## Compartments

The Compartments property determines which of the available compartments are displayed by default for the various types of elements. The value for this property is a comma-delimited string containing the names of the compartments that should be visible.

Since the available compartments vary from element to element, it is recommended that you do not try to set the value of this property by using the Properties window or directly in the .prp file. Rather, you should use an element's Display Options to set which compartments are visible, and then use the Make Default option to apply these settings at the diagram or project level for new elements of this type.

*Default = Blank*

## QuickNavigationCategories

Graphic elements in diagrams can include a list of shortcuts that are accessible from the title bar of the element. You can use the property QuickNavigationCategories to specify the categories of items that should be included in this shortcut list and the order in which the categories should appear in the list. The value of the property should be a comma-separated list of categories. The categories that can be included are:

- Activities
- DiagramReferences
- Diagrams
- Hyperlinks
- InternalBlockDiagrams
- MainBehavior
- MainDiagram
- References
- StateCharts

If you want greater control over the content of the navigation list (such as showing/hiding specific elements within a given category, or changing the order of elements within a specific category), use the Compartments list, which can be found under "Display Options".

*Default = Hyperlinks*

## ShowAnnotationContents

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the text displayed on the element in the diagram determined by the value that was set for the property ShowAnnotationContents. The property can take one of the following values:



- Name - the name of the element
- Label - the label assigned to the element
- Description - the content of the Description field
- Specification (for Requirements only) - the content of the Specification field

*Default = Description*

## ShowForm

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the appearance of the element is determined by the value that was set for the property ShowForm.

The possible values for this property vary for the different elements, as does the default value used. The values used are:

- Plain - No color background behind text
- Note - Color background behind text
- Pushpin - Color background plus pin icon

*Default = Note*

## ShowName

The ShowName property determines the text to display next to a graphic element in a diagram. For most elements, you can provide a name and a label. Your descriptive label can be useful in cases where the name itself might not be sufficient due to various constraints, such as the inability to use spaces if the name of the element is to appear in the code.

The possible values for this property vary for the different elements, as does the default value used. The possible values are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is displayed with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

Note that for some elements, such as destruction elements in sequence diagrams and timing diagrams, the property is a boolean property that hides/shows the name of the element.

*Default = Name\_only*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## Constraint

The Constraint metaclass contains properties relating to the display of Constraint elements in timing diagrams.

## Compartments

The Compartments property determines which of the available compartments are displayed by default for the various types of elements. The value for this property is a comma-delimited string containing the names of the compartments that should be visible.

Since the available compartments vary from element to element, it is recommended that you do not try to set the value of this property by using the Properties window or directly in the .prp file. Rather, you should use an element's Display Options to set which compartments are visible, and then use the Make Default option to apply these settings at the diagram or project level for new elements of this type.

*Default = Specification*

## ConstraintNotation

The ConstraintNotation property determines how Constraint elements are displayed. This property can be set to one of two styles:

- Note\_Style
- Box\_Style

If ConstraintNotation is set to Note\_Style, then the appearance of the element is determined by the value selected for the property Constraint::ShowForm: Note, Plain, or PushPin.

If ConstraintNotation is set to Box\_Style, then the element resembles a class-box, and you can control the display of compartments for the element.

*Default = Note\_Style*

## QuickNavigationCategories

Graphic elements in diagrams can include a list of shortcuts that are accessible from the title bar of the element. You can use the property QuickNavigationCategories to specify the categories of items that should be included in this shortcut list and the order in which the categories should appear in the list. The value of the property should be a comma-separated list of categories. The categories that can be included are:

- Activities
- DiagramReferences
- Diagrams
- Hyperlinks
- InternalBlockDiagrams
- MainBehavior
- MainDiagram
- References
- StateCharts

If you want greater control over the content of the navigation list (such as showing/hiding specific elements within a given category, or changing the order of elements within a specific category), use the Compartments list, which can be found under "Display Options".

*Default = Hyperlinks*

## ShowAnnotationContents

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the text displayed on the element in the diagram determined by the value that was set for the property ShowAnnotationContents. The property can take one of the following values:

- Name - the name of the element
- Label - the label assigned to the element
- Description - the content of the Description field
- Specification (for Requirements only) - the content of the Specification field

*Default = Description*

## ShowForm

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the appearance of the element is determined by the value that was set for the property ShowForm.

The possible values for this property vary for the different elements, as does the default value used. The

values used are:

- Plain - No color background behind text
- Note - Color background behind text
- Pushpin - Color background plus pin icon

*Default = Note*

## ShowName

The ShowName property determines the text to display next to a graphic element in a diagram. For most elements, you can provide a name and a label. Your descriptive label can be useful in cases where the name itself might not be sufficient due to various constraints, such as the inability to use spaces if the name of the element is to appear in the code.

The possible values for this property vary for the different elements, as does the default value used. The possible values are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is displayed with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

Note that for some elements, such as destruction elements in sequence diagrams and timing diagrams, the property is a boolean property that hides/shows the name of the element.

*Default = Name\_only*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## CreateMessage

Contains properties relating to CreateMessage elements.

### DefaultName

When you add a new model element, it is assigned a default name based on the type of element, for example, class\_0 or class\_1. The same type of generic approach is used when you create elements based on a new term. So if your model contains a "new term" stereotype called SystemConfiguration, when you add new elements of this type, they will have names such as systemconfiguration\_0 or systemconfiguration\_1.

For "new terms", you have an option to change this default naming scheme. You can provide a value for the property DefaultName (at the level of the relevant stereotype) to provide a different string that should be used as the basis for the default name when you create an element of that type. So, you can call your "new term" stereotype SystemConfiguration but then assign a value such as SC to the DefaultName property for that stereotype. Then, each new element of that type that you create will have a default name such as sc\_0 or sc\_1.

*Default = createmessage\_\$Index*

## Depends

The Depends metaclass contains properties relating to the display of Dependency lines in timing diagrams.

### line\_style

The line\_style property specifies the default line style for a graphical item.

