

# Toolset Guide

RATIONAL ROSE® REALTIME

VERSION: 2003.06.00

PART NUMBER: 800-026113-000

WINDOWS/UNIX



## **Legal Notices**

©1993-2003, Rational Software Corporation. All rights reserved.

Part Number: 800-026113-000

Version Number: 2003.06.00

This manual (the "Work") is protected under the copyright laws of the United States and/or other jurisdictions, as well as various international treaties. Any reproduction or distribution of the Work is expressly prohibited without the prior written consent of Rational Software Corporation.

Rational, Rational Software Corporation, the Rational logo, Rational Developer Network, AnalystStudio, ClearCase, ClearCase Attache, ClearCase MultiSite, ClearDDTS, ClearGuide, ClearQuest, ClearTrack, Connexis, e-Development Accelerators, DDTS, Object Testing, Object-Oriented Recording, ObjecTime, ObjecTime Design Logo, Objectory, PerformanceStudio, PureCoverage, PureDDTS, PureLink, Purify, Quantify, Rational Apex, Rational CRC, Rational Process Workbench, Rational Rose, Rational Suite, Rational Suite ContentStudio, Rational Summit, Rational Visual Test, Rational Unified Process, RUP, RequisitePro, ScriptAssure, SiteCheck, SiteLoad, SoDA, TestFactory, TestFoundation, TestStudio, TestMate, VADS, and XDE, among others, are trademarks or registered trademarks of Rational Software Corporation in the United States and/or in other countries. All other names are used for identification purposes only, and are trademarks or registered trademarks of their respective companies.

Portions covered by U.S. Patent Nos. 5,193,180 and 5,335,344 and 5,535,329 and 5,574,898 and 5,649,200 and 5,675,802 and 5,754,760 and 5,835,701 and 6,049,666 and 6,126,329 and 6,167,534 and 6,206,584. Additional U.S. Patents and International Patents pending.

U.S. GOVERNMENT RIGHTS. All Rational software products provided to the U.S. Government are provided and licensed as commercial software, subject to the applicable license agreement. All such products provided to the U.S. Government pursuant to solicitations issued prior to December 1, 1995 are provided with "Restricted Rights" as provided for in FAR, 48 CFR 52.227-14 (JUNE 1987) or DFARS, 48 CFR 252.227-7013 (OCT 1988), as applicable.

WARRANTY DISCLAIMER. This document and its associated software may be used as stated in the underlying license agreement. Except as explicitly stated otherwise in such license agreement, and except to the extent prohibited or limited by law from jurisdiction to jurisdiction, Rational Software Corporation expressly disclaims all other warranties, express or implied, with respect to the media and software product and its documentation, including without limitation, the warranties of merchantability, non-infringement, title or fitness for a particular purpose or arising

from a course of dealing, usage or trade practice, and any warranty against interference with Licensee's quiet enjoyment of the product.

**Third Party Notices, Code, Licenses, and Acknowledgements**

Portions Copyright ©1992-1999, Summit Software Company. All rights reserved.

Microsoft, the Microsoft logo, Active Accessibility, Active Client, Active Desktop, Active Directory, ActiveMovie, Active Platform, ActiveStore, ActiveSync, ActiveX, Ask Maxwell, Authenticode, AutoSum, BackOffice, the BackOffice logo, bCentral, BizTalk, Bookshelf, ClearType, CodeView, DataTips, Developer Studio, Direct3D, DirectAnimation, DirectDraw, DirectInput, DirectX, DirectXJ, DoubleSpace, DriveSpace, FrontPage, Funstone, Genuine Microsoft Products logo, IntelliEye, the IntelliEye logo, IntelliMirror, IntelliSense, J/Direct, JScript, LineShare, Liquid Motion, Mapbase, MapManager, MapPoint, MapVision, Microsoft Agent logo, the Microsoft eMBEDded Visual Tools logo, the Microsoft Internet Explorer logo, the Microsoft Office Compatible logo, Microsoft Press, the Microsoft Press logo, Microsoft QuickBasic, MS-DOS, MSDN, NetMeeting, NetShow, the Office logo, Outlook, PhotoDraw, PivotChart, PivotTable, PowerPoint, QuickAssembler, QuickShelf, RelayOne, Rushmore, SharePoint, SourceSafe, TipWizard, V-Chat, VideoFlash, Visual Basic, the Visual Basic logo, Visual C++, Visual C#, Visual FoxPro, Visual InterDev, Visual J++, Visual SourceSafe, Visual Studio, the Visual Studio logo, Vizact, WebBot, WebPIP, Win32, Win32s, Win64, Windows, the Windows CE logo, the Windows logo, Windows NT, the Windows Start logo, and XENIX, are either trademarks or registered trademarks of Microsoft Corporation in the United States and/or in other countries.

Sun, Sun Microsystems, the Sun Logo, Ultra, AnswerBook 2, medialib, OpenBoot, Solaris, Java, Java 3D, ShowMe TV, SunForum, SunVTS, SunFDDI, StarOffice, and SunPCi, among others, are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Purify is licensed under Sun Microsystems, Inc., U.S. Patent No. 5,404,499.

Licensee shall not incorporate any GLOBEtrotter software (FLEXIm libraries and utilities) into any product or application the primary purpose of which is software license management.

BasicScript is a registered trademark of Summit Software, Inc.

**Design Patterns: Elements of Reusable Object-Oriented Software**, by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Copyright © 1995 by Addison-Wesley Publishing Company, Inc. All rights reserved.

Additional legal notices are described in the legal\_information.html file that is included in your Rational software installation.

# Contents

<b>Preface</b> .....	<b>xxvii</b>
Audience .....	xxviii
Other Resources .....	xxviii
Rational Rose RealTime Integrations With Other Rational Products .....	xxix
Contacting Rational Customer Support .....	xxx
<b>1 Using the Online Help</b> .....	<b>1</b>
Using the Online Help System .....	1
Accessing What's This Help .....	1
Accessing Extended Help .....	1
Tutorials .....	2
Using the Help Viewer .....	2
Getting More Out of Help .....	2
Finding a Help Topic .....	3
Creating a List of Favorite Help Topics .....	3
Maintaining a Single Favorites List .....	3
Copying a Help Topic .....	4
Printing the Current Help Topic .....	5
Obtaining Help in a Dialog .....	5
Finding Topics Using the Toolbar Buttons .....	5
Hiding or Showing the Navigation Pane .....	6
Using Accessibility Shortcut Keys in the Help Viewer .....	6
Using the Context Menu Commands .....	9
About the Search Tab .....	9
Searching for Help Topics .....	10
Searching for Words or Phrases .....	11
Defining Search Terms .....	12
Using Nested Expressions when Searching .....	12
Searching within Search Results .....	14

Changing the Help Viewer . . . . .	14
Customizing the Help Viewer . . . . .	14
Changing Format or Styles for Accessibility . . . . .	15
Viewing Topics Grouped by Information Type. . . . .	15
Changing the Font Size of a Topic . . . . .	15
Changing Colors in the Topic Pane of the Help Viewer . . . . .	16
Related Documentation . . . . .	16
<b>2 Overview of Rational Rose RealTime . . . . .</b>	<b>17</b>
Developing Using Rational Rose RealTime . . . . .	17
Using Languages and Code Generation . . . . .	18
Compiling Models . . . . .	18
Using the Services Library . . . . .	19
Capsules, Protocols, Ports, Capsule State and Structure Diagrams . . . . .	19
Capsules. . . . .	20
Protocols. . . . .	20
Ports . . . . .	21
State Diagrams . . . . .	21
Capsule Structure Diagrams . . . . .	21
Executable Models . . . . .	21
Constructing Models in Rational Rose RealTime . . . . .	22
Modeling Elements . . . . .	22
Required Elements . . . . .	22
Diagrams . . . . .	24
Development Process . . . . .	25
Essential Workflows . . . . .	26
<b>3 User Interface Overview. . . . .</b>	<b>29</b>
Startup Screen . . . . .	29
Create New Model Dialog . . . . .	30
Using the Startup Frameworks . . . . .	31
Application Window . . . . .	37
Browsers . . . . .	37
Toolbar . . . . .	38
Diagrams. . . . .	38
Toolboxes . . . . .	38
Menu Bar. . . . .	38
About Rational Rose RealTime Dialog . . . . .	39

The Toolbar . . . . .	39
Menus . . . . .	43
Menu Bar . . . . .	44
File Menu . . . . .	45
File Menu Operations . . . . .	45
Edit Menu . . . . .	49
Parts Menu . . . . .	54
View Menu . . . . .	56
Browse Menu . . . . .	57
Select Diagram Dialog . . . . .	57
Browse Menu Operations . . . . .	58
Build Menu . . . . .	61
Report Menu . . . . .	64
Query Menu . . . . .	66
Tools Menu . . . . .	68
Layout . . . . .	68
Add-Ins Menu . . . . .	75
Window Menu . . . . .	76
Help Menu . . . . .	77
Browsers . . . . .	78
Tabs . . . . .	80
Model View Tab . . . . .	80
Containment View Tab . . . . .	80
Inheritance Tab . . . . .	80
RTS Tab . . . . .	80
Navigating . . . . .	80
Displaying the Browser . . . . .	81
Refreshing the Browser . . . . .	81
Multiple Browsers . . . . .	81
Filtering . . . . .	81
Diagram Editors . . . . .	82
Diagram Specification - General Tab . . . . .	83
Diagram Specification - Diagrams Tab . . . . .	83
Adding Icons to a Diagram . . . . .	83
Opening Specifications . . . . .	84
Shortcut Menu . . . . .	84
Background Shortcut Menu . . . . .	84
Scroll Bars . . . . .	85

Overview Navigator and Toolset Buttons, and Class, Capsule, and Protocol Specification Context Menus . . . . .	86
Overview Navigator Button . . . . .	86
Toolset Buttons . . . . .	86
Context Options for Specification Dialogs . . . . .	86
Context Options for Other Controls . . . . .	87
Sequence Diagram Context Menu . . . . .	89
Toolboxes . . . . .	91
<b>Specification Dialogs . . . . .</b>	<b>92</b>
Spreadsheet-type Functionality for List Controls within a Specification Dialog. . . . .	93
Tabs . . . . .	95
Actions Tab . . . . .	95
Attributes Tab . . . . .	96
Components Tab . . . . .	97
Detail Tab . . . . .	97
Files Tab . . . . .	98
General Tab . . . . .	98
Operations Tab . . . . .	99
Relations Tab . . . . .	101
Swimlanes Tab . . . . .	101
Transitions Tab . . . . .	101
Unit Information Tab . . . . .	101
Descriptions . . . . .	101
Scratch Pad Packages . . . . .	103
Searching and Sorting . . . . .	104
Using Sort. . . . .	104
Find In Model Dialog. . . . .	105
Replace Dialog . . . . .	106
Searching Code . . . . .	106
<b>4 Other Application Windows. . . . .</b>	<b>109</b>
Description Window . . . . .	109
Displaying the Description Window . . . . .	109
Documentation Tab. . . . .	110
Code Tab . . . . .	110
Word Wrap . . . . .	110
Pull-down Menu . . . . .	111
Popup Menu . . . . .	111



Adding Documentation to Model Elements .....	111
Adding Code to Model Elements .....	112
Output Window .....	112
Log Tab .....	112
Build Log Tab .....	113
Saving Build Output to a Log File .....	113
Build Errors Tab .....	114
Filtering Build Results .....	114
Sorting Build Results .....	114
Unknown Compiler Message Stream .....	114
Find Tab .....	115
Watch Tab .....	115
Refreshing the Watch Values.....	115
Specification History Window .....	115
Locking Specification Dialogs .....	116
ToolTips .....	116
Keyboard Shortcuts .....	116
Specification History Shortcut Menu .....	117
<b>5 Printing.....</b>	<b>119</b>
Print Specifications .....	119
General Tab .....	119
Properties Dialog .....	120
Diagrams Tab.....	120
Specifications Tab .....	121
Layout Tab .....	122
Print Setup .....	123
Printer Area .....	123
Paper Area.....	123
Orientation Area.....	123
<b>6 Opening and Saving Models .....</b>	<b>125</b>
Unique Ids .....	125
Opening Models .....	129
Model Specification .....	129
General Tab .....	129
Source Control Tab .....	130
Files Tab.....	130
Unit Information Tab.....	130
A Workspace .....	131
User-specific Working Environment Settings (.rtusr, .rtto and .rtwks) .....	132

Opening Models from ObjecTime Developer 5.2.1 .....	132
Limitations and Restrictions .....	133
Opening Rational Rose Models .....	134
Limitations and Restrictions .....	135
Importing Rational Rose Generated Code .....	136
Limitations and Restrictions .....	136
<b>7 Use Case Diagrams .....</b>	<b>137</b>
Creating a Use Case Diagram .....	137
Using the Use Case Diagram Editor .....	138
Usage Tips .....	139
Use Case Diagram Toolbox .....	139
<b>8 Defining Use Cases and Actors .....</b>	<b>141</b>
Creating a Use Case .....	141
Use Case Specification .....	141
General Tab .....	142
Diagram Tab .....	143
Relations Tab .....	143
Files Tab .....	143
Creating an Actor .....	143
Actor Specification .....	144
<b>9 Creating Class Diagrams .....</b>	<b>145</b>
Creating a Class Diagram .....	145
Using the Class Diagram Editor .....	146
Class Diagram Toolbox .....	149
Creating Relationships .....	153
Creating Association Relationships .....	154
Association Properties .....	155
Association Specification .....	155
General Tab .....	155
Detail Tab .....	156
End A and B General Tabs .....	157
End A and B Detail Tabs .....	158
Creating Aggregation Relationships .....	160
Creating an Association Class .....	162
Aggregation Specification .....	162

Creating Inheritance Relationships .....	162
Creating an Inheritance Tree .....	163
Exclusions .....	163
Generalize Specification .....	163
General Tab .....	164
Inheritance in Rational Rose RealTime .....	164
Promoting and Demoting Elements .....	165
Potential Conflicts Caused by Promote/Demote .....	165
Excluding Elements .....	165
Reinheriting Excluded Elements .....	166
Rearranging Inheritance Hierarchies .....	166
Inheritance Tab in Browser .....	166
Creating Dependency Relationships .....	167
Graphical Notation .....	168
Naming .....	168
Valid Applications .....	168
Add Class Dependencies Wizard .....	168
Dependency Specification .....	168
General Tab .....	169
Creating Reflexive Relationships .....	170
Changing the Directionality of an Association .....	170
Creating Package Relationships .....	170
Creating Realize Relationships .....	171
Naming .....	171
Valid Applications .....	171
Realize Relationship Specification .....	171
General Tab .....	171
Inserting Dependencies, Generalizations, and Realizations on the Relations Tab .....	172
Inserting Dependencies .....	172
Inserting Generalizations .....	173
Inserting Realizations .....	175
Changing the End Class .....	176
Adding and Hiding Classes, and Filtering Class Relationships .....	178
Using State Machine Code Generation for Classes .....	178
Configuring a Simple Model .....	178
Generating Component Libraries for Classes without RTS Dependencies ...	180
Creating State Machine Trigger Operations .....	182
Configuring the <i>trigger</i> Stereotype for an Operation .....	183