The possible values are:

- straight\_arrows - straight line.
- rectilinear\_arrows - rectilinear line with right-angled corners placed at appropriate locations, depending on the starting and ending points of the line
- spline\_arrows - curved line without corners
- rounded\_rectilinear\_arrows - similar to ordinary rectilinear lines but use rounded bends rather than straight ninety-degree angles

*Default = spline\_arrows*

*Default = straight\_arrows*

## ShowName

The ShowName property determines the text to display next to a graphic element in a diagram. For most elements, you can provide a name and a label. Your descriptive label can be useful in cases where the name itself might not be sufficient due to various constraints, such as the inability to use spaces if the name of the element is to appear in the code.

The possible values for this property vary for the different elements, as does the default value used. The possible values are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is displayed with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

Note that for some elements, such as destruction elements in sequence diagrams and timing diagrams, the property is a boolean property that hides/shows the name of the element.

*Default = None*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## DestroyMessage

Contains properties relating to DestroyMessage elements.

## DefaultName

When you add a new model element, it is assigned a default name based on the type of element, for example, class\_0 or class\_1. The same type of generic approach is used when you create elements based on a new term. So if your model contains a "new term" stereotype called SystemConfiguration, when you add new elements of this type, they will have names such as systemconfiguration\_0 or systemconfiguration\_1.

For "new terms", you have an option to change this default naming scheme. You can provide a value for the property DefaultName (at the level of the relevant stereotype) to provide a different string that should be used as the basis for the default name when you create an element of that type. So, you can call your "new term" stereotype SystemConfiguration but then assign a value such as SC to the DefaultName property for that stereotype. Then, each new element of that type that you create will have a default name such as sc\_0 or sc\_1.

*Default = destroymessage\_ \$Index*

## DestructionEvent

Contains properties that control the display of destruction events in timing diagrams.

## ShowName

The ShowName property determines the text to display next to a graphic element in a diagram. For most elements, you can provide a name and a label. Your descriptive label can be useful in cases where the name itself might not be sufficient due to various constraints, such as the inability to use spaces if the name of the element is to appear in the code.

The possible values for this property vary for the different elements, as does the default value used. The possible values are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is displayed with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

Note that for some elements, such as destruction elements in sequence diagrams and timing diagrams, the

property is a boolean property that hides/shows the name of the element.

*Default = True*

## DiagramFrame

The DiagramFrame metaclass contains properties that control the display of a diagram frame.

### DefaultSize

The DefaultSize property specifies default values for diagram frame dimensions.

*Default = 20, 20, 590, 500*

### ShowName

The ShowName property determines the text to display next to a graphic element in a diagram. For most elements, you can provide a name and a label. Your descriptive label can be useful in cases where the name itself might not be sufficient due to various constraints, such as the inability to use spaces if the name of the element is to appear in the code.

The possible values for this property vary for the different elements, as does the default value used. The possible values are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is displayed with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

Note that for some elements, such as destruction elements in sequence diagrams and timing diagrams, the property is a boolean property that hides/shows the name of the element.

*Default = Name*

### ShowStereotype



The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## EventMessage

Contains properties relating to EventMessage elements.

### DefaultName

When you add a new model element, it is assigned a default name based on the type of element, for example, class\_0 or class\_1. The same type of generic approach is used when you create elements based on a new term. So if your model contains a "new term" stereotype called SystemConfiguration, when you add new elements of this type, they will have names such as systemconfiguration\_0 or systemconfiguration\_1.

For "new terms", you have an option to change this default naming scheme. You can provide a value for the property DefaultName (at the level of the relevant stereotype) to provide a different string that should be used as the basis for the default name when you create an element of that type. So, you can call your "new term" stereotype SystemConfiguration but then assign a value such as SC to the DefaultName property for that stereotype. Then, each new element of that type that you create will have a default name such as sc\_0 or sc\_1.

*Default = eventmessage\_\$Index*

## General

The General metaclass contains properties that specify the general behavior of a timing diagram.

### DefaultDiagramForm

When you create a new timing diagram, the New Diagram dialog allows you to select whether the diagram should be a compact timing diagram or an elaborated timing diagram.

You can use the property DefaultDiagramForm to specify which of these two options - compact or elaborated - should be selected by default when the dialog is displayed.

*Default = Compact*

## FillColor

The Fillcolor property specifies the default fill color for an object.

*Default = RGB (243, 243, 243)*

## ShowInvariantLines

In elaborated timing diagrams, you have the option of displaying horizontal lines as visual guides for each of the invariants added to a lifeline.

You can hide/show these lines for a specific diagram by selecting the relevant option in the popup menu for timing diagrams.

If you want to change the default behavior for elaborated timing diagrams, you can modify the value of the property ShowInvariantLines.

*Default = True*

## ShowSequenceNumbers

The ShowSequenceNumber property determines if sequence numbers are shown.

*Default = Cleared*

## SysMLDiagramKindTitle

When a diagram is created in SysML projects, the diagram frame contains a header for the diagram that includes information such as the name of the package that contains the diagram. The property SysMLDiagramKindTitle is used to specify a prefix for the diagram frame header that reflects the type of diagram. For example, when you create an internal block diagram, the header in the diagram frame starts with the prefix "ibd".

Note that the structure of the entire header is controlled by the property General::Graphics::DiagramHeader.

## Message

The Message metaclass contains properties that control the display of messages in a timing diagram.

## DefaultName

When you add a new model element, it is assigned a default name based on the type of element, for example, class\_0 or class\_1. The same type of generic approach is used when you create elements based on a new term. So if your model contains a "new term" stereotype called SystemConfiguration, when you add new elements of this type, they will have names such as systemconfiguration\_0 or systemconfiguration\_1.

For "new terms", you have an option to change this default naming scheme. You can provide a value for the property DefaultName (at the level of the relevant stereotype) to provide a different string that should be used as the basis for the default name when you create an element of that type. So, you can call your "new term" stereotype SystemConfiguration but then assign a value such as SC to the DefaultName property for that stereotype. Then, each new element of that type that you create will have a default name such as sc\_0 or sc\_1.

*Default = message\_ \$Index*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Cleared*

## ShowStereotypeOfOperation

The ShowStereotypeOfOperation is a Boolean property that determines if a message displays its realized operation stereotype

*Default = Cleared*

## Note

Contains properties relating to the display of Note elements.

## ShowForm

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the appearance of the element is determined by the value that was set for the property ShowForm.

The possible values for this property vary for the different elements, as does the default value used. The values used are:

- Plain - No color background behind text
- Note - Color background behind text
- Pushpin - Color background plus pin icon

*Default = Note*

## ReplyMessage

Contains properties relating to ReplyMessage elements.

### DefaultName

When you add a new model element, it is assigned a default name based on the type of element, for example, class\_0 or class\_1. The same type of generic approach is used when you create elements based on a new term. So if your model contains a "new term" stereotype called SystemConfiguration, when you add new elements of this type, they will have names such as systemconfiguration\_0 or systemconfiguration\_1.

For "new terms", you have an option to change this default naming scheme. You can provide a value for the property DefaultName (at the level of the relevant stereotype) to provide a different string that should be used as the basis for the default name when you create an element of that type. So, you can call your "new term" stereotype SystemConfiguration but then assign a value such as SC to the DefaultName property for that stereotype. Then, each new element of that type that you create will have a default name such as sc\_0 or sc\_1.

*Default = replymessage\_\$Index*

## Requirement

The Requirement metaclass contains properties relating to the display of Requirement elements in timing diagrams.

### Compartment

The Compartments property determines which of the available compartments are displayed by default for the various types of elements. The value for this property is a comma-delimited string containing the names of the compartments that should be visible.

Since the available compartments vary from element to element, it is recommended that you do not try to set the value of this property by using the Properties window or directly in the .prp file. Rather, you should use an element's Display Options to set which compartments are visible, and then use the Make Default option to apply these settings at the diagram or project level for new elements of this type.

*Default = ID, Specification*

## QuickNavigationCategories

Graphic elements in diagrams can include a list of shortcuts that are accessible from the title bar of the element. You can use the property QuickNavigationCategories to specify the categories of items that should be included in this shortcut list and the order in which the categories should appear in the list. The value of the property should be a comma-separated list of categories. The categories that can be included are:

- Activities
- DiagramReferences
- Diagrams
- Hyperlinks
- InternalBlockDiagrams
- MainBehavior
- MainDiagram
- References
- StateCharts

If you want greater control over the content of the navigation list (such as showing/hiding specific elements within a given category, or changing the order of elements within a specific category), use the Compartments list, which can be found under "Display Options".

*Default = Hyperlinks*

## RequirementNotation

The RequirementNotation property determines how Requirement elements are displayed. This property can be set to one of two styles:

- Note\_Style
- Box\_Style

If RequirementNotation is set to Note\_Style, then the appearance of the element is determined by the value selected for the property Requirement::ShowForm: Note, Plain, or PushPin.

If RequirementNotation is set to Box\_Style, then the element resembles a class-box, and you can control the display of compartments for the element.

*Default = Note\_Style*

## ShowAnnotationContents

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the text displayed on the element in the diagram determined by the value that was set for the property

ShowAnnotationContents. The property can take one of the following values:

- Name - the name of the element
- Label - the label assigned to the element
- Description - the content of the Description field
- Specification (for Requirements only) - the content of the Specification field

*Default = Description*

## ShowForm

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the appearance of the element is determined by the value that was set for the property ShowForm.

The possible values for this property vary for the different elements, as does the default value used. The values used are:

- Plain - No color background behind text
- Note - Color background behind text
- Pushpin - Color background plus pin icon

*Default = Note*

## ShowName

The ShowName property determines the text to display next to a graphic element in a diagram. For most elements, you can provide a name and a label. Your descriptive label can be useful in cases where the name itself might not be sufficient due to various constraints, such as the inability to use spaces if the name of the element is to appear in the code.

The possible values for this property vary for the different elements, as does the default value used. The possible values are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is displayed with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

Note that for some elements, such as destruction elements in sequence diagrams and timing diagrams, the

property is a boolean property that hides/shows the name of the element.

*Default = Name\_only*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## SpecificationAnnotationFormat

When a requirement is displayed on a diagram, you can choose to have the element show one of the following: its name, its label, its description, the specification text for the requirement.

If you choose to display the specification text, you can use the property SpecificationAnnotationFormat to customize the text that is displayed.

The following keywords can be included in the value of the property: \$Specification (to include the specification text) and \$ID (to include the ID that was specified in the Features dialog for the requirement).

Note that this property affects the displayed text only when a requirement uses the "Note" notation style. When using the "Box" notation style, only the specification text is displayed, regardless of the value of this property.

*Default = \$Specification*

## TimeAxis

The TimeAxis metaclass contains properties that control the display of a time axis in a timing diagram.

### Font

Determines the font settings used for timing diagram text

Default: Tahoma, 8, NoBold, NoItalic

## IncrementValue

The value by which a unit of time increases on an axis.

Default: 1

## StartValue

The starting value (expressed as an integer) of time for an axis

Default: 0

## TrackMouseMove

This property determines if mouse movement tracking is shown or hidden on a timing axis.

Default: Enabled

## UnitLabel

The UnitLabel is a string value indicating the unit of time shown on the axis.

Default: Seconds

## Timeout

Contains properties relating to timeouts.

## DefaultName

When you add a new model element, it is assigned a default name based on the type of element, for example, class\_0 or class\_1. The same type of generic approach is used when you create elements based on a new term. So if your model contains a "new term" stereotype called SystemConfiguration, when you add new elements of this type, they will have names such as systemconfiguration\_0 or systemconfiguration\_1.

For "new terms", you have an option to change this default naming scheme. You can provide a value for the property DefaultName (at the level of the relevant stereotype) to provide a different string that should be used as the basis for the default name when you create an element of that type. So, you can call your "new term" stereotype SystemConfiguration but then assign a value such as SC to the DefaultName property for that stereotype. Then, each new element of that type that you create will have a default name such as sc\_0 or sc\_1.