Generating State Machine Code .....	185
Support for Code Sync .....	187
Considerations .....	189
Hello World Implementation and Header Files .....	190
Using Constructors .....	195
C Language .....	196
C++ Language .....	197
Using Return, Break, and Continue Statements .....	198
Specifying History .....	199
No Refinement .....	200
Overriding Virtual Operations .....	200
Generation of Parameterized and Instantiated Classes .....	200
Parameterized Classes .....	201
Relationships .....	203
Instantiated Classes .....	203
Relationships .....	204
Limitations .....	205
<b>10 Creating Collaboration Diagrams .....</b>	<b>207</b>
Creating Capsule Structure .....	207
Using the Structure Editor .....	208
UML Options .....	209
Structure Diagram Browser Context Menu Options .....	209
Structure Diagram Toolbox .....	211
Creating a Port .....	212
Creating a Non-Wired Port Using a System Protocol .....	213
Port Specification .....	213
General Tab .....	213
Files Tab .....	217
Port Role Specification Dialog .....	218
Adding a Capsule Role .....	219
Capsule Role Specification .....	219
General Tab .....	219
Connecting Ports on Capsule Roles Together .....	221
Connector Specification .....	221
General Tab .....	221
Creating a Collaboration Diagram .....	222

Using the Collaboration Diagram Editor . . . . .	222
Relationship Between Collaborations and Sequences . . . . .	223
Opening a Sequence Diagram . . . . .	223
Sequence Overlays . . . . .	223
Code Generation . . . . .	223
Collaboration Diagram Toolbox . . . . .	224
Classifier Role Specification . . . . .	225
General Tab . . . . .	226
Files Tab . . . . .	226
Association Role Specification . . . . .	226
General Tab . . . . .	226
Files Tab . . . . .	227
<b>11 Creating State Diagrams . . . . .</b>	<b>229</b>
Creating Capsule State Machines . . . . .	229
Using the State Diagram Editor . . . . .	230
State Diagram Toolbox . . . . .	232
State Specification . . . . .	234
General Tab . . . . .	234
Entry Actions / Exit Actions Tabs . . . . .	234
Aggregating and Decomposing State Machines . . . . .	235
Transition Specification . . . . .	235
General Tab . . . . .	235
Triggers Tab . . . . .	235
Actions Tab . . . . .	236
Files Tab . . . . .	236
Choice Point Specification . . . . .	237
General Tab . . . . .	237
Condition Tab . . . . .	237
Files Tab . . . . .	237
Initial State Specification . . . . .	237
General Tab . . . . .	237
Files Tab . . . . .	238
Junction Point Specification . . . . .	238
General Tab . . . . .	238
Files Tab . . . . .	239
Event Editor Dialog . . . . .	239
EventGuard Specification Dialog Box . . . . .	240
Adding a State . . . . .	242
Adding a Choice Point . . . . .	242

Drawing Transitions Between States .....	242
Specifying the Transition .....	245
Drawing the Initial Transition .....	245
Defining State Transition Trigger Events .....	246
State Diagrams .....	246
Joining Transitions .....	247
Creating Nested States .....	248
Positioning from a Superclass for Transitions .....	248
State Diagram - Showing Triggers and Code for Transitions .....	250
Identifying Self Transitions on the Transitions Tab in the State Specification Dialog Box .....	254
Descriptions .....	255
<b>12 Creating Activity Diagrams .....</b>	<b>257</b>
Modeling Using Activity Diagrams .....	258
Activity Diagrams .....	258
Creating an Activity Diagram .....	260
Activity Diagram Specification Dialog .....	261
Activity Diagram Specification Dialog - General Tab .....	262
StateMachine Specification for State/Activity .....	262
StateMachine Specification for State/Activity - General Tab .....	262
StateMachine Specification for State/Activity - Files Tab .....	263
Activity Diagram Tools .....	263
Activities .....	264
Activity History .....	264
Specifying Actions for Activities .....	264
Nested Activities .....	265
Manipulating Nested Activities .....	265
Creating Nested Activities .....	265
Activity Specification Dialog .....	266
Activity Specification Dialog - General Tab .....	266
Activity Specification Dialog - Actions Tab .....	268
Activity Specification Dialog - Transitions Tab .....	268
Activity Specification Dialog - Swimlanes Tab .....	268
Activity Specification Dialog - Files Tab .....	269

Actions . . . . .	269
Action Specification Dialog . . . . .	270
Action Specification Dialog - Detail Tab . . . . .	270
Action Specification Dialog - Files Tab . . . . .	271
Decisions . . . . .	271
Decision Specification Dialog . . . . .	272
Decision Specification Dialog - General Tab . . . . .	272
Decision Specification Dialog - Transitions Tab . . . . .	273
Decision Specification Dialog - Swimlanes Tab . . . . .	273
Decision Specification Dialog - Files Tab . . . . .	273
End State . . . . .	274
Start State . . . . .	274
States . . . . .	275
Specifying Actions for States . . . . .	275
Nested States . . . . .	276
Manipulating Nested States . . . . .	276
Creating Nested States . . . . .	276
State History . . . . .	277
State Specification Dialog . . . . .	277
State Specification Dialog - General Tab . . . . .	277
State Specification Dialog - Actions Tab . . . . .	279
State Specification Dialog - Transitions Tab . . . . .	279
State Specification Dialog - Swimlanes Tab . . . . .	279
State Specification Dialog - Files Tab . . . . .	280
Trigger Specification Dialog . . . . .	280
Trigger Specification Dialog - Detail Tab . . . . .	280
Trigger Specification Dialog - Files Tab . . . . .	281
Synchronizations . . . . .	281
Synchronization Specification Dialog . . . . .	282
Synchronization Specification Dialog - General Tab . . . . .	282
Synchronization Specification Dialog - Transitions Tab . . . . .	283
Synchronization Specification Dialog - Files Tab . . . . .	283
Transitions . . . . .	283

Transition Specification Dialog . . . . .	284
Transition Specification Dialog - General Tab . . . . .	284
Transition Specification Dialog - Detail Tab . . . . .	285
Transition Specification Dialog - Files Tab . . . . .	286
Swimlanes . . . . .	286
Creating Swimlanes . . . . .	287
Deleting a Swimlane . . . . .	287
Moving a Swimlane . . . . .	288
Displaying Multiple Views of a Swimlane . . . . .	288
Changing the Assignment of Responsibility of a Swimlane . . . . .	289
Swimlane Specification Dialog . . . . .	289
Swimlane Specification Dialog - General Tab . . . . .	289
Swimlane Specification Dialog - Files Tab . . . . .	290
Objects and Object Flows . . . . .	290
Objects . . . . .	290
Object State . . . . .	291
Object Flow . . . . .	291
Object Flows and Transitions . . . . .	292
Modeling Object State changes . . . . .	292
Creating an Object . . . . .	293
Creating an Object Flow . . . . .	293
Adding the Object, Object Flow, and Lock Selection Tools to the Toolbar . . . . .	293
Object Specification Dialog . . . . .	294
Object Specification Dialog - General Tab . . . . .	294
Object Specification Dialog - Incoming Object Flows Tab . . . . .	296
Object Specification Dialog - Outgoing Object Flows Tab . . . . .	296
Object Specification Dialog - Files Tab . . . . .	296
Object Flow Specification Dialog . . . . .	296
Object Flow Specification Dialog - General Tab . . . . .	297
Object Flow Specification Dialog - Files Tab . . . . .	297
Cutting Objects on Activity Diagrams . . . . .	298
Copying Objects on Activity Diagrams . . . . .	298
Pasting Objects on Activity Diagrams . . . . .	298



<b>13 Creating Sequence Diagrams .....</b>	<b>299</b>
Creating a Sequence Diagram .....	299
Creating a New Diagram .....	299
From the Browser .....	300
From the Structure Diagram Browser .....	300
From the Collaboration or Structure Diagram .....	300
Editing a Diagram .....	300
Adding Instances .....	301
Defining Messages .....	302
Specifying Message Details .....	302
Cloning a Sequence Diagram .....	302
Using Copy and Paste within Sequence Diagrams .....	302
Interaction Instances .....	303
Messages .....	303
Standard Diagram Elements .....	305
Known Limitations .....	305
Using the Sequence Diagram Editor .....	305
Opening Collaboration Diagrams .....	306
Reorienting Messages .....	306
Moving Messages .....	307
Sequence Diagram Toolbox .....	307
Interaction Instance Specification .....	310
General Tab .....	310
Files Tab .....	311
Interaction Specification .....	311
General Tab .....	311
Files Tab .....	311
Local Action Specification .....	312
General Tab .....	312
Detail Tab .....	312
Local State Specification .....	312
General Tab .....	312
Detail Tab .....	313
Message Specification .....	313
General Tab .....	313
Detail Tab .....	313
Port Detail Tab .....	314
Send Message Specification - Adding Ports to Capsule Classes .....	315
Sequence Validation Dialog .....	318

Focus of Control . . . . .	319
Coloring a Focus of Control . . . . .	320
Navigating Sequence Diagrams . . . . .	321
Saving Sequence Diagrams as Controlled Units . . . . .	324
Uncontrolling Sequence Diagrams . . . . .	326
Importing and Exporting Sequence Diagrams . . . . .	326
RRTDI . . . . .	326
Control Interaction Scripts . . . . .	326
ControllInteractions_CheckOut.ebs . . . . .	327
ControllInteractions_AddSequenceDiagrams.ebs . . . . .	327
ControllInteractions_CheckIn.ebs . . . . .	328
Running Scripts to Make Sequence Diagrams Controllable . . . . .	328
<b>14 Defining Capsules and Classes . . . . .</b>	<b>333</b>
Creating a Class . . . . .	333
Creating New Attributes . . . . .	334
Creating New Operations . . . . .	334
Class Specification . . . . .	335
Class Specification Content . . . . .	335
Class Specification - General Tab . . . . .	336
Class Specification - Detail Tab . . . . .	337
Class Specification - Operations Tab . . . . .	339
Class Specification - Attributes Tab . . . . .	341
Class Specification - Nested Tab . . . . .	342
Class Specification - Components Tab . . . . .	344
Class Specification - Relations Tab . . . . .	344
Class Specification - Files Tab . . . . .	345
Class Specification - Diagrams Tab . . . . .	345
Attribute Specification Dialog . . . . .	345
General Tab . . . . .	346
Detail Tab . . . . .	347
Operation Specification Dialog . . . . .	347
General Tab . . . . .	348
Detail Tab . . . . .	349
Validation Tab . . . . .	350
Semantics Tab . . . . .	352

<b>Parameter Specification Dialog</b> .....	<b>352</b>
Files Tab .....	353
<b>Creating a Capsule Class</b> .....	<b>353</b>
<b>Capsule Diagrams</b> .....	<b>354</b>
State Diagram .....	354
Structure Diagram .....	354
Undocking the Capsule Diagrams .....	354
<b>Capsule Specification</b> .....	<b>354</b>
Capsule Specification - General Tab .....	355
Capsule Specification - Diagrams Tab .....	356
Capsule Specification - Operations Tab .....	356
Capsule Specification - Attributes Tab .....	357
Capsule Specification - Capsule Roles Tab .....	358
Capsule Specification - Ports Tab .....	359
Capsule Specification - Connectors Tab .....	359
Capsule Specification - Relations Tab .....	360
Capsule Specification - Components Tab .....	360
Capsule Specification - Files Tab .....	360
<b>15 Defining Protocols</b> .....	<b>361</b>
Protocol Specification .....	361
Protocol Specification - General Tab .....	362
Protocol Specification - Signals Tab .....	362
Protocol Specification - Relations Tab .....	363
Protocol Specification - Components Tab .....	363
Protocol Specification - Diagrams Tab .....	364
Protocol Specification - Files Tab .....	364
Signal Specification .....	364
Signal Specification - General Tab .....	365
Signal Specification - Files Tab .....	365
<b>16 Defining Packages</b> .....	<b>367</b>
Introduction to Packages .....	367
Creating a Package .....	367
Packages and Class Diagrams .....	368
Package Specification .....	368
Package Specification - General Tab .....	369
Package Specification - Detail Tab .....	370
Package Specification - Relations Tab .....	370

Package Specification - Components Tab . . . . .	371
Package Specification - Files Tab . . . . .	371
Package Specification - Model Elements Tab. . . . .	371
Moving Model Elements. . . . .	372
Impact of Moving Classes or Diagrams on Configuration Management . . . . .	375
<b>17 Creating the Component and Deployment Views . . . . .</b>	<b>377</b>
Using the Component Diagram Editor . . . . .	377
Component Diagram Toolbox. . . . .	379
Using the Deployment Diagram Editor. . . . .	380
Deployment Diagram Elements . . . . .	381
Deployment Diagram Toolbox . . . . .	382
<b>18 Importing and Exporting . . . . .</b>	<b>385</b>
Importing a Petal or Package File . . . . .	385
Importing Code from Rational Rose to Rational Rose RealTime. . . . .	385
Using the Code Import Process . . . . .	386
Preparing the Rational Rose Model for Import . . . . .	386
Launching the C++ Analyzer. . . . .	387
Specifying Export Options and Selecting a Source File Location. . . . .	388
Analyzing the Code. . . . .	391
Using CodeCycle to Add Tags to Code. . . . .	393
Importing the Code . . . . .	394
Referencing an External Library . . . . .	396
Using the Convert Rose Component Wizard . . . . .	397
Exporting a File . . . . .	399
<b>19 Using Source Control. . . . .</b>	<b>401</b>
Fundamentals of Source Control in Rational Rose RealTime . . . . .	401
Using Source Control in Rational Rose RealTime . . . . .	402
Maintaining Integrity When a Model is Under Source Control . . . . .	403
Source Control Settings . . . . .	404
Optimizing Performance . . . . .	408
Accessing Source Control Operations . . . . .	408
Source Control Operations. . . . .	410
Adding Elements to Source Control. . . . .	414
Performing an Unreserved Checkout . . . . .	415