*Default = Tm*

## TimingInstanceLine

Contains properties that control the display of an instance line in a timing diagram.

### ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = False*

# UseCaseExtensions

---

The UseCaseExtensions subject contains properties that determine extensions to use cases, as described in version 1.4 of the UML standard. Currently, these properties are informative only - they do not affect the implementation of the model.

## Dependency

The Dependency metaclass contains properties that control the extensions to use case dependencies, as defined in version 1.4 of the UML standard.

## Condition

The Condition property specifies the condition applied to the extend relationship between use cases. If the condition is met, the extension is applied.

*Default = Empty string*

## ExtensionPoint

The ExtensionPoint property specifies the extension point that is relevant for the relationship. This should correspond to one of the extension points defined for the use case (specified in the Use Case Features window).

*Default = Empty string*

# UseCaseGe

---

The UseCaseGe subject contains metaclasses that contain properties that determine the default appearance of elements in use case diagrams.

## Actor

The Actor metaclass contains properties that control the appearance of actors in use case diagrams.

### ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values for this property varies for the different elements, as does the default value used. The different values used are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is opened with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

*Default = Relative*

### ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

# Association

The Association metaclass contains properties that control the appearance of association lines in use case diagrams.

## color

*The color property specifies the default color of the border of a graphical item, such as an object box.  
(Default = 255,0,0)*

## DefaultLabel

The DefaultLabel property is used to specify the text area that should be highlighted for editing when you draw an association. The options available are:

- the target role
- the source role
- the target multiplicity
- the source multiplicity
- the name of the association (default behavior prior to release 8.1.4)

*Default = Target\_Role*

## line\_style

The line\_style property specifies the default line style for a graphical item.

The possible values are:

- straight\_arrows - straight line.
- rectilinear\_arrows - rectilinear line with right-angled corners placed at appropriate locations, depending on the starting and ending points of the line
- spline\_arrows - curved line without corners
- rounded\_rectilinear\_arrows - similar to ordinary rectilinear lines but use rounded bends rather than straight ninety-degree angles

*Default = spline\_arrows*

*Default = straight\_arrows*

## ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values for this property varies for the different elements, as does the default value used. The different values used are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is opened with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

*Default = Name*

## ShowRoleVisibility

For associations and links, if you have chosen to display the names of the association ends, you can use the property ShowRoleVisibility to specify that alongside the name there should be an indication of the visibility of the member that represents the association end.

When this property is set to True, the following symbols are displayed to reflect the visibility:

- public +
- protected #
- private -
- static \$

For individual associations or links, you can turn on the display of member visibility information by opening the Display Options dialog and selecting the Visibility check box.

*Default = False*

## ShowSourceMultiplicity

An association, link, aggregation, and composition have two ends and each end might be assigned a multiplicity number (1, 1.x, and so on). This property controls whether the assigned multiplicity number is visible (selected check box) or not (cleared check box) on the "source" end of the line.

*Default = Cleared*

## ShowSourceQualifier

The ShowSourceQualifier property is used for a number of connector elements, such as Associations. When set to Checked, the source element attribute defined as a qualifier for the association is displayed alongside the connector, for example attribute\_0. The property can be set at the diagram level. The Display Options window can be used to change the qualifier show/hide setting for an individual connector. However, it does not change the value of the property at the diagram level.

*Default = Cleared*

## ShowSourceRole

The ShowSourceRole property is used for a variety of connector elements, such as Links and Associations. When set to Checked, the source end of the relationship is displayed alongside the connector, for example itsClass\_1. The property is set at the diagram level.

*Default = Cleared*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = None*

## ShowTargetMultiplicity

An association, link, aggregation, and composition have two ends and each end might be assigned a multiplicity number (1, 1.x, and so on). This property controls whether the assigned multiplicity number is visible (selected check box) or not (cleared check box) on the "target" end of the line.

*Default = Cleared*

## ShowTargetQualifier

The ShowTargetQualifier property is used for a number of connector elements, such as Associations. When set to Checked, the target element attribute defined as a qualifier for the association is displayed alongside the connector, for example attribute\_1. The property can be set at the diagram level. The Display Options window can be used to change the qualifier show/hide setting for an individual connector. However, it does not change the value of the property at the diagram level.

*Default = Cleared*

## ShowTargetRole

The ShowTargetRole property is used for a variety of connector elements, such as Links and Associations. When set to Checked, the target end of the relationship is displayed alongside the connector, for example itsClass\_2. The property is set at the diagram level.

*Default = Cleared*

## AutoPopulate

The AutoPopulate metaclass contains properties that can be used to control the appearance of diagrams that are drawn automatically by Rational Rhapsody.

## ArrowDirection

The ArrowDirection property is used when Rational Rhapsody automatically generates a diagram, and it determines whether the flow of connectors in the diagram runs from top to bottom or bottom to top.

There are two situations where Rational Rhapsody automatically generates diagrams:

- If you have selected the Populate Diagrams option for Reverse Engineering (for those diagrams where this feature is supported).
- If you double-click a diagram in the browser that was generated by using the Rational Rhapsody API.

*Default = Bottom-Top*

## Comment

The Comment metaclass contains properties that control the appearance of comments in use case diagrams.

## CommentNotation

The CommentNotation property determines how Comment elements are displayed. This property can be set to one of two styles:

- Note\_Style
- Box\_Style

If CommentNotation is set to Note\_Style, then the appearance of the element is determined by the value selected for the property Comment::ShowForm: Note, Plain, or PushPin.

If CommentNotation is set to Box\_Style, then the element resembles a class-box, and you can control the display of compartments for the element.

*Default = Note\_Style*

## Compartments

The Compartments property determines which of the available compartments are displayed by default for the various types of elements. The value for this property is a comma-delimited string containing the names of the compartments that should be visible. Since the available compartments vary from element to element, it is recommended that you do not try to set the value of this property by using the Properties window or directly in the .prp file. Rather, you should use an element Display Options to set which compartments are visible, and then use the Make Default option to apply these settings at the diagram or project level for new elements of this type.