Options for Obtaining Change Management Information When Loading a Model . . . . .	417
Updating the Log . . . . .	419
Changing the CM Retrieval Option. . . . .	419
CM Retrieval Options. . . . .	420
Limitations . . . . .	421
Checking Out Files When a Newer Version Exists. . . . .	422
Get Dialog . . . . .	422
Controlling a Unit with an Uncontrolled Parent. . . . .	425
Changing Unit Ownership . . . . .	425
Limitations . . . . .	426
Viewing the ClearCase Version Tree for a VOB. . . . .	426
<b>20 Naming Guidelines . . . . .</b>	<b>429</b>
Introduction to Naming Guidelines . . . . .	429
Assigning Names . . . . .	429
Special Case Notes . . . . .	430
Using Logical Names for Model Elements . . . . .	430
Logical Name Example . . . . .	432
<b>21 Building and Executing Models. . . . .</b>	<b>437</b>
Building and Running Models . . . . .	437
Is Rational Rose RealTime a Compiler? . . . . .	438
Real-Time Services (Services Library) . . . . .	438
Before You Start. . . . .	438
Building . . . . .	439
Executing . . . . .	439
Building Basics. . . . .	439
Top-level Capsule. . . . .	440
Assigning an Active Component . . . . .	440
Creating a Component. . . . .	441
Starting a Build. . . . .	441
Generate Dialog. . . . .	442
Unable to Compile a Component? . . . . .	443
Reviewing Build Results. . . . .	444

Opening Code Generated for Model Elements . . . . .	445
Selecting Elements . . . . .	445
Selecting a Single Element . . . . .	445
Selecting Multiple Elements . . . . .	447
Using an Editor . . . . .	447
Build Menu . . . . .	447
Build Settings Dialog . . . . .	450
Active Component . . . . .	450
Active Component Instances List . . . . .	450
Build Log Tab . . . . .	450
Build Errors Tab . . . . .	451
Unknown Compiler Message Stream . . . . .	451
Component Specification . . . . .	451
Specification Content . . . . .	451
Component Specification - General Tab . . . . .	452
Component Specification - References Tab . . . . .	452
Component Specification - Relations Tab . . . . .	453
Component Specification - Files Tab . . . . .	453
Generating Documentation Fields . . . . .	453
Using Generated Documentation Fields . . . . .	457
Component Dependencies . . . . .	461
<b>22 Common Build Errors . . . . .</b>	<b>463</b>
Understanding Build Errors . . . . .	463
Missing Class Dependencies . . . . .	464
Capsule Role Name Same as Capsule Name . . . . .	464
Linking Wrong Services Library Set . . . . .	464
Compiler Not Installed Correctly . . . . .	464
Compile a Simple Hello World Program . . . . .	465
Check Environment Variables . . . . .	465
Review Your Compiler Flag Settings . . . . .	465
System Does Not Understand the Make Command . . . . .	465
Check Environment Variables . . . . .	465
Ensure that Component has Correct Make Types Configured . . . . .	465
Name Conflicts . . . . .	466
Missing Header Files, Object Files, and Libraries . . . . .	466
Compile Fails on Valid C++ Models with VC++ 5.0 or VC++ 6.0 . . . . .	467
Error Linking Capsule - Error From nmake . . . . .	467

Windows NT Compilation Command Line Limits . . . . .	467
Source File Compilation . . . . .	467
Linking . . . . .	468
Model Management - Importing Model Compilation Results . . . . .	468
Build Log Tab - Saving and Importing Compilation Results . . . . .	468
Saving the Build Output to a File Directly from the Build Log Tab . . . . .	468
Importing from the Build Log Tab . . . . .	470
Build Errors Tab - Importing Compilation Results . . . . .	471
<b>23 Running and Debugging . . . . .</b>	<b>475</b>
Execution Basics . . . . .	476
Creating a Component Instance . . . . .	476
Running a Component Instance with Purify . . . . .	477
Interpreting the Purify Log Reports . . . . .	479
Running a Component Instance without Purify . . . . .	479
Observing a Running Component Instance . . . . .	481
Rational Rose RealTime Execution Interface . . . . .	482
Target Control Programs . . . . .	482
Overriding Target Control . . . . .	482
Observability Interface . . . . .	483
Overview of Observability Options . . . . .	483
Component Instance Menu . . . . .	484
RTS Browser . . . . .	485
Execution Control and Information Pane . . . . .	486
Capsule Instance Folder . . . . .	487
Probes Folder . . . . .	487
Monitors . . . . .	488
Animation . . . . .	488
Opening a Monitor . . . . .	489
Probes . . . . .	489
Navigating to Model Elements from Debug Monitors . . . . .	490
Trace Windows . . . . .	490
Deleting Messages . . . . .	491
Trace Configuration . . . . .	491
Using Different Types of Traces . . . . .	492
Opening a Sequence Diagram . . . . .	492
Creating a Sequence Diagram From a Trace . . . . .	492

Probes . . . . .	493
Inject Window . . . . .	494
Capsule Instance Trace . . . . .	494
Trace Event Message Dialog . . . . .	494
Creating a Sequence Diagram From a Message Trace . . . . .	495
Dragging Capsule Instances into a Trace . . . . .	495
Message Trace Configuration Dialog . . . . .	496
Threshold Field . . . . .	496
Column Check Boxes . . . . .	496
Execution Watch Tab . . . . .	496
Refreshing the Watch Values . . . . .	497
Run-time Exception While Running a Component Instance . . . . .	497
Instance Browser . . . . .	498
Source Code Debugging . . . . .	498
Source Debugger Integration without Target Observability . . . . .	500
Setting Breakpoints . . . . .	500
Setting Breakpoints on State Machines . . . . .	501
Setting Breakpoints for Operations . . . . .	507
Customizing Rational Rose RealTime for Target Control and Observability	507
Running from Outside the Toolset . . . . .	508
Purify . . . . .	508
Observability Command Line Parameter . . . . .	508
Component Instance Menu . . . . .	509
Using the Command Line . . . . .	509
Command Line Arguments . . . . .	509
Application-Specific Command Line Arguments . . . . .	510
Loading and Running Component Instances on Embedded Targets . . . . .	510
Utility Scripts . . . . .	511
Component Instance Specification . . . . .	511
Component Instance Specification - General Tab . . . . .	511
Component Instance Specification - Detail Tab . . . . .	512
Overview of Observability Options . . . . .	514
Observability Options . . . . .	515



Processor Specification Dialog . . . . .	516
Processor specification - General Tab . . . . .	516
Processor Specification - Detail Tab . . . . .	516
Using Windows CE . . . . .	517
Using Debugger Modes . . . . .	521
Unloading a Debugger . . . . .	525
Device Specification . . . . .	525
General Tab . . . . .	525
Detail Tab . . . . .	526
Files Tab . . . . .	526
Connection Specification . . . . .	526
General Tab . . . . .	526
Detail Tab . . . . .	527
Files Tab . . . . .	527
Probe Specification . . . . .	527
Probe Specification - General Tab . . . . .	527
Probe Specification - Files Tab . . . . .	528
Probe Specification - Detail Tab . . . . .	528
Creating Inject Messages . . . . .	528
Examples . . . . .	529
Injecting a Message . . . . .	530
<b>24 Using Code Sync to Change Generated Code . . . . .</b>	<b>531</b>
Code Sync Overview . . . . .	531
Intended Code Sync Usage . . . . .	532
Limitations . . . . .	532
Enabling and Disabling Code Sync . . . . .	533
Identifying Code Sync Areas . . . . .	533
Code Sync Identification Tags . . . . .	533
Designated Code Sync Areas . . . . .	534
Compiling Code Externally . . . . .	535
Invoking Code Sync from the Toolset . . . . .	535
Reconciling Changes in the Code Sync Summary . . . . .	535
Accepting Changes . . . . .	536
Common Code Sync Errors . . . . .	536
Error: Cannot code-sync; file I/O error on: <filename> . . . . .	537
Error: Cannot code-sync <filename> beyond line <lineNum> . . . . .	537
Error: Could not find trailing CodeSync tag for [ <LocationSpecifier> ] . . . . .	537
Warning: Use tabs for indenting code-sync regions . . . . .	537

<b>25</b>	<b>Generating Documentation</b>	<b>539</b>
	Linking External Files to Model Elements	539
	Generate Documentation Dialog	540
	Inserting a Diagram into an MS Word Document.	541
	Option A	541
	Option B	541
	Using OLE	542
	Creating a Link	542
	Inserting a Link	542
	Navigating.	542
	Editing Diagrams	542
<b>26</b>	<b>Customizing the Toolset</b>	<b>543</b>
	Stereotypes	543
	Creating a Custom Framework for Rose RealTime Models	543
	Creating a New Stereotype for the Current Model	544
	Creating a New Stereotype Configuration File	545
	Creating a New Stereotype for all Rose RealTime Models	545
	Creating Stereotypes for Classes.	548
	Adding Stereotypes to the Diagram Toolbox	548
	Creating Stereotype Icons	548
	Creating a Diagram Icon.	549
	Controlling the Display of Stereotypes	549
	Controlling Stereotype Display in the Browser	549
	Controlling How Existing Stereotypes Display in a Diagram.	549
	Controlling the Display of Stereotypes Added to Diagrams	550
	Toolset Options	550
	Options Dialog	550
	General Tab.	551
	File Tab	552
	Font/Color Tab.	554
	Diagram Tab	555
	Filtering Tab	558
	Compartments Tab	558
	Browser Tab	559
	Editor Tab	559
	THIDC_AA1oolbars Tab	560
	Language/Environment Tab	560
	Customizing the Diagram Toolbox	561

Customize Toolbar Dialog .....	561
Toolbar Button List .....	561
Add-In Manager Dialog .....	561
Managing Model Properties .....	562
Displaying or Modifying the Values of Model Properties .....	562
Removing an Overriding Item Level Model Property .....	563
Making a Model Property Item Specific .....	563
Reinstalling the State and Value of the Last Committed Change .....	563
Attaching a Model Property Set to a Single Element or a Collection of Elements .	
563	
Displaying or Editing a Specific Model Property Set .....	564
Creating a New Model Property Set .....	564
Deleting a Model Property Set .....	564
<b>Keyboard Shortcuts .....</b>	<b>565</b>
General Shortcuts .....	565
Editing Shortcuts .....	568
Debugging Shortcuts .....	569
Build and RTS Shortcuts .....	570
Specification Code Editor Shortcuts .....	570
Browser Shortcuts .....	571
Rational Rose RealTime Keyboard Shortcut Summary .....	571
<b>Index .....</b>	<b>573</b>



# Preface

This guide describes the Graphical User Interface for the Rational Rose RealTime toolset. This guide is organized as follows:

- *Using the Online Help* on page 1
- *Overview of Rational Rose RealTime* on page 17
- *User Interface Overview* on page 29
- *Other Application Windows* on page 109
- *Printing* on page 119
- *Opening and Saving Models* on page 125
- *Use Case Diagrams* on page 137
- *Defining Use Cases and Actors* on page 141
- *Creating Class Diagrams* on page 145
- *Creating Collaboration Diagrams* on page 207
- *Creating State Diagrams* on page 229
- *Creating Activity Diagrams* on page 257
- *Creating Sequence Diagrams* on page 299
- *Defining Capsules and Classes* on page 333
- *Defining Protocols* on page 361
- *Defining Packages* on page 367
- *Creating the Component and Deployment Views* on page 377
- *Importing and Exporting* on page 385
- *Using Source Control* on page 401
- *Naming Guidelines* on page 429
- *Building and Executing Models* on page 437
- *Common Build Errors* on page 463
- *Running and Debugging* on page 475
- *Using Code Sync to Change Generated Code* on page 531
- *Generating Documentation* on page 539
- *Customizing the Toolset* on page 543
- *Keyboard Shortcuts* on page 565

## Audience

---

This guide is intended for all readers including managers, project leaders, analysts, developers, and testers.

This guide is specifically designed for software development professionals familiar with the target environment they intend to port to.

## Other Resources

---

- Online Help is available for Rational Rose RealTime.

Select an option from the **Help** menu.

All manuals are available online, either in HTML or PDF format. To access the online manuals, click **Rational Rose RealTime Documentation** from the **Start** menu.

- To send feedback about documentation for Rational products, please send e-mail to [techpubs@rational.com](mailto:techpubs@rational.com).
- For more information about Rational Software technical publications, see: <http://www.rational.com/documentation>.
- For more information on training opportunities, see the Rational University Web site: <http://www.rational.com/university>.
- For articles, discussion forums, and Web-based training courses on developing software with Rational Suite products, join the Rational Developer Network by selecting **Start > Programs > Rational Suite > Logon to the Rational Developer Network**.

## Rational Rose RealTime Integrations With Other Rational Products

---

Integration	Description	Where it is Documented
Rose RealTime–ClearCase	You can archive Rose RT components in ClearCase.	<ul style="list-style-type: none"> <li>▪ <i>Toolset Guide: Rational Rose RealTime</i></li> <li>▪ <i>Guide to Team Development: Rational Rose RealTime</i></li> </ul>
Rose RealTime–UCM	Rose RealTime developers can create baselines of Rose RT projects in UCM and create Rose RealTime projects from baselines.	<ul style="list-style-type: none"> <li>▪ <i>Toolset Guide: Rational Rose RealTime</i></li> <li>▪ <i>Guide to Team Development: Rational Rose RealTime</i></li> </ul>
Rose RealTime–Purify	When linking or running a Rose RealTime model with Purify installed on the system, developers can invoke the Purify executable using the <b>Build &gt; Run with Purify</b> command. While the model executes and when it completes, the integration displays a report in a Purify Tab in RoseRealTime.	<ul style="list-style-type: none"> <li>▪ Rational Rose RealTime Help</li> <li>▪ <i>Toolset Guide: Rational Rose RealTime</i></li> <li>▪ <i>Installation Guide: Rational Rose RealTime</i></li> </ul>
Rose RealTime–RequisitePro	You can associate RequisitePro requirements and documents with Rose RealTime elements.	<ul style="list-style-type: none"> <li>▪ <i>Addins, Tools, and Wizards Reference: Rational Rose RealTime</i></li> <li>▪ <i>Using RequisitePro</i></li> <li>▪ <i>Installation Guide: Rational Rose RealTime</i></li> </ul>
Rose RealTime–SoDa	You can create reports that extract information from a Rose RealTime model.	<ul style="list-style-type: none"> <li>▪ <i>Installation Guide: Rational Rose RealTime</i></li> <li>▪ <i>Rational SoDA User's Guide</i></li> <li>▪ SoDA Help</li> </ul>

## Contacting Rational Customer Support

---

If you have questions about installing, using, or maintaining this product, contact Rational Customer Support.