*Default = Empty MultiLine*

## QuickNavigationCategories

Graphic elements in diagrams can include a list of shortcuts that are accessible from the title bar of the element. You can use the property QuickNavigationCategories to specify the categories of items that should be included in this shortcut list and the order in which the categories should appear in the list. The value of the property should be a comma-separated list of categories. The categories that can be included are:

- Activities
- DiagramReferences
- Diagrams
- Hyperlinks
- InternalBlockDiagrams
- MainBehavior
- MainDiagram
- References
- StateCharts

If you want greater control over the content of the navigation list (such as showing/hiding specific elements within a given category, or changing the order of elements within a specific category), use the Compartments list, which can be found under "Display Options".

*Default = Hyperlinks*

## ShowAnnotationContents

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the text displayed on the element in the diagram determined by the value that was set for the property ShowAnnotationContents. The property can take one of the following values:



- Name - the name of the element
- Label - the label assigned to the element
- Description - the content of the Description field
- Specification (for Requirements only) - the content of the Specification field

*Default = Description*

## ShowForm

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the appearance of the element is determined by the value that was set for the property ShowForm.

The possible values for this property vary for the different elements, as does the default value used. The values used are:

- Plain - No color background behind text
- Note - Color background behind text
- Pushpin - Color background plus pin icon

*Default = Note*

## ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values for this property varies for the different elements, as does the default value used. The different values used are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is opened with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

*Default = Relative*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## Complete

The metaclass Complete contains properties that determine whether or not Rational Rhapsody automatically draws the relations that exist between an element added to a diagram and elements already on the diagram.

### Complete\_Relation

The Complete\_Relation property is used to specify that when an element is added to a diagram, Rational Rhapsody should automatically draw the relations that exist between the element and elements already on the diagram.

*Default = Cleared*

## Constraint

The Constraint metaclass contains properties that control the constraints in use case diagrams.

### Compartments

The Compartments property determines which of the available compartments are displayed by default for the various types of elements. The value for this property is a comma-delimited string containing the names of the compartments that should be visible. Since the available compartments vary from element to element, it is recommended that you do not try to set the value of this property by using the Properties window or directly in the .prp file. Rather, you should use an element Display Options to set which compartments are visible, and then use the Make Default option to apply these settings at the diagram or project level for new elements of this type.

*Default = Specification*

### ConstraintNotation

The ConstraintNotation property determines how Constraint elements are displayed. This property can be

set to one of two styles:

- Note\_Style
- Box\_Style

If ConstraintNotation is set to Note\_Style, then the appearance of the element is determined by the value selected for the property Constraint::ShowForm: Note, Plain, or PushPin.

If ConstraintNotation is set to Box\_Style, then the element resembles a class-box, and you can control the display of compartments for the element.

*Default = Note\_Style*

## QuickNavigationCategories

Graphic elements in diagrams can include a list of shortcuts that are accessible from the title bar of the element. You can use the property QuickNavigationCategories to specify the categories of items that should be included in this shortcut list and the order in which the categories should appear in the list. The value of the property should be a comma-separated list of categories. The categories that can be included are:

- Activities
- DiagramReferences
- Diagrams
- Hyperlinks
- InternalBlockDiagrams
- MainBehavior
- MainDiagram
- References
- StateCharts

If you want greater control over the content of the navigation list (such as showing/hiding specific elements within a given category, or changing the order of elements within a specific category), use the Compartments list, which can be found under "Display Options".

*Default = Hyperlinks*

## ShowAnnotationContents

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the text displayed on the element in the diagram determined by the value that was set for the property ShowAnnotationContents. The property can take one of the following values:

- Name - the name of the element
- Label - the label assigned to the element
- Description - the content of the Description field
- Specification (for Requirements only) - the content of the Specification field

*Default = Description*

## ShowForm

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the appearance of the element is determined by the value that was set for the property ShowForm.

The possible values for this property vary for the different elements, as does the default value used. The values used are:

- Plain - No color background behind text
- Note - Color background behind text
- Pushpin - Color background plus pin icon

*Default = Note*

## ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values for this property varies for the different elements, as does the default value used. The different values used are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is opened with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

*Default = Relative*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).

- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## Depends

The Depends metaclass contains properties that control the appearance of dependency relation lines in use case diagrams.

### color

*The color property specifies the default color of the border of a graphical item, such as an object box.  
(Default = 0,0,255)*

### line\_style

The line\_style property specifies the default line style for a graphical item.

The possible values are:

- straight\_arrows - straight line.
- rectilinear\_arrows - rectilinear line with right-angled corners placed at appropriate locations, depending on the starting and ending points of the line
- spline\_arrows - curved line without corners
- rounded\_rectilinear\_arrows - similar to ordinary rectilinear lines but use rounded bends rather than straight ninety-degree angles

*Default = spline\_arrows*

*Default = straight\_arrows*

### ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values for this property varies for the different elements, as does the default value used. The different values used are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0

- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is opened with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

*Default = None*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## DiagramFrame

The DiagramFrame metaclass contains properties that control the appearance of an activity diagram frame.

### DefaultSize

The DefaultSize property specifies default values for diagram frame dimensions.

*Default = 20, 20, 590, 500*

### ShowName

The ShowName property determines the text to display next to a graphic element in a diagram. For most elements, you can provide a name and a label. Your descriptive label can be useful in cases where the name itself might not be sufficient due to various constraints, such as the inability to use spaces if the name of the element is to appear in the code.

The possible values for this property vary for the different elements, as does the default value used. The possible values are:

- Description - the content of the description field; relevant for elements such as comments

- **Full\_path** - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- **Label** - the label provided for the element
- **Name** - the name of the element
- **Name\_only** - the name of the element only (as opposed to the full or relative path)
- **None** - nothing should be displayed
- **Relative** - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is displayed with the class name only if the class is not positioned inside the package in the diagram.
- **Specification** - the content of the specification field; relevant for elements such as constraints

Note that for some elements, such as destruction elements in sequence diagrams and timing diagrams, the property is a boolean property that hides/shows the name of the element.