Your Location	Telephone	Facsimile	E-mail
North, Central, and South America	+1 (800) 433-5444 (toll free) +1 (408) 863-4000 Cupertino, CA	+1 (781) 676-2460 Lexington, MA	support@rational.com
Europe, Middle East, Africa	+31 20 4546-200 Netherlands	+31 20 4546-201 Netherlands	support@europe.rational.com
Asia Pacific	+61-2-9419-0111 Australia	+61-2-9419-0123 Australia	support@apac.rational.com

**Note:** When you contact Rational Customer Support, please be prepared to supply the following information:

- Your name, company name, telephone number, and e-mail address
- Your operating system, version number, and any service packs or patches you have applied
- Product name and release number
- Your Service Request number (SR#) if you are following up on a previously reported problem

When sending email concerning a previously-reported problem, please include in the subject field: "[SR#XXXXX]", where XXXXX is the Service Request number of the issue. For example, "[SR#0176528] - New data on rational rose realtime install issue".



## Contents

This chapter is organized as follows:

- *Using the Online Help System* on page 1
- *Using the Help Viewer* on page 2
- *About the Search Tab* on page 9
- *Changing the Help Viewer* on page 14
- *Related Documentation* on page 16

## Using the Online Help System

---

Comprehensive online help is available in HTML Help format. It includes four navigation tabs: **Contents**, **Index**, **Search**, and **Favorites**. There are various hyperlinks between related topics, and examples and multimedia demos for specific features of the toolset.

### Accessing What's This Help

Context-sensitive help is available for many topics from the toolset. You can access the help three ways:

- Clicking **Help > Contents**
- Selecting context-sensitive Help button on a dialog and clicking on items on the dialog box.
- Pressing SHIFT + F1 from anywhere in the toolset.

### Accessing Extended Help

Extended Help includes information on how to perform tasks using the currently selected tool, the Rational Unified Process (RUP), and other sources of information. The information accessed from Extended Help depends on the context from which it is invoked. The context varies from task to task.

Extended Help is based on a set of databases that identify the context for the information (called the target). After you install, you have access to the RUP database. You can create and register any number of databases, each containing different information and pointers to different HTML pages. For example, you may have a database that contains pointers to pages that are appropriate for all projects in your organization. You may have another that is specific to a project or project type. You can also create a personal database with pointers to information that is important to you.

## Tutorials

The online help includes tutorials to help you learn to use the main features of Rational Rose RealTime.

## Using the Help Viewer

---

The following topics describe most of the general features available in the Help Viewer:

- *Getting More Out of Help* on page 2
- *Creating a List of Favorite Help Topics* on page 3
- *Copying a Help Topic* on page 4
- *Printing the Current Help Topic* on page 5
- *Obtaining Help in a Dialog* on page 5
- *Finding Topics Using the Toolbar Buttons* on page 5
- *Hiding or Showing the Navigation Pane* on page 6
- *Using Accessibility Shortcut Keys in the Help Viewer* on page 6
- *Using the Context Menu Commands* on page 9

## Getting More Out of Help

To find more information when using the HTML Help Viewer, you can:

- Link to another topic, a Web page, a list of other topics, or a program, by clicking the colored, underlined words.
- View topics that contain related information by clicking topic titles under the headings **Related Topics**, **Related Tasks**, and **See Also**, which may appear at the end of a topic.
- Verify if a word or phrase contained in a topic is in the **Index** by selecting the word, and then pressing F1.

- Click **Stop** or **Refresh** on the Help toolbar to interrupt a download to refresh a Web page.
- Add a frequently used topic to your **Favorites** list. For important information on how to use the **Favorites** list, see the file `rosert_readme.html`.
- Right-click the **Contents** tab or **Topic** pane for context menu commands.

## Finding a Help Topic

In the **Navigation** pane, click one of the following tabs:

- To browse through a table of contents, click the **Contents** tab. The table of contents is an expandable list of important topics.
- To see a list of index entries, click the **Index** tab, and then type a word or scroll through the list. Topics are often indexed under more than one entry.
- To locate every occurrence of a word or phrase that may be contained in a help file, click the **Search** tab, and then type the word.

### Notes

Click on a contents entry, index entry, or search results entry to display the corresponding topic.

## Creating a List of Favorite Help Topics

**To quickly create a list of Favorite help topics:**

- 1 Locate the help topic you want to make a favorite topic.
- 2 Click the **Favorites** tab, and then click **Add**.

**Note:** Since the online Help system is modular, adding a help topic to a favorites lists only adds that topic to favorites list for the currently open help component. To create a single favorites list, see *Maintaining a Single Favorites List* on page 3.

## Maintaining a Single Favorites List

The Rational Rose RealTime online Help system is modularized. Consequently, if you select **Add** on the **Favorites** tab to add current help topic to your favorites list, this entry will only appear in the favorites list for that component of the online help.

### **To maintain a single list of "favorite" help topics:**

- 1 Use the **Search** or **Index** tabs to find the desired online help topic.
- 2 Click the **Locate** button in the Toolbar of the Online Help window to see where this help topic appears in the **Contents** tab.
- 3 Close the online help.
- 4 Open the online help, using **Help > Contents**.
- 5 From the **Contents** tab only, find the help topic.
- 6 Click the **Favorites** tab.
- 7 Click **Add**.

### **Notes**

- To return to a favorite topic, click the **Favorites** tab, select the topic, and then click **Display**.
- If you want to rename a topic, select the topic, and then type a new name in the **Current topic** box.
- To remove a favorite topic, select the topic and then click **Remove**.

## **Copying a Help Topic**

### **To copy a Help topic:**

- 1 In the **Topic** pane, right-click the topic you want to copy, and then click **Select All**.
- 2 Right-click again, and then click **Copy** to copy the topic to the Clipboard.
- 3 Open the document you want to copy the topic to.
- 4 Position your cursor where you want the information to appear.
- 5 On the **Edit** menu, click **Paste**.

### **Notes**

If you want to copy only part of a topic, select the text you want to copy, right-click, and then click **Copy**.

## Printing the Current Help Topic

Right-click a topic in a Help window, and then click **Print**.

### Notes

If you print from the **Contents** tab (by right-clicking an entry, and then clicking **Print**) you will see options to print only the current topic, or the current topic and all of its subtopics.

## Obtaining Help in a Dialog

Click the question mark in the upper-right corner of the dialog, and then click an item in the dialog.

### Notes

- To close the pop-up window, click anywhere on the screen.
- If the dialog does not have the question mark, click **Help** or press F1.
- You can also obtain help on an item by right-clicking it.
- Not all dialog boxes include dialog-level help.

## Finding Topics Using the Toolbar Buttons

The following navigational buttons appear on the Toolbar in the Help Viewer:

- **Back** displays the last topic you viewed.
- **Forward** displays the next topic in a previously displayed sequence of topics.
- **Home** displays the Home page topic for the help file you are viewing.
- **Refresh** updates Web content that is currently displayed in the Topic pane.
- **Stop** terminates the downloading of file information.

### Notes

The toolbar in your Help Viewer may not contain all of these navigational buttons.

## Hiding or Showing the Navigation Pane

On the Toolbar in the Help window, click **Hide** or **Show** to close or display the **Navigation** pane, which contains the **Contents**, **Index**, **Search**, and **Favorites** tabs.

### Notes

If you close the **Help Viewer** with the Navigation pane hidden, it appears minimized the next time you open the Help Viewer.

## Using Accessibility Shortcut Keys in the Help Viewer

You can use the following keyboard shortcuts to navigate in the **HTML Help Viewer**:

To	Press
Close the Help Viewer.	ALT+F4
Switch between the Help Viewer and other open windows.	ALT+TAB
Display the Options menu.	ALT+O
Change Microsoft Internet Explorer settings. The <b>Internet Options</b> dialog box contains accessibility settings. To change these settings click the <b>General</b> tab, and then click <b>Accessibility</b> .	ALT+O, and then press I
Hide or show the Navigation pane.	ALT+O, and then press T
Print a topic.	ALT+O, and then press P, or right-click in the
Move back to the previous topic.	ALT+LEFT ARROW, or ALT+O, and then press B
Move forward to the next topic (provided you have viewed it just previously).	ALT+RIGHT ARROW, or ALT+O, and then press F
Turn on or off search highlighting.	ALT+O, and then press O
Refresh the topic that appears in the Topic pane (this is useful if you have linked to a Web page).	F5, or ALT+O, and then press R
Return to the home page (help authors can specify a home page for a help system).	ALT+O, and then press H
Stop the viewer from opening a page (this is also useful if you are linking to the Web and want to stop a page from downloading).	ALT+O, and then press S

Jump to a predetermined topic or Web page. The help author who builds a compiled help (.chm) file can add two links, on the **Options** menu, to important topics or Web pages. When you select a **Jump** command you go to one of those topics or Web pages. ALT+O, and then press 1 or 2

Switch between the Navigation pane and the Topic pane. F6

Scroll through a topic. UP ARROW and DOWN ARROW, or PAGE UP and PAGE DOWN

Scroll through all the links in a topic or through all the options on a Navigation pane tab. TAB

### **For the Contents tab:**

<b>To</b>	<b>Press</b>
Display the <b>Contents</b> tab.	ALT+C
Open and close a book or folder.	PLUS SIGN and MINUS SIGN, or LEFT ARROW and RIGHT ARROW
Select a topic.	DOWN ARROW and UP ARROW
Display the selected topic.	ENTER

### **For the Index tab:**

<b>To</b>	<b>Press</b>
Display the <b>Index</b> tab.	ALT+N
Type a keyword to search for.	ALT+W, and then type the word
Select a keyword in the list.	UP ARROW and DOWN ARROW
Display the associated topic.	ALT+D

### **For the Search tab:**

<b>To</b>	<b>Press</b>
Display the <b>Search</b> tab.	ALT+S
Type a keyword to search for.	ALT+W, and then type the word
Start a search.	ALT+L

Select a topic in the results list. ALT+T, and then UP ARROW and DOWN ARROW

Display the selected topic. ALT+D

**The following options are only available if full-text search is enabled.**

Search for a keyword in the result list of a prior search. ALT+U

Search for words similar to the keyword. For example, to find words like “running” and “runs” for the keyword “run.” ALT+M

Only search through topic titles. ALT+R

### **For the Favorites tab:**

<b>To</b>	<b>Press</b>
-----------	--------------

Display the <b>Favorites</b> tab.	ALT+I
-----------------------------------	-------

Add the currently displayed topic to the Favorites list.	ALT+A
--	-------

Select a topic in the Favorites list.	ALT+P, and then UP ARROW and DOWN ARROW
---------------------------------------	---

Display the selected topic.	ALT+D
-----------------------------	-------

Remove the selected topic from the list.	ALT+R
--	-------

### **Notes**

- You can also access context menu commands using the keyboard.
- Shortcut keys also work in secondary and pop-up windows.
- If you use a shortcut key in the Navigation pane, you lose focus in the Topic pane. To return to the Topic pane, press F6.
- The **Match similar words** box on the **Search** tab is selected if you used it for your last search.



## Using the Context Menu Commands

There are several commands on the context menu that you can use to display and customize information.

Command	Description
Right-click in the table of contents, and then click <b>Open All</b> .	Opens all books or folders in the table of contents. This command only works if the <b>Contents</b> tab is displayed.
Right-click in the table of contents, and then click <b>Close All</b> .	Closes all books or folders. This command only works if the <b>Contents</b> tab is displayed.
Right-click, and then click <b>Print</b> .	Prints the topic.
Right-click in the table of contents, and then click <b>Customize</b>	Opens the <b>Customize Information</b> wizard, which allows you to customize the documentation. If the help file was built with information types, you can use this wizard to select a subset of topics to view. For example, you could choose to see only overview topics.

### Notes

- You can click SHIFT+F10 to display the context menu, and then click the appropriate shortcut keys, or you can enable Mousekeys. Use a Mousekey combination to display the context menu, and then click the appropriate shortcut keys.

## About the Search Tab

---

The **Search** tab allows you to search through every word in a help file to find a match. For example, if you perform a full-text search on the word "generate", every topic that contains the word "generate" displays.

### To use full-text search:

- 1 Click the **Search** tab, and then type the word or phrase you want to find.
- 2 Click **List Topics**, select the topic you want, and then click **Display**.

### **To highlight words in searched topics:**

When searching for words in help topics, each occurrence of the word or phrase appears highlighted in the topics that were found.

To highlight all instances of a search word or phrase, click **Options** on the Toolbar, and then click **Search Highlight On**.

### **Notes**

- To disable this option, click **Options** on the toolbar, and then click **Search Highlight Off**.
- When viewing a long topic, only the first five hundred instances of a search word or phrase are highlighted.

## **Searching for Help Topics**

A basic search consists of the word or phrase you want to find. To further refine your search, you can use wildcard expressions, nested expressions, boolean operators, similar word matches, a previous results list, or topic titles.

The basic rules for formulating queries are:

- Searches are not case-sensitive.
- You can search for any combination of letters (a-z) and numbers (0-9).
- The search ignores any punctuation marks, such as a period, colon, semicolon, comma, and hyphen.
- To set apart each search element, group the elements of your search using double quotation marks or parentheses.

**Note:** You cannot search for quotation marks.

### **Notes**

When searching for a file name with an extension, group the entire string in double quotation marks, for example, use "filename.ext". Otherwise, the period is interpreted as a separation character and the search will attempt to find two separate terms. The default operation between terms is **AND**, so the search uses the logical equivalent to "filename AND ext."

### To find information with advanced full-text search:

- 1 Click the **Search** tab, and then type the word or phrase you want to find.
- 2 Click to add boolean operators to your search.
- 3 Click **List Topics**, select the topic you want, and then click **Display**.
- 4 To sort the topic list, click the **Title**, **Location**, or **Rank** column heading.

### Notes

- You can precisely define a search by using wildcard expressions, nested expressions, and boolean operators.
- You can request similar word matches, search only the topic titles, or search the results of a previous search.
- You can set the Help Viewer to highlight all instances of search terms that are found in topic files. Click **Options**, and then click **Search Highlight On**. This feature only works with Internet Explorer 4.0 or later.

## Searching for Words or Phrases

You can search for words or phrases and use wildcard expressions. Wildcard expressions allow you to search for one or more characters using a question mark or asterisk. The table below describes the results of these different kinds of searches.

Search for	Example	Results
A single word	select	Topics that contain the word "select". (You will also find its grammatical variations, such as "selector" and "selection".)
A phrase	"new operator" or new operator	Topics that contain the literal phrase "new operator" and all its grammatical variations.  Without the quotation marks, the query is equivalent to specifying "new <b>AND</b> operator", which will find topics containing both of the individual words, instead of the phrase.
Wildcard expressions	esc* or 80?86	Topics that contain the terms "ESC", "escape", and "escalation". The asterisk cannot be the only character in the term.  Topics that contain the terms "80186", "80286", "80386", and so on. The question mark cannot be the only character in the term.

## Notes

- Select **Match similar words** to include minor grammatical variations for the phrase you search.

## Defining Search Terms

The **AND**, **OR**, **NOT**, and **NEAR** operators enable you to precisely define your search by creating a relationship between search terms. The following table shows how you can use each of these operators. If no operator is specified, **AND** is used. For example, the query "spacing border printing" is equivalent to "spacing **AND** border **AND** printing."

Search for	Example	Results
Both terms in the same topic.	dib <b>AND</b> palette	Topics containing both the words "dib" and "palette."
Either term in a topic.	raster <b>OR</b> vector	Topics containing either the word "raster" or the word "vector" or both.
The first term without the second term.	ole <b>NOT</b> dde	Topics containing the word "OLE" but not the word "DDE."
Both terms in the same topic, close together.	user <b>NEAR</b> kernel	Topics containing the word "user" within eight words of the word "kernel."

## Notes

- The "|", "&", and "!" characters do not work as boolean operators (you must use **OR**, **AND**, and **NOT**).

## Using Nested Expressions when Searching

Nested expressions allow you to create complex searches for information. For example, "control **AND** ((active **OR** dde) **NEAR** window)" finds topics containing the word "control" along with the words "active" and "window" close together, or containing "control" along with the words "dde" and "window" close together.

The basic rules for searching help topics using nested expressions are as follows:

- You can use parentheses to nest expressions within a query. The expressions in parentheses are evaluated before the rest of the query.
- If a query does not contain a nested expression, it is evaluated from left to right. For example: "Control **NOT** active **OR** dde" finds topics containing the word "control" without the word "active," or topics containing the word "dde." On the other hand, "control **NOT** (active **OR** dde)" finds topics containing the word "control" without either of the words "active" or "dde."
- You cannot nest expressions more than five levels deep.

#### **To search for words in the titles of HTML files:**

- 1 Click the **Search** tab.
- 2 Type the word or phrase you want to find.
- 3 Select the **Search titles only** option.
- 4 Click **List Topics**, select a topic from the list, and then click **Display**.

#### **Notes**

- All HTML topic files are searched, including any that are not listed in the **Contents** tab.

#### **To find words similar to your search term:**

This feature enables you to include minor grammatical variations for the phrase you search. For example, a search on the word "add" finds all references to "add," "adds," and "added."

- 1 Click the **Search** tab, type the word or phrase you want to find, and then select the **Match similar words** check box.
- 2 Click **List Topics**, select the topic you want, and then click **Display**.

#### **Notes**

- This feature only locates variations of the word with common suffixes. For example, a search on the word "add" finds "added," but it will not find "additive."

## Searching within Search Results

This feature enables you to narrow a search that results in too many topics found. You can search through your results list from previous search by using this option.

### To search the results from a previous search:

- 1 On the **Search** tab, select the **Search previous results** option.
- 2 Click **List Topics**, select the topic you want, and then click **Display**.

### Notes

- To search through all of the files in a Help system, ensure that you clear this option.
- The **Search** tab opens with the **Search previous results options** selected if you previously used this feature.

## Changing the Help Viewer

---

You can make various modifications to the Help Viewer. You can modify the following:

- *Customizing the Help Viewer* on page 14
- *Changing Format or Styles for Accessibility* on page 15
- *Viewing Topics Grouped by Information Type* on page 15
- *Changing the Font Size of a Topic* on page 15
- *Changing Colors in the Topic Pane of the Help Viewer* on page 16

## Customizing the Help Viewer

You can change the size and position of the Help Viewer and the panes in the Help Viewer by doing the following:

- To resize the Navigation or Topic pane, move your mouse to point to the divider between the two panes. When the pointer changes to a double-headed arrow, drag the divider right or left.
- To proportionately shrink or enlarge the entire Help Viewer, move your mouse to point to any corner of the Help Viewer. When the pointer changes to a double-headed arrow, drag the corner.

- To change the height or width of the Help Viewer, move your mouse to point to the top, bottom, left, or right edge of the Help Viewer. When the pointer changes to a double-headed arrow, drag the edge.
- To reposition the Help Viewer on your screen, click the Title Bar and drag the Help Viewer to a new location.

#### **Notes**

- The Help Viewer will appear with the last size and position settings you specified when it is opened again.

### **Changing Format or Styles for Accessibility**

- 1 On the **Options** menu, click **Internet Options**, and then click **Accessibility**.
- 2 In the **Accessibility** dialog box, and select any desired options.
- 3 Click **OK**.

#### **Notes**

- These changes do not apply to the Navigation pane or toolbar of the Help Viewer.
- This will also change your accessibility settings for Internet Explorer 4.0.

### **Viewing Topics Grouped by Information Type**

You can customize your help system so that it includes only those help topics that are relevant to you.

For example, if you have a help system for an educational software program that includes topics for administrators, teachers, and students, you can customize your Help so that it includes only the topics that are important to teachers and students.

To group your information by type, on the Toolbar, click **Options**, and then click **Customize**.

### **Changing the Font Size of a Topic**

On the **Options** menu, click **Internet Options**, and then click **Fonts**.

#### **Notes**

- These changes do not apply to the Navigation pane or toolbar of the Help Viewer.
- This setting also changes your font settings for Internet Explorer.

## Changing Colors in the Topic Pane of the Help Viewer

- 1 In Microsoft Internet Explorer, on the **View** menu, click **Internet Options**.
- 2 On the **General** tab, click **Colors**.
- 3 In the **Colors** dialog box, select the options you want, and then click **OK**.
- 4 To apply the new color settings, in the **Internet Options** dialog box, click **OK**.

### Notes

- These changes do not apply to the Navigation pane or toolbar of the Help Viewer.
- This setting also changes your color settings for Internet Explorer 4.0.

## Related Documentation

---

The following documents are related to the *Rational Rose RealTime Toolset Guide*:

- *Installation Guide, Rational Rose RealTime*
- *Modeling Language Guide, Rational Rose RealTime*
- *Guide to Team Development Guide, Rational Rose RealTime*



## Contents

This chapter is organized as follows:

- *Developing Using Rational Rose RealTime* on page 17
- *Using Languages and Code Generation* on page 18
- *Using the Services Library* on page 19
- *Capsules, Protocols, Ports, Capsule State and Structure Diagrams* on page 19
- *Constructing Models in Rational Rose RealTime* on page 22
- *Development Process* on page 25
- *Essential Workflows* on page 26

## Developing Using Rational Rose RealTime

---

Rational Rose RealTime is a software development environment tailored to the demands of real-time software. Developers use Rational Rose RealTime to create models of the software system based on the Unified Modeling Language constructs, to generate the implementation code, compile, then run and debug the application.

You can use Rational Rose RealTime through all phases of the software development lifecycle; from initial requirements analysis through design, implementation, test and final deployment. It provides a single interface for model-based development that integrates with other tools required during the different phases of development. For example, developers work directly through Rational Rose RealTime to generate and compile the code that implements the model. The actual compilation is performed behind the scenes by a compiler/linker outside of the toolset.

Using Rational Rose RealTime, developers work at a higher level of abstraction specifying behavior in state diagrams and communication relationships in collaboration diagrams. This is a natural and logical evolution in computer languages. Just as third generation language tools provided greater productivity than assembly language coding, visual development tools provide significant productivity gains over current third generation languages.

Rational Rose RealTime includes features for:

- Creating UML models using the elements and diagrams defined in the UML
- Generating complete code implementations (applications) for those models
- Executing, testing and debugging models at the modeling language level using visual observation tools
- Using Change Management systems for team development

## Using Languages and Code Generation

---

Code generation of models is provided by specialized language add-ins. The content of the generated code, regardless of the language, is based on the specification of each model element, and the values of the model properties attached to model elements.

Language add-ins provide custom properties that store language-specific information for each model element. In addition to Rational Rose, Rational Rose RealTime add-ins also provide a Services Library that provides support for specialized real-time services, such as concurrency, message passing, and timing services. The Services Library is included as a library file and is linked into every executable created with Rational Rose RealTime.

By providing both code generation and the specialized Services Libraries, Rational Rose RealTime lets you generate, compile, and run models.

### Compiling Models

You compile models generated from Rational Rose RealTime using commercial compilers and linkers. Rational Rose RealTime generates the code, and then calls the specified compiler and linker to compile and link the generated source code with the pre-compiled Services Library.

**Note:** Rational Rose RealTime does not include a compiler or linker. You must first install a compiler and linker before you can build and run a model.

## Using the Services Library

---

To construct a functioning Rational Rose RealTime model, at a minimum, you will require a defined structure and behavior for the model, and the Rational Rose RealTime Services Library.

The Services Library is essentially a framework for real-time systems. It includes functionality for controlling concurrent execution of finite state machines, for delivering messages, and for providing timing and logging services. A framework is similar to a library of classes and operations used by an application, but with an inversion of control. This means that the main control lies in the framework, and the framework invokes functions in the application to pass control to application objects, as required. Application classes are sub-classed from framework classes so that they inherit certain operations.

There is no **main()** function in a Rational Rose RealTime model. The **main()** function is contained in the Services Library and handles the creation of capsules in your model, as well as starting the execution of their state machines. After you describe the capsules and define state machines for them, they are automatically created and executed by the Services Library. The capsule state machines can, in turn, invoke operations on other classes (data classes), and send messages to other capsules. The Services Library is responsible for managing the creation and destruction of capsules, and the delivery of messages between capsules (including messages across threads).

The addition of these real-time notations to the UML concepts allows the toolset to generate complete code for the model which is tied in to the Services Library. When you generate code and compile a model in Rational Rose RealTime, the toolset will link it with a Services Library that was compiled for the specific language and platform you use.

## Capsules, Protocols, Ports, Capsule State and Structure Diagrams

---

In addition to supporting the core UML constructs, Rational Rose RealTime uses the extensibility features of the UML to define some new constructs that are specialized for real-time system development. These new constructs allow code generation of elements that can use the services provided in the Services Library, such as concurrent state machines, concurrency, message passing, and timing services.

In fact, many real-time projects must implement most of the above services. Using the added modeling elements in Rational Rose RealTime allows you to concentrate on implementing the functionality of the system right away without having to hand-code the common real-time services and concurrency support.

## Capsules

- A capsule is a stereotype of a class.

Has much of the same properties as regular classes with added semantics for modeling of communication relationships between capsules and modeling of its event based behavior using a state diagram.

- Provides built-in support for light weight concurrent objects.

Because of the message based nature and high encapsulation of capsules, they can be easily distributed to different physical threads of control without any change to the capsule.

- Highly encapsulated objects using message based communication to other capsules via its port objects.

The advantage of the message-based interfaces is that a capsule has no knowledge of its environment outside of these interfaces, making it a much more **distributable**, **reusable**, and **robust** than regular objects.

- Capsule can aggregate other capsules.

Like classes, a capsules structure is defined by its attributes (encapsulation of objects of other types of classes). But it can also be defined by attributes that are other capsules, which we call **capsule roles**.

## Protocols

- Defines the set of messages exchanged between a set of capsules.
- Messages are defined from the perspective of both the receiver and the sender.

There are therefore different perspectives of a protocol, which we call protocol roles. Protocol roles represent the communication from the perspective of one participant in the communication scenario.

- Messages that are sent between capsules contain a required signal name (which identifies the message), an optional priority (relative importance of this message compared to other unprocessed messages on the same thread), and optional application data.

## Ports

- Ports are objects whose purpose is to send and receive messages to and from capsule instances.
- They are owned by the capsule instance in the sense that they are created along with their capsule and destroyed when the capsule is destroyed.
- To specify which messages can be sent to and from a port, a port realizes a protocol role. The protocol role is the specification of a set of the messages that can be received (in) and sent (out) from the port. The protocol role essentially defines the port type.

## State Diagrams

- Uses the same notation as defined in the UML.
- Are generated to source code and make up the behavior of capsules.
- All trigger events are defined by a port and signal pair. A capsule's behavior is therefore based on the receipt of messages.
- Final states are not allowed on capsule state diagrams.
- Junction points do not support the continuation kind attribute; that is, if a transition is not continued, it defaults to history (except for internal transitions).

## Capsule Structure Diagrams

- A new diagram has been introduced to specify the capsule's interface (ports) and its internal composition (capsule roles). The diagram is called a capsule structure diagram, and it is based on the UML 1.3 specification collaboration diagram.
- This is a specification type of diagram, and not an interaction diagram (object) as collaboration diagrams in other versions of Rose are.
- Allow you to specify the communication relationships between capsules.

## Executable Models

The addition of the capsule and the formal semantics surrounding the capsule structure allows Rational Rose RealTime to generate, compile and run a complete implementation based on a model containing capsules.

The ability to execute models has a revolutionary impact on the software development process. The results are higher quality software, and shorter and more predictable delivery cycles. Executing models is the surest way to find problems and issues that whiteboarding and document reviews do not find. Even high-level architectural models can be executed.

Use model execution to better understand the problem, to detect errors and problems in requirements and architecture specifications, to explore alternative designs quickly, and to test design models continuously during the development process.

**Process note:** To make the best use of Rational Rose RealTime, you should aim to get your model running as often as possible. Making small, incremental changes and running your model each day will bring much better results than making widespread changes and working for weeks to get the model running again.

## Constructing Models in Rational Rose RealTime

---

This section describes:

- Modeling Elements
- Diagrams
- Development Process
- Essential Workflows

### Modeling Elements

There are many different modeling elements supported in Rational Rose RealTime, and it is not easy for new users to know which elements to use to accomplish their goals. In practice, there are only a few elements that are required to construct a running model. The other elements provide greater flexibility and control over the expression of your design model.