*Default = Name*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- **Label** - Show only the stereotype label (text).
- **Bitmap** - Show only the stereotype bitmap.
- **None** - Do not show stereotypes in diagrams.

*Default = Label*

## Flow

The Flow metaclass contains properties that control how information flows are displayed in use case diagrams.

### color

*The color property specifies the default color of the border of a graphical item, such as an object box. (Default = 0,147,0)*

### flowKeyword

The flowKeyword property is a Boolean value that specifies whether the flow keyword for the information flow is displayed in the diagram.

*Default = Checked*

## infoItemsColor

*The infoItemsColor property specifies the color used to draw information items in diagrams. (Default = 0,0,255)*

## line\_style

The line\_style property specifies the default line style for a graphical item.

The possible values are:

- straight\_arrows - straight line.
- rectilinear\_arrows - rectilinear line with right-angled corners placed at appropriate locations, depending on the starting and ending points of the line
- spline\_arrows - curved line without corners
- rounded\_rectilinear\_arrows - similar to ordinary rectilinear lines but use rounded bends rather than straight ninety-degree angles

*Default = spline\_arrows*

*Default = rectilinear\_arrows*

## ShowConveyed

The ShowConveyed property determines whether or not item flows should be displayed alongside the flows that convey them, and if so, what text should be displayed for the item flows. The property can take any of the following values:

- Name - the name of the item flow
- Label - the label of the item flow
- None - nothing should be displayed for the item flow

Note that this property only affects the display of new flows added to a diagram. The display of item flows for flows already on a diagram can be controlled by selecting the Display Options... item from the context menu for flows.

*Default = Name*

## General

Contains properties relating to the general appearance of use case diagrams.



## SysMLDiagramKindTitle

When a diagram is created in SysML projects, the diagram frame contains a header for the diagram that includes information such as the name of the package that contains the diagram. The property SysMLDiagramKindTitle is used to specify a prefix for the diagram frame header that reflects the type of diagram. For example, when you create an internal block diagram, the header in the diagram frame starts with the prefix "ibd".

Note that the structure of the entire header is controlled by the property General::Graphics::DiagramHeader.

*Default = uc*

## Inheritance

The Inheritance metaclass contains properties that control the appearance of inheritance lines in use case diagrams.

### line\_style

The line\_style property specifies the default line style for a graphical item.

The possible values are:

- straight\_arrows - straight line.
- rectilinear\_arrows - rectilinear line with right-angled corners placed at appropriate locations, depending on the starting and ending points of the line
- spline\_arrows - curved line without corners
- rounded\_rectilinear\_arrows - similar to ordinary rectilinear lines but use rounded bends rather than straight ninety-degree angles

*Default = spline\_arrows*

*Default = tree*

### ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values for this property varies for the different elements, as does the default value used. The different values used are:

- Description - the content of the description field; relevant for elements such as comments

- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is opened with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

*Default = None*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## Note

The Note metaclass contains properties that control the appearance of notes in use case diagrams.

## ShowForm

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the appearance of the element is determined by the value that was set for the property ShowForm.

The possible values for this property vary for the different elements, as does the default value used. The values used are:

- Plain - No color background behind text
- Note - Color background behind text
- Pushpin - Color background plus pin icon

*Default = Note*

# Package

The Package metaclass contains properties that specify the appearance of packages in use case diagrams.

## QuickNavigationCategories

Graphic elements in diagrams can include a list of shortcuts that are accessible from the title bar of the element. You can use the property QuickNavigationCategories to specify the categories of items that should be included in this shortcut list and the order in which the categories should appear in the list. The value of the property should be a comma-separated list of categories. The categories that can be included are:

- Activities
- DiagramReferences
- Diagrams
- Hyperlinks
- InternalBlockDiagrams
- MainBehavior
- MainDiagram
- References
- StateCharts

If you want greater control over the content of the navigation list (such as showing/hiding specific elements within a given category, or changing the order of elements within a specific category), use the Compartments list, which can be found under "Display Options".

*Default = MainBehavior,MainDiagram,Hyperlinks,Diagrams*

## ShowName

The ShowName property specifies how the name of an object should be displayed. The possible values are as follows:

- Full\_path - Show the object name with the full path.. For example, "Default::A.B."
- Relative - Show the object name with a relative path. For example, "A.B."
- Name\_only - Show only the object name without any path information. For example, "B."

*Default = Name\_only*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values

are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## Requirement

The Requirement metaclass contains properties that control the appearance of requirements in use case diagrams.

### Compartments

The Compartments property determines which of the available compartments are displayed by default for the various types of elements. The value for this property is a comma-delimited string containing the names of the compartments that should be visible. Since the available compartments vary from element to element, it is recommended that you do not try to set the value of this property by using the Properties window or directly in the .prp file. Rather, you should use an element Display Options to set which compartments are visible, and then use the Make Default option to apply these settings at the diagram or project level for new elements of this type.

*Default = ID, Specification*

### QuickNavigationCategories

Graphic elements in diagrams can include a list of shortcuts that are accessible from the title bar of the element. You can use the property QuickNavigationCategories to specify the categories of items that should be included in this shortcut list and the order in which the categories should appear in the list. The value of the property should be a comma-separated list of categories. The categories that can be included are:

- Activities
- DiagramReferences
- Diagrams
- Hyperlinks
- InternalBlockDiagrams
- MainBehavior
- MainDiagram
- References
- StateCharts

If you want greater control over the content of the navigation list (such as showing/hiding specific

elements within a given category, or changing the order of elements within a specific category), use the Compartments list, which can be found under "Display Options".

*Default = Hyperlinks*

## RequirementNotation

The RequirementNotation property determines how Requirement elements are displayed. This property can be set to one of two styles:

- Note\_Style
- Box\_Style

If RequirementNotation is set to Note\_Style, then the appearance of the element is determined by the value selected for the property Requirement::ShowForm: Note, Plain, or PushPin.

If RequirementNotation is set to Box\_Style, then the element resembles a class-box, and you can control the display of compartments for the element.