As you have already seen from the overview description of the Rational Rose RealTime services library, the services library is essentially a framework for executing capsules. Capsules are the main initiators and controllers of activity in a Rational Rose RealTime model. You must define at least one capsule, but typically many more, in order to generate and compile code for the model. The capsules in the model should contain other classes, usually referred to as data classes, because they must be invoked from a capsule behavior before they can perform any action. Also, they are primarily used by capsules to contain detail data, which is operated on within the capsule.

### Required Elements

First of all, there are two elements that most developers will make use of at the start of a project to understand the problem domain and begin the process of turning the vague problem descriptions into detailed designs and implementations. These two elements are use cases and actors. They are used together (along with use case diagrams) for use case modeling, which helps architects, designers, testers, and others

involved in the project understand the original system requirements and relate those requirements to elements in the design model. Use cases and actors are not strictly required to create an executable model, but use case modeling is highly recommended as an effective method in the overall analysis and design process.

Your model then must consist of capsule classes, and protocols, which specify the messages that capsules use to communicate. Almost all models also contain some data classes for capsules to use to store and operate on detailed data. Larger models contain many hundreds or thousands of capsule, protocol and data classes. These models should contain packages to organize the classes into related units.

A capsule must have a state machine defined for it in order to perform any useful behavior. The state machine defines the set of valid inputs that can be processed by the capsule. A complete code implementation is generated for capsule state machines, and any user-defined code to be performed as actions on state transitions is embedded in the generated code.

In any non-trivial model, some capsules will have structure defined for them that describes the interfaces which capsules use to communicate with each other. These interfaces are called ports. The structure also describes how capsules are contained by other capsules to construct composite systems. When one capsule is contained by another capsule, it is referred to as a capsule role.

Before you can compile a model you must create a component that describes which classes should be compiled as a unit, and the various settings that should be used to control the code generation, compilation and link processes.

Finally, in order to run your model, you must specify the deployment by adding a processor to the deployment diagram. The processor specifies the processing node (workstation or embedded target) on which your model will be executed. The compiled component must be mapped as a component instance on the processor so that the tool can load the compiled component onto the specified processor for execution and observation.

### **Further Reading**

These are all the model elements that are required in order to create an executable model in Rational Rose RealTime. Each of the possible elements in the model is described further in the *Modeling Language Guide*.

## Diagrams

Diagrams are an essential part of the model. Diagrams describe how the different elements are combined together to make up the system. They also specify other forms of relationships among the model elements.

There are eight diagrams supported in the Rational Rose RealTime tool, not all of which are required to create an executable model. Although not all diagrams are required, they exist for a purpose: the combination of these diagrams provides an excellent description of the total composition and behavior of the model.

The supported diagrams are:

- use case diagrams
- class diagrams
- state diagrams
- collaboration diagrams
- capsule structure diagrams
- sequence diagrams
- component diagrams
- deployment diagrams

Of these eight diagrams, only class diagrams, state diagrams, and capsule structure diagrams are essential in the development of an executable model.

The capsule structure diagrams describe the composition and connectivity of the capsules in the model. This is essential in the creation of anything more than the most trivial model. The generated code for the capsules reflects the information in the capsule structure diagram.

State diagrams must be created for each capsule that has any significant functionality. The tool will generate the implementation code for capsule state diagrams (no code is generated for state diagrams for other classes), and the capsule state diagrams provide the starting point for all behavior in the model. Class diagrams are used to define inheritance relationships between classes (capsule, protocol and data classes can all be subclassed). Class diagrams can also be used to show other associations among classes. The class diagrams may result in code being generated to implement the relationships defined in the diagrams, depending on the detailed settings for those relationships.

Component diagrams are used to specify the parameters for compilation of the model. In more complex systems, a hierarchy of components is compiled to make an executable.

Deployment diagrams must also be used to get a model running. The deployment diagram specifies how the model will be deployed on the destination hardware.



## Further Reading

Each of these diagrams is explained in more detail in the *Modeling Language Guide*. The instructions for creating the diagrams are contained in the individual chapters for the diagrams in this guide.

## Development Process

---

The Rational Rose RealTime toolset is oriented to the use of an iterative, object-oriented development process. However, detailed description of the development process is beyond the scope of this document. We strongly recommend that you look at the Rational Unified Process (RUP) to gain a better understanding of the iterative object-oriented development process. See <http://www.rational.com>. At a more detailed level, the process of creating an executable model in Rational Rose RealTime can be summarized as follows:

Use the use case modeling elements and use case diagram to develop a detailed, semi-formal understanding of the problem. The use case elements can be associated with design elements as the design model evolves to maintain traceability.

Create capsules, protocols, classes, use class diagrams, capsule structure diagrams, and capsule state diagrams to develop the structure and behavior of the model. Add detailed implementation code to the capsule state diagrams and to class operations.

In addition, use collaboration diagrams and sequence diagrams to capture the intended behavior of the system for various use cases. Use Rational Rose RealTime's execution and debugging tools to validate the model behavior at run-time. Use collaboration and sequence diagrams to help you in the design process by making the communication patterns in the design evident. They will also help others understand your design.

Once the design has stabilized, use state diagrams for classes and protocols to capture the abstract design so that others can understand. This is particularly important for protocols, where the state machine specifies how a capsule using that protocol must behave.

Use the component diagram to specify the configuration of the model for compilation purposes.

Use the deployment diagram to indicate how the components should be executed. Also, the deployment diagram can be used to document the physical structure of the target system.

## Further Reading

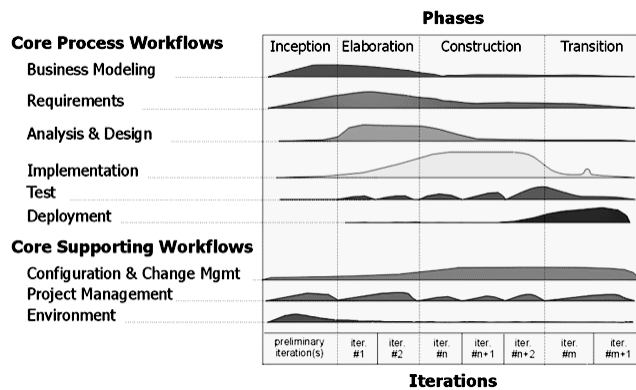
An overview of the Rational Unified Process is available in the Online Help book: Development process reference. The complete Rational Unified Process description is available at <http://www.rational.com>.

## Essential Workflows

---

The following chart describes the workflows and project phases defined in the Rational Unified Process.

**Figure 1 Workflows in the Rational Unified Process**



Rational Rose RealTime is not applicable to all of these workflows. The following workflows are important in the context of Rational Rose RealTime:

**Requirements** - are typically captured in text documents or databases outside of the Rational Rose RealTime toolset. Some analysis of the requirements is performed to develop a more abstract model of the problem. The abstract model is called a use case model. The Rational Rose RealTime use case modeling tools are used in this workflow. In addition, design model elements can be traced back to requirements in two ways: through associations between use case model elements and design model elements captured in class diagrams, and through linking external files (requirements specifications, for example) to model elements.

**Analysis & Design** - is the primary workflow supported in the Rational Rose RealTime toolset. All of the Rational Rose RealTime class and capsule modeling tools are used in the analysis and design. There is no clear distinction between analysis and design in the Rational Rose RealTime toolset. They are part of the same process, which is the process of turning vague problem descriptions into specifications of software-based solutions. The end goal of this process is a design model, which in

Rational Rose RealTime is complete enough to be executable. Execution of the design model is used as the basis for verifying whether the design meets the requirements. Intermediate artifacts such as design documents can be produced from the model.

**Implementation** - in a traditional development process without Rational Rose RealTime, there is typically a gulf between analysis & design and implementation. In implementation, developers take the design specifications and produce code to implement those specifications. The mapping is not always straightforward, and the design specifications may be vague, incomplete or erroneous, leading to confusion, lost productivity and time delays. With Rational Rose RealTime, the implementation is automatically produced directly from the model. Implementation details are still necessary, but they are added directly within the model framework. Going back-and-forth is not required to keep the model in sync with the implementation. It is always in sync.

**Test** - testing in Rational Rose RealTime involves compiling a model and running it. A number of tools are provided in the run-time interface to assist with testing.

**Configuration and Change Management** - this workflow is ongoing, and is essential for orderly development in a large team environment. Configuration and change management involves putting the model and model elements under source control. See the *Guide to Team Development*.

### **Further Reading**

Each of these workflows and the impact of the tools in Rational Rose RealTime on those workflows is described in the Online Help.



## Contents

This chapter is organized as follows:

- *Startup Screen* on page 29
- *Create New Model Dialog* on page 30
- *Application Window* on page 37
- *The Toolbar* on page 39
- *Menus* on page 43
- *Browsers* on page 78
- *Diagram Editors* on page 82
- *Specification Dialogs* on page 92
- *Searching and Sorting* on page 104

## Startup Screen

---

The Startup screen has direct links to:

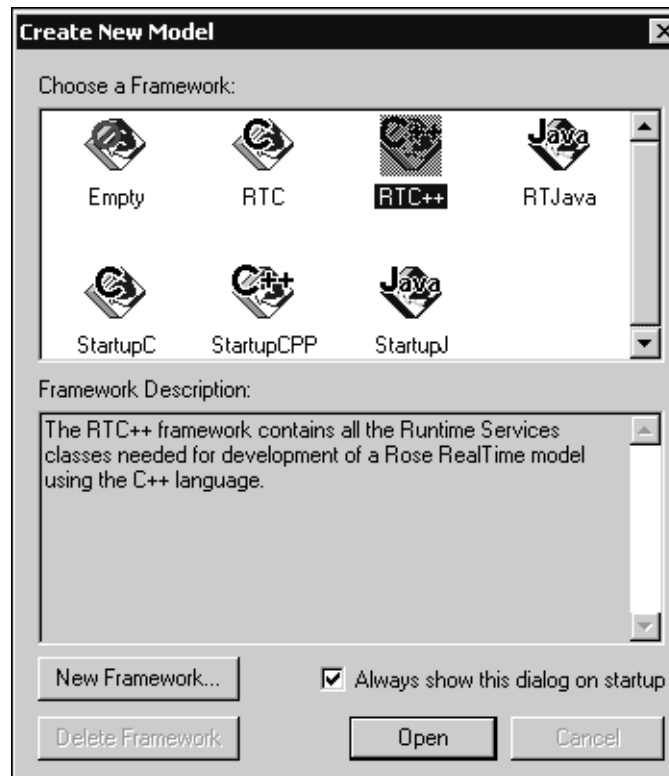
- **What's New?** - To review the new features of this release of Rational Rose RealTime
- **Tutorials** - To review various tutorials tailored to your skill level and backgrounds
- **Online Help** - To access the Rational Rose RealTime help system

You have the option of always showing this screen on startup or bypassing it.

## Create New Model Dialog

---

When you start Rational Rose RealTime, the **Create New Model** dialog appears.



In the **Create New Model** dialog, you have the following frameworks available:

- **Empty** - Lets you to open a model without any shared package, which is useful for pure modeling and use case development. The Empty framework allows the you to open a model without any shared packages that provide language-specific Runtime Services.

**Note:** The **Empty** framework is useful for creating use case designs but should not be used for developing real-time applications.

- **RTC** - Lets you to create a model in the C language. The RTC framework contains all the Runtime Services classes needed for development of a Rational Rose RealTime model using the C language.
- **RTC++** - Lets you to create a model in the C++ language. The RTC++ framework contains all the Runtime Services classes needed for development of a Rational Rose RealTime model using the C++ language.

- **RTJava** - Lets you to create a model in the Java language. The RTJava framework contains all the Runtime Services classes needed for development of a Rational Rose RealTime model using the Java language. The Java Runtime Services classes utilize the Java language classes, therefore, these classes may also appear in the framework.
- **StartupC** - Provides a framework containing two example C language Hello World executables; one using the common **main** function, and the other using a capsule with a trivial state machine. For additional information on the Startup frameworks, see *Using the Startup Frameworks* on page 31.
- **StartupCPP** - Provides a framework containing two example C++ language Hello World executables; one using the common **main** function, and the other using a capsule with a trivial state machine. For additional information on the Startup frameworks, see *Using the Startup Frameworks* on page 31.
- **StartupJ** - Provides a framework containing two example Java language Hello World executables; one using the common **main** function, and the other using a capsule with a trivial state machine. For additional information on the Startup frameworks, see *Using the Startup Frameworks* on page 31.

**Note:** Any framework model that you create as a template appears in the **Create New Model** dialog. For information on creating a framework, see *Creating a Custom Framework for Rose RealTime Models* on page 543.

To open Rational Rose RealTime without automatically displaying the **Create New Model** dialog box, click **File > New > Create New Model**, and then clear the **Always show this dialog on startup** option.

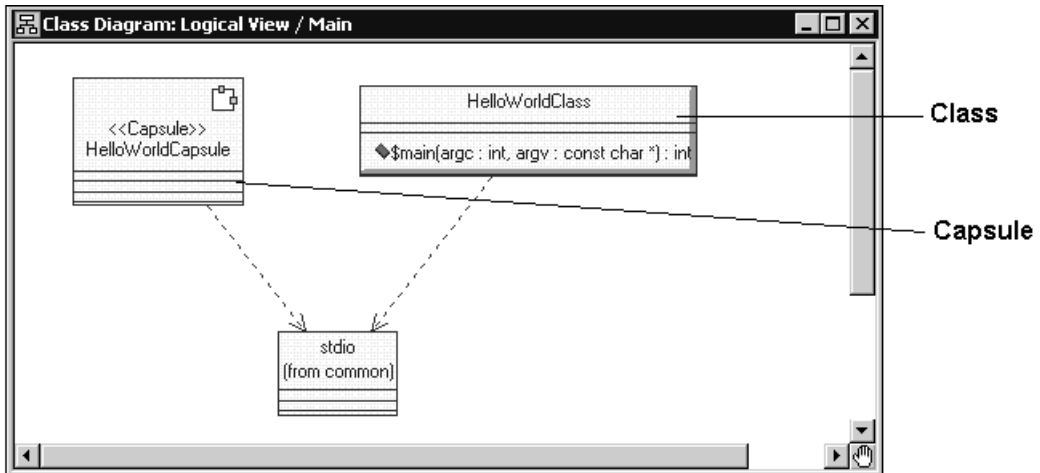
## Using the Startup Frameworks

There is one executable Hello World Startup Framework provided for the C, C++, and Java languages. Each Startup framework demonstrates an executable Hello World application using the **main** function, and using a capsule with state machine.

**Note:** This topic describes the Startup framework for the C++ language. The C and Java frameworks are similar.

Each framework has a main **Class** diagram (Figure 2).

Figure 2 Class Diagram - Main



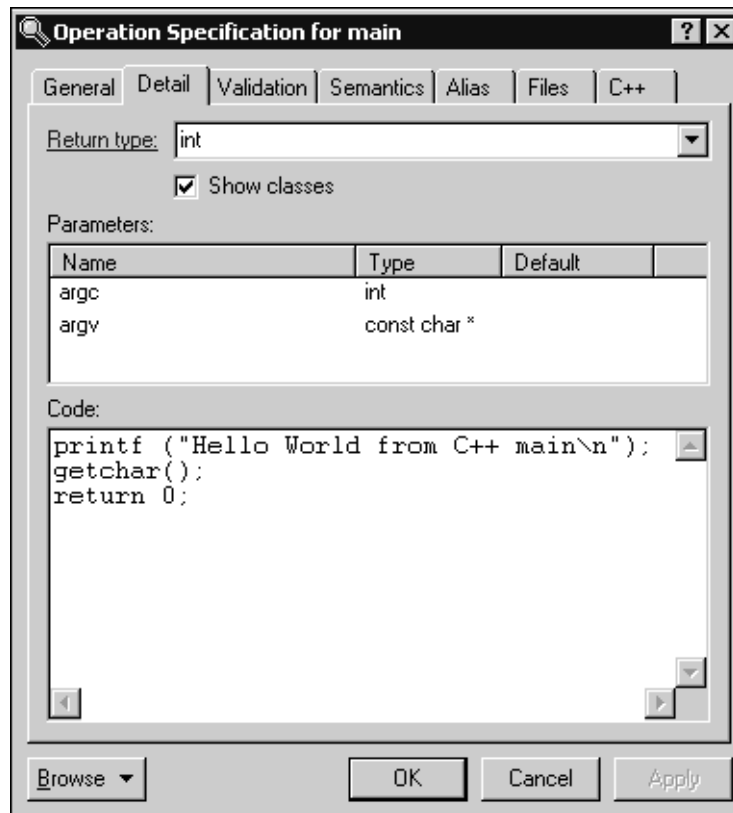
The simplest Hello World example is implemented by the class `HelloWorldClass`. The class and the capsule depend on `stdio`.

**Note:** The external file has been modeled to make it visible in the design.

Figure 3 shows the **Operation Specification** dialog for the `main` function for the `HelloWorldClass` class.



Figure 3 Operation Specification Dialog For the Main Operation

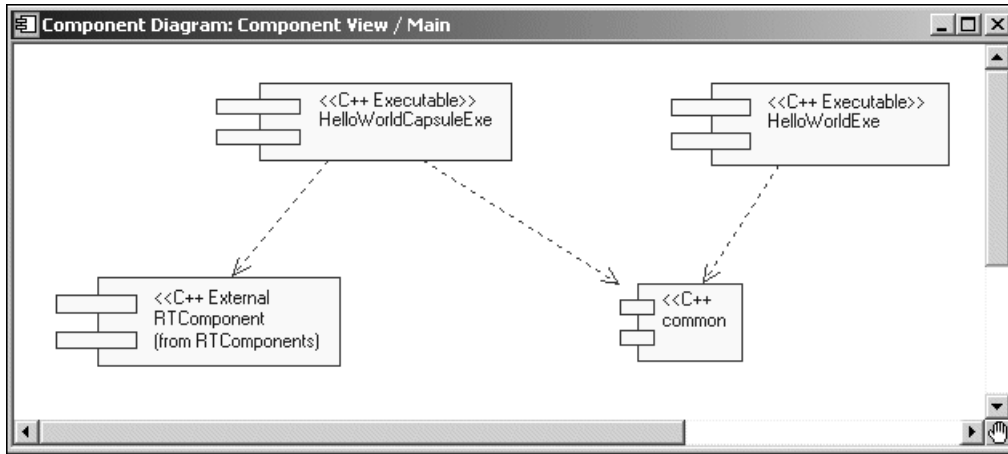


The generated code for the **main** operation will look like the following:

```
int main( int argc, const char * argv )
{
printf ("Hello World from C++ main\n");
getchar();
return 0;
}
```

Figure 4 shows the **Component View** for this Hello World startup model. The Component diagram describes the build structure.

**Figure 4 Component Diagram**



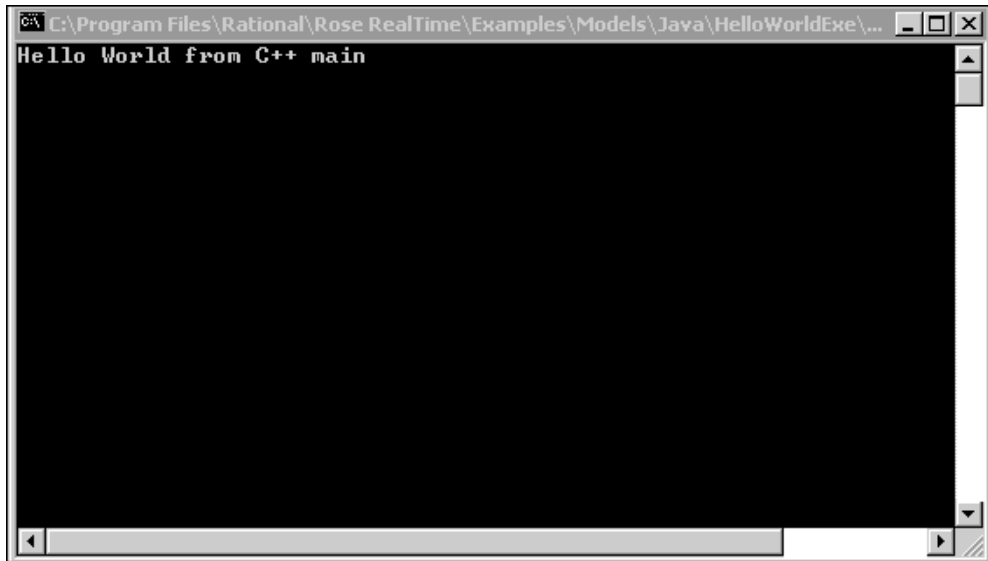
To generate the corresponding code, right click on the component in the diagram or in the model browser and click **Build > Build**, and then select **Generate and Compile**.

**Note:** Before you build, you will need to open the **Specification** dialog for the component and configure it for your compiler.

You can run the generated executable from the command-line, or you can start it from within the modeling environment. To run the executable, from the **Deployment View** on the **Model View** tab in the browser, right-click on the component instance called **HelloWorldExeInstance**, and click **Run**.

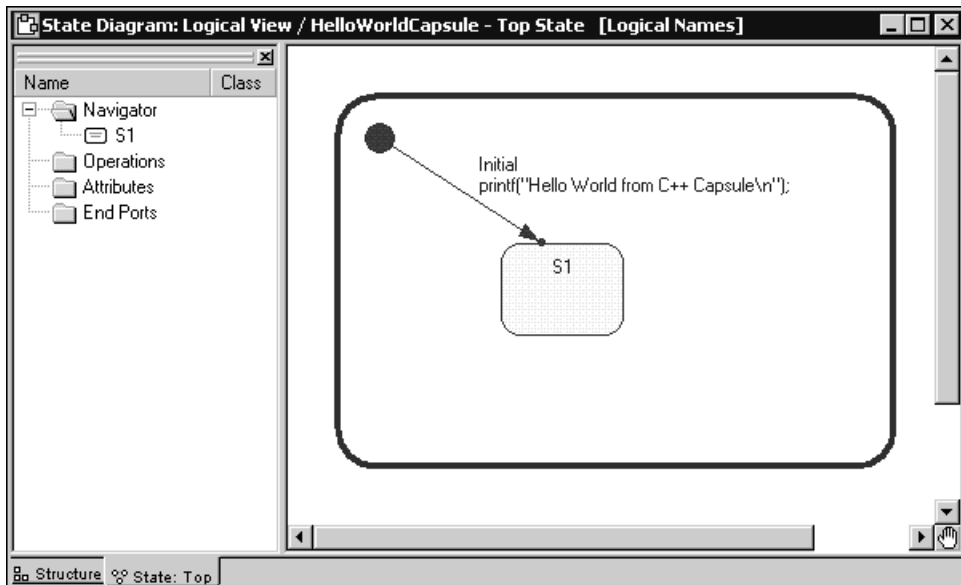
Figure 5 shows the results that appear in the console when you run the class-based executable.

**Figure 5 Console Output**



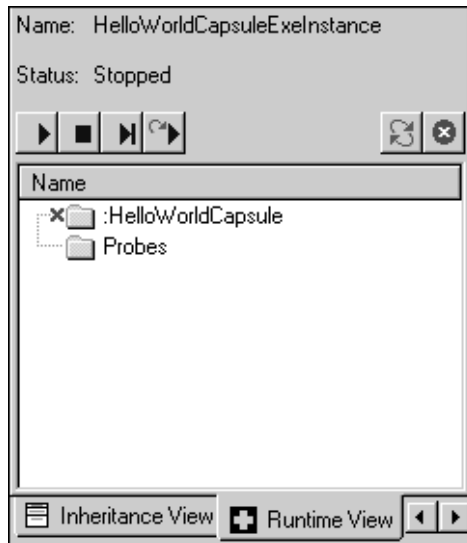
In the capsule-based model, the code is in the initial transition of the state machine. You can inspect the code on the **State** diagram using the **Code** tab, or by opening the **Transition Specification** dialog.

**Figure 6 State Diagram for HelloWorldCapsule**



If you run the capsule-based executable, the **Runtime View** debugger will appear.

**Figure 7 Runtime View Tab**




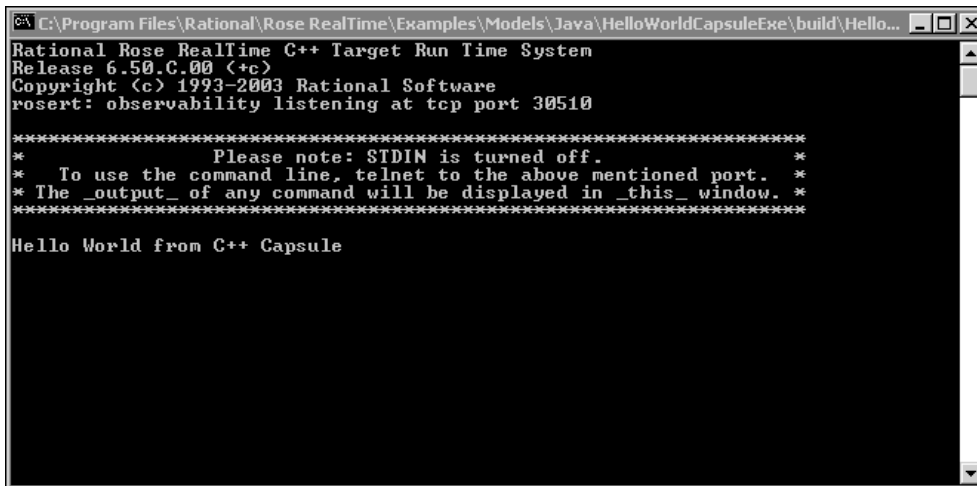
Click **Run** (  ) to start the executable.

Figure 8 shows the results that appear in the console when you run the capsule-based executable.

**Figure 8 Console Results**



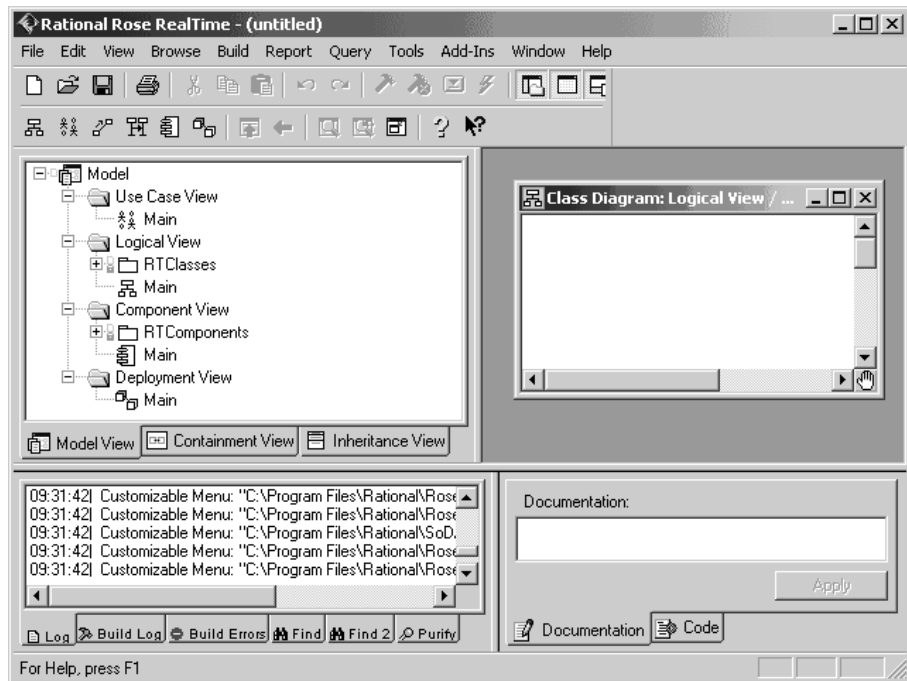
# Application Window

The main elements of the Rational Rose RealTime user interface are:

- The Toolbar
- Menus
- Browsers
- Diagram Editors
- Specification Dialogs

Figure 9 shows an RTC++ example of the application window as it appears when the application is first started (with no model loaded).

**Figure 9 Application Window**



## Browsers

Model elements are created and viewed through browsers. The primary tab in the browser is the **Model View** tab, that provides access to all elements of the current model. Browsers list the model elements - usually in a hierarchical way - allowing elements to be expanded to show additional information. Also, most browser lists can be filtered in various ways.

The **Containment View** shows the containment hierarchy of the capsule classes in the model.

The **Inheritance View** shows the inheritance hierarchy of the capsule classes, data classes, and protocols in the model.

## Toolbar

There is a standard toolbar (see *The Toolbar* on page 39) in the main application window that contains icons for a standard set of tools, which can be invoked at any time. This toolbar can be undocked and moved as a separate window.

## Diagrams

Much of the model information is captured graphically in Diagram Editors. Additional non-graphical information, such as detailed program code, can be entered through Specification Dialogs. In many cases, information can be entered through a diagram or specification dialog, or Code window.