*Default = Note\_Style*

## ShowAnnotationContents

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the text displayed on the element in the diagram determined by the value that was set for the property ShowAnnotationContents. The property can take one of the following values:

- Name - the name of the element
- Label - the label assigned to the element
- Description - the content of the Description field
- Specification (for Requirements only) - the content of the Specification field

*Default = Description*

## ShowForm

When the Note\_Style style is selected for annotation elements, such as comments or constraints, the appearance of the element is determined by the value that was set for the property ShowForm.

The possible values for this property vary for the different elements, as does the default value used. The values used are:

- Plain - No color background behind text
- Note - Color background behind text
- Pushpin - Color background plus pin icon

*Default = Note*

## ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values for this property varies for the different elements, as does the default value used. The different values used are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is opened with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

*Default = Relative*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*

## SpecificationAnnotationFormat

When a requirement is displayed on a diagram, you can choose to have the element show one of the following: its name, its label, its description, the specification text for the requirement.

If you choose to display the specification text, you can use the property SpecificationAnnotationFormat to customize the text that is displayed.

The following keywords can be included in the value of the property: \$Specification (to include the specification text) and \$ID (to include the ID that was specified in the Features dialog for the requirement).

Note that this property affects the displayed text only when a requirement uses the "Note" notation style. When using the "Box" notation style, only the specification text is displayed, regardless of the value of this property.

*Default = \$Specification*

## SystemBox

The SystemBox metaclass contains properties that control the appearance of system boxes in use case diagrams.

### color

*The color property specifies the default color of the border of a graphical item, such as an object box. (Default = 128,0,255)*

### Fillcolor

The Fillcolor property specifies the default fill color for the object.

*Default = 0,255,255*

### line\_width

*The line\_width property specifies the default line width, in pixels, for drawing lines (for example, action state lines). (Default = 1)*

### name\_color

*The name\_color property specifies the default color of names of graphical items. (Default = 128,128,0)*

## UseCase

The UseCase metaclass contains properties that control the appearance of use cases in use case diagrams.

### LabelsStyle

Ordinarily, if you draw a use case element on a use case diagram, Rational Rhapsody displays the name of the use case inside the element. If you have chosen to display the label instead, then the label is opened

inside the element.

This means that in cases where the label is very long, you must enlarge the element in order to have the entire label displayed.

The LabelsStyle property can be used to get around this constraint. If you change the value of the property to Caption, then the label is also displayed below the element and the text display area is automatically enlarged so that the entire label is always displayed.

Note that the value of this property also affects the display of the use case name if you have chosen to show the name rather than the label.

*Default = "Default"*

## ShowName

The ShowName property determines the text that should be displayed next to a graphic element in a diagram. For most elements, Rational Rhapsody allows you to provide a name and a label. This allows you to provide a descriptive label in cases where the name itself might not be sufficient due to various constraints. For example, the inability to use spaces if the name of the element is to appear in the code. The possible values for this property varies for the different elements, as does the default value used. The different values used are:

- Description - the content of the description field; relevant for elements such as comments
- Full\_path - the full path describing the hierarchical position of an element, for example, package\_1::package\_1b::class\_0
- Label - the label provided for the element
- Name - the name of the element
- Name\_only - the name of the element only (as opposed to the full or relative path)
- None - nothing should be displayed
- Relative - path describing the hierarchical position of an element, but only including the information that is not apparent from the depiction of the element in the diagram. For example, the name of the package containing a class is opened with the class name only if the class is not positioned inside the package in the diagram.
- Specification - the content of the specification field; relevant for elements such as constraints

*Default = relative*

## ShowStereotype

The ShowStereotype property specifies how stereotypes are shown in UML diagrams. The possible values are as follows:

- Label - Show only the stereotype label (text).
- Bitmap - Show only the stereotype bitmap.
- None - Do not show stereotypes in diagrams.

*Default = Label*



# UseCaseDiagram

The UseCaseDiagram metaclass contains a property that specifies the background color of a use case diagram.

## FillColor

The Fillcolor property specifies the default fill color for the object.

*Default = 218,218,218*

# VisualStudio

---

The VisualStudio subject contains properties related to integration of Rhapsody with Visual Studio.

## DefaultEnvironments

The DefaultEnvironments metaclass contains properties related to the default environment to use for Visual Studio configurations in your Rhapsody models.

### CleanIDETagsWhileDisconnect

When reconnecting to a project from the Rational Rhapsody toolbar in Visual Studio, a number of configuration settings are restored to their default value. These settings include:

- The Directory field on the Settings tab of the Features window.
- The IDESolution tag on the Tags tab of the Features window.
- The IDEWorkspace tag on the Tags tab of the Features window.

If you want Rational Rhapsody to keep the values that were previously set for these settings, rather than restoring the default values, set the value of the property CleanIDETagsWhileDisconnect to False.

*Default = True*

### OverridePropertiesWhileConnect

When reconnecting to a project from the Rational Rhapsody toolbar in Visual Studio, the values of a number of properties are modified, even for projects that are read-only. These properties include:

- CG::Configuration::MainGenerationScheme
- CG::File::ImplementationHeader
- <lang>\_CG::MSVC::EntryPoint

If you don't want the values of these properties to be modified, change the value of OverridePropertiesWhileConnect to False.

*Default = True*

## VisualStudio

If you have an existing configuration in your Rhapsody model, and you select Change to > Visual Studio Configuration from the pop-up menu for the configuration, then the value of the Environment field for the configuration (on the Settings tab of the Features windows) will be changed to the value that was selected for the property VisualStudio::DefaultEnvironments::VisualStudio.

*Default = MSVC*

# WebComponents

---

The WebComponents subject contains metaclasses that contain properties that control whether Rational Rhapsody components can be managed from the Web. This subject also contains the properties that specify the necessary framework for code generation.

## Attribute

The Attribute metaclass contains properties that determine whether attributes can be managed from the Web.

### ApplyUserHelpers

If you provide a getter and/or setter for an attribute, instead of having Rational Rhapsody generate them, then the webify code that is generated uses these getters/setters. If for some reason you do not want Rational Rhapsody to use the user-provided getters/setters in the webify code, you can set the value of the ApplyUserHelpers property to False.

If the property is set to False, Rational Rhapsody does not use the user-provided getters/setters, nor does it autogenerate getters/setters for this purpose.

*Default = Checked*

### WebManaged

The WebManaged property is a Boolean value that specifies whether the element can be managed from the Web. This property can be used as an alternative to stereotyping an element.

*Default = Cleared*

## Class

The Class metaclass contains properties that determine whether classes can be managed from the Web.

### WebifyFullClassName

The string used in the code to register webified elements cannot exceed 64 characters. To overcome this limitation, you can set the value of the WebifyFullClassName property to False, and then Rational Rhapsody uses only the class name rather than the full name of the class (including namespaces).

*Default = Checked*

## WebifyPropagateLinks

If you specify a class to be WebManaged, classes that are associated with the class also contain calls to webify operations. If, however, such associated classes are not specified as WebManaged, then these calls result in compilation errors. In such cases, you can prevent Rational Rhapsody from generating calls to webify operations in the associated classes by setting the value of the WebifyPropagateLinks property to False.

*Default = Checked*

## WebManaged

The WebManaged property is a Boolean value that specifies whether the element can be managed from the Web. This property can be used as an alternative to stereotyping an element.

*Default = Cleared*

## Configuration

The Configuration metaclass contains properties that specify the configuration settings for the model.

## CommunicationLayerScheme

The CommunicationLayerScheme property specifies which communication layer to use with the Webify Toolkit. The possible values are as follows:

- Auto - Leave the default value for the program. Currently, this is SUN Java.
- JavaScript - Make sure the communication works with JavaScript.
- Microsoft VM - Java communication layer. There is a cosmetic advantage to this option.

Change the property value to JavaScript if you have the problem with an empty right pane, as described in the Known Limitations section of the Release Notes.

*Default = Auto*

Note: Because Microsoft no longer supports VM in the IE environment, the Webify feature uses SUN Java by default.

This affects any models with the WebComponents::Configuration::CommunicationlayerScheme property that is set to Auto or Microsoft VM.

- If this property is set to JavaScript, then there should be no change.
- If the above property is set to Auto or MicrosoftVM, then you need to be sure that your Sun Java is enabled in Microsoft Internet Explorer.

Note that the required VM version is 1.4.2.04 or higher.

## HomePageURL

The HomePageURL property specifies the URL to the home page. This setting corresponds to the home page attribute defined in the Advanced Webify Toolkit Settings window.

*Default = cgibin.?Abs\_App=Abstract\_Default*

## Port

The Port property specifies the server port. This setting corresponds to the server port attribute defined in the Advanced Webify Toolkit Settings window.

*Default = 80*

## RefreshPeriod

The RefreshPeriod property specifies the refresh timeout. This setting corresponds to the refresh period parameter defined in the Advanced Webify Toolkit Settings window.

*Default = 1000*

## SignaturePageURL

The SignaturePageURL property specifies the URL to the signature page. This setting corresponds to the signature page attribute defined in the Advanced Webify Toolkit Settings window.

*Default = sign.htm*

## Event

The Event metaclass contains a property that determines whether events can be managed from the Web.

## WebManaged

The WebManaged property is a Boolean value that specifies whether the element can be managed from the Web. This property can be used as an alternative to stereotyping an element.

*Default = Cleared*

## File

The File metaclass contains a property that determines whether files can be managed from the Web.

### WebManaged

The WebManaged property is a Boolean value that specifies whether the element can be managed from the Web. This property can be used as an alternative to stereotyping an element.

*Default = Cleared*

## Operation

The Operation metaclass contains a property that determines whether operations can be managed from the Web.

### WebManaged

The WebManaged property is a Boolean value that specifies whether the element can be managed from the Web. This property can be used as an alternative to stereotyping an element.

*Default = Cleared*

## WebFramework

The WebFramework metaclass contains properties that control the instrumentation code that is generated for Web-enabled elements.

### GenerateInstrumentationCode

The GenerateInstrumentationCode property is a Boolean value that determines whether code generation for the corresponding configuration is enabled. The value of this property corresponds to the Web Instrumentation checkbox on the Settings page for a configuration.

*Default = Cleared*

### WebInstrumentationIncludes

The `WebInstrumentationIncludes` property is used to specify the dependencies that must be included for elements that are web-enabled.

*Default = `WebComponents/WebComponentsTypes.h` (C, C++),  
`com.ibm.rational.rhapsody.webComponents.*` (Java)*



# WSDL

---

The WSDL subject contains properties that support Web Service Description Language.

## Package

The Package metaclass contains properties that support WSDL (Web Service Description Language).

## Namespaces

The Namespaces property is used when generating a WSDL specification file from a "services" stereotyped model (in Rational Rhapsody by using the NetCentric profile). In addition, the Namespaces property defines the XML namespace used within the WSDL file. Namespaces identify where the data types used in the XML file are defined. You can enter a string or a URL.

*Default = xsd=http://www.w3.org/2001/XMLSchema soap=http://schemas.xmlsoap.org/wsdl/soap/*

## TargetNamespace

The TargetNamespace property is used when generating a WSDL specification file from a "services" stereotyped model (in Rational Rhapsody by using the NetCentric profile). The user defines the "targetNamespace" attribute of the "definition" WSDL tag. The "targetNamespace" is an XML attribute. Here is where newly created elements and attributes reside.

*Default = http://www.yourCompanyName.com/yourProductName/*

# XSD

---

The XSD subject contains properties that support XML Schema Definition.

## Type

The Type metaclass contains properties that support XSD (XML Schema Definition).

### ImpXSDType

The ImpXSDType property is used when generating WSDL specification file from "services" stereotyped model. When a user designs a "services" stereotyped model in Rational Rhapsody, this means no "XSD" types is used. They are using the usual types such as int, char, and so on. When they generate a WSDL specification from their model, these types should be mapped to XSD types that are already being modeled in Rational Rhapsody in the NetCentric profile. The mapping is being done through the ImpXSDType property.

*Default = Empty string*