Each diagram or specification that you open is displayed in a window within the application window. These diagram and specification windows can be iconified. If the active window displays a diagram, that diagram is referred to as the current diagram. If the active window displays a specification, that specification is referred to as the current specification.

## Toolboxes

Every diagram has an associated toolbox (see *Toolboxes* on page 91), that contains icons of tools that can be applied to that diagram. If the current diagram is write-protected, the diagram toolbox is not displayed. The diagram toolbox is dimmed when a diagram is displayed.

## Menu Bar

The menu bar lists commands available for operations in any diagram or specification window. Depending on the kind of diagram or specification displayed in the active window, some menu commands may not apply. These commands are dimmed. If the current diagram is write-protected, additional commands are rendered inaccessible.



### **Open Existing Model**

Opens the Load Model dialog. If you have a model open when you select the open model, you are asked if you want to save the current model. Selecting **No** discards all changes since your last save. Selecting **Yes** saves your changes and opens a new model. See *Opening Models* on page 129 for more information.

### **Save Model**

Opens the Save Model to dialog. Enter a new filename. After the model is named and saved, selecting this button automatically saves your changes to the current model without displaying the dialog.

### **Print Diagram**

Opens the Print Specification dialog, that allows you to specify how and where diagrams are printed. To change printer setup select **File > Print Setup**.

### **Cut**

Removes icons or relationships from your model. An item (or items) must be selected to activate the icon. Cutting an element also cuts associated relationships. You can cut multiple-selected items.

### **Copy**

Copies a component to a new location of the same model - or a new model - without affecting the original component.

### **Paste**

Pastes a component, that has previously been cut or copied to the clipboard, to another location.

### **Undo**

Undoes the last operation performed. Not all operations can be undone. If the Undo tool is dimmed, the last operation cannot be undone.

### **Redo**

Redoes the last operation that was undone.



### **Build Component**

Initiates a model verification, generates the source code for the component, and invokes the external compiler and linker to create an executable version of the component. Only the model elements that have changed will be generated and recompiled.

### **Stop Build**

Stops a build in progress.

### **Load Process**

Loads the components instances specified in the Build Settings dialog. The component must be successfully built before it can run.

If the Attach Target observability flag was set on the Component Instance Specification dialog and a Target observability Port number filled in, the execution interface is displayed allowing you to control the execution of the model.

### **Run Component Instance**

Loads the component instances specified in the Build Settings Dialog. The component must be successfully built before it can run.

### **View Browser(s)**

Displays or hides all existing browsers. Existing browsers are those that have been created, but not closed.

### **View Description**

Displays the Description Window, that contains the Documentation Tab and the Code Tab.

### **View Output**

Displays the Output window.

### **Browse Class Diagram**

Opens a class diagram in the Class diagram editor (see *Using the Class Diagram Editor* on page 146). Selecting this command opens a dialog allowing you to select from available class diagrams to open.

### **Browse Use Case Diagram**

Opens a Use Case diagram in the Use Case diagram editor (see *Using the Use Case Diagram Editor* on page 138). Selecting this command opens a dialog allowing you to select from available use case diagrams to open.

### **Browse Collaboration Diagram**

Opens a collaboration diagram using either the Collaboration diagram editor (see *Using the Collaboration Diagram Editor* on page 222) or the capsule structure Editor (see *Using the Structure Editor* on page 208). Selecting this command opens a dialog allowing you to select from available collaboration diagrams to open.

### **Browse Sequence Diagram**

Opens a sequence diagram in the sequence diagram editor (see *Using the Sequence Diagram Editor* on page 305). Selecting this command opens a dialog allowing you to select from available sequence diagrams to open.

### **Browse Component Diagram**

Opens the component diagram in the Component diagram editor (see *Using the Component Diagram Editor* on page 377).

### **Browse Deployment Diagram**

Opens the deployment diagram in the Deployment diagram editor (see *Using the Deployment Diagram Editor* on page 380).

### **Browse Parent**

Displays the "parent" of the selected diagram or specification. If you have a specification selected, the specification for the parent of the "named" item is displayed.

### **Browse Previous Diagram**

Displays the last displayed diagram. To go back more than one diagram, repeatedly press the **Browse Previous Diagram** button.

### **Fit in Window**

Centers and displays any diagram within the limits of the window. This command changes the zoom factor so that the entire diagram displays.

This command does not change the state of the diagram. Changes made after clicking **Fit In Window** may require that you re-click **Fit In Window** to center and resize the diagram again.

### **Undo Fit in Window**

Reverts the diagram and window sizing back to its appearance prior to the **Fit in Window** operation.

### **Scale to Fit**

Scales the diagram to fit within the current diagram window geometry.

### **Help Contents**

Activates the online help system.

### **Context Sensitive Help**

Activates the online help system and opens the help about that particular topic.

## **Menus**

---

This section provides information on the following topics:

- *Menu Bar* on page 44
- *File Menu* on page 45
- *Edit Menu* on page 49
- *Parts Menu* on page 54
- *View Menu* on page 56
- *Browse Menu* on page 57
- *Build Menu* on page 61
- *Report Menu* on page 64
- *Query Menu* on page 66
- *Tools Menu* on page 68
- *Add-Ins Menu* on page 75
- *Window Menu* on page 76
- *Help Menu* on page 77

## Menu Bar

The menu bar provides drop-down menus for all operations on models and model elements.

Some menu items are context-sensitive, and only operate when certain types of model elements are selected. Context-sensitive menu items are grayed-out when they are not applicable.

**Figure 11 Main Menu Bar**



The menu bar contains the following menus:

- File Menu
- Edit Menu
- Parts Menu
- View Menu
- Browse Menu
- Build Menu
- Report Menu
- Query Menu
- Tools Menu
- Add-Ins Menu
- Window Menu
- Help Menu

Not all menus are displayed at all times. Some of these menus appear only in context of particular diagrams. The **Parts** menu, for example, appears only when a capsule structure diagram or state diagram is open.

## File Menu

The operations available on the **File** menu may vary according to the current active window or the type of element selected.

### File Menu Operations

#### New

Opens the **Create New Model** dialog. There are four frameworks listed: **Empty**, **RTC**, **RTC++**, and **RTJava**. Additionally, any framework model that you create to be used as a template, appears in the dialog.

To create a new Rational Rose RealTime model containing all the classes required for development for the C, C++, or Java language, click the framework for the specified language. The **Model** browser appears with the packages and classes populated in the **Logical View** and **Component View**.

For information on creating a framework, see *Creating a Custom Framework for Rose RealTime Models* on page 543.

**Note:** The **Empty** framework is useful for creating use case designs but should not be used for developing RealTime applications.

A new model is unnamed until it is saved by a **Save** or **Save As** command.

Any open models are closed before a new model is created. You are prompted to save changes if necessary.

By default, a new model contains one empty class diagram - the main class diagram for the top level of the new model. You should place packages and classes representing your highest-level abstractions in this diagram. The new model is automatically created as a controlled unit.

#### Open

Loads a model or model kernel from a model file. A file browser is opened to let you to select a .rtmdl file. If there is an accompanying workspace (.rtwks) file, the tool asks if you want to open it instead.

Any open models are closed before opening another model. You are prompted to save changes if necessary.

If the selected model file's access control in the platform file system is read-only, the application write-protects the associated model.

When opening the model, the tools checks whether the model's saved character set is the same as the current system default charSet. If not, a dialog displays the following warning:

```
"Non system default character set in file."
```

See *Opening Models* on page 129.

### **Open Workspace...**

Opens an existing workspace. A file browser is opened allowing you to select a .rtwks file.

### **Save Workspace**

Saves the workspace. This action creates four files: a .rtwks file containing configuration management settings; a .rtmdl file containing the representation of the model itself; a .rtto file containing Target observability items, including probes and inject messages; and a .rtusr file containing various application settings.

### **Save Workspace As...**

Saves the workspace. This action creates four files: a .rtwks file containing configuration management settings; a .rtmdl file containing the representation of the model itself; a .rtto file containing Target observability items, including probes and inject messages; and a .rtusr file containing various application settings. A file browser is presented to specify the file name and location.

### **Save Model**

Saves the model. Writes the model out as a .rtmdl file. If the model has not been saved before, a file browser is presented to specify the file name and location.

If a destination model file's access control in the platform file system is read-only, an ERROR! dialog is displayed indicating the software cannot write that file.

To control temporary and backup files created during a Save procedure, refer to the Customizing the Diagram Toolbox in the **Tools > Options** dialog.

### **Save Model As...**

Saves the model. Writes the model out as a .rtmdl file. A file browser is displayed to specify the file name and location. This command displays the **Save Model To** dialog, in which you can specify the new file name and the location where you want to save the current model.

## **Import...**

Imports a model file created by another tool. See *Importing a Petal or Package File* on page 385 for more information.

This command displays the Read Petal dialog so you can specify the petal file you want to import. Use this command to import the contents of a petal file into the current model. This command requires the active window to contain a class or component diagram.

When you import a petal file that contains elements, the diagram in your active window is used to select a destination. If the active window is the top level diagram, these elements are imported into the top level of the current model. Otherwise, these elements are imported into the package that encloses the diagram in the active window.

Each imported element is compared to the corresponding element in the current model. If any of the elements in the petal file already exist in the current model, error messages are sent to the log. If an imported package contains elements that already exist in the current model, a dialog is displayed telling you that these elements have not been imported. All diagrams in the model are appropriately updated, including those imported from the petal file.

When you import a petal file that contains a complete model, that model is opened. If a model is already open with unsaved changes, the Save Confirmation dialog is displayed, prompting you to save your changes before closing the current model and opening the model contained in the petal file.

## **Export Model...**

Exports model files in alternative formats. Primarily used to exchange models with other versions of Rational Rose and other Rational Rose tools.

This command displays the Write Petal dialog so that you can specify the name and location of the petal file. Use this command to export selected items from the current model to a petal file. The Export command displays the name of the element type selected. If nothing is selected, Export Model is displayed. You can export:

- The entire model
- Classes
- Logical Packages
- Component Packages

Begin by displaying diagrams containing the items you want to export. Select the specific classes, logical and component packages to be exported, and pull down the **File** menu. The Export command indicates the number of items selected. If no items are selected, the entire model is exported. If any item not on the above list (such as a relationship or adornment) is selected, the Export command is not executable.

When a logical or component package is exported, all diagrams it contains are also exported. When individual classes are exported, however, only their state transition diagrams are exported with them. Exporting the entire model does export all diagrams contained within it.

Exporting to a petal file is useful when you want to transfer:

- elements from one model to another
- a model or its elements between different computing platforms
- a model or its elements to a new software release

### **Print...**

Prints the model. *Printing* on page 119

### **Print Setup...**

Changes the print setup before printing.

### **Edit Path Map**

Edits or creates pathmap variables. Opens the Virtual Path Map dialog. Using the dialog, you can create an entry to represent a mapping between a virtual path symbol and an actual pathname. This feature allows you to work with models moved or copied among workspaces and archives by redefining the actual directory associated with the user-defined symbol.

### **Recent Files**

Opens recently edited models.

### **Recent Workspaces**

Opens recently used workspaces.

### **Exit**

Exits the application.



## Edit Menu

### Undo

Undoes the last operation. Some operations may not be undone. If Undo is not possible, the **Undo** menu item is grayed-out.

### Redo

Redoes the last undone operation.

### Cut

Removes the selected item and places it in the buffer. When you cut an item, all relationships for that item are also cut. For example, if A is a generalization of B and you cut A, the generalization is also cut.

This command works only on the graphic representation of a diagram. It does not change the current model. Not all elements can be cut and pasted. If after selecting an element or group of elements the **Cut** menu item is grayed-out, the cutting and pasting of one or more of those elements is not supported; for example, you cannot cut and paste multiple elements in a capsule state diagram.

### Copy

Copies the selected item into the buffer. Use this command to copy the currently selected item or items to the clipboard. From the clipboard, you can

- paste items into other diagrams
- paste items into documents you create with any standard word-processing software

The Copy command provides a simple means of importing a class from one package to another.

If a relationship is copied and your selection does not include the items at both ends of that relationship, you cannot paste that relationship. In this circumstance a Warning dialog appears, allowing you to cancel or continue. You can also disable subsequent warnings for the remainder of your session.

This command works only on the graphic representation of a diagram. It does not change the current model.

### Paste

Pastes whatever is currently in the buffer into the selected destination (the active window).

## Delete

Deletes the currently selected item(s) from a diagram or specification.

When used in a Class diagram, the Delete command removes each selected icon from the current diagram. The model is not changed unless the deleted icon is unnamed.

In the structure and behavior diagrams of a capsule, delete always deletes the model object.

When you delete an item, all relationships associated with that item are also deleted.

In collaboration and interaction diagrams, you are prevented from deleting an icon representing a component or relationship if there are no other icons representing that component or relationship in the active diagram. In a collaboration diagram, you are prevented from deleting an icon representing an object if that object has one or more links, and you are prevented from deleting a link if that link has one or more messages. In all of these cases, you can use the Delete from Model command.

## Duplicate

Duplicates a selected model item into the package that owns the diagram and validates its name in the context.

## Select All

Selects all elements on the current diagram or all text in the current editor.

This command is useful when you want to:

- Generate reports on all the classes in a single diagram using commands on the **Report** menu
- Populate a class diagram with all of the information about those classes using commands on the **Query** menu
- Change the font size or characteristics for all of the text in a diagram using commands on the **Options** menu

To deselect all items, click at a point on the diagram that is not already highlighted.

**Note:** The following two items are only available from Sequence diagrams.

## Attach Text Label

Attaches Text Label to a graphic element.

## Detach Text Label

Detaches Text Label from a graphic element.

## **Delete from Model**

Deletes the selected item(s) from the model.

This command can be used from diagrams to delete items from the current model. When you delete an item from the model, all icons representing that item are removed from any diagrams in which they appear. The specification for the item is also deleted.

This command cannot be used in specifications. To delete an item from the model via a specification, use the Delete command.

## **Relocate**

As the analysis and design of an application proceeds, it is common to refine the application's logical and/or physical architecture from one iteration to the next. Such refinements can include:

- Relocating a class or logical package from one logical package to another
- Relocating a component or component package from one component package to another

The Relocate command supports these refinements. It allows you to relocate a model element (class, component, package, or association) to a new logical or component package.

The component will now be contained by the component package containing the current diagram. However, relocating a component or component package has no effect on any diagram in the model.

## **Diagram Object Properties**

Lets you customize various software features. The characteristics you set via this menu item affect only the selected icon(s).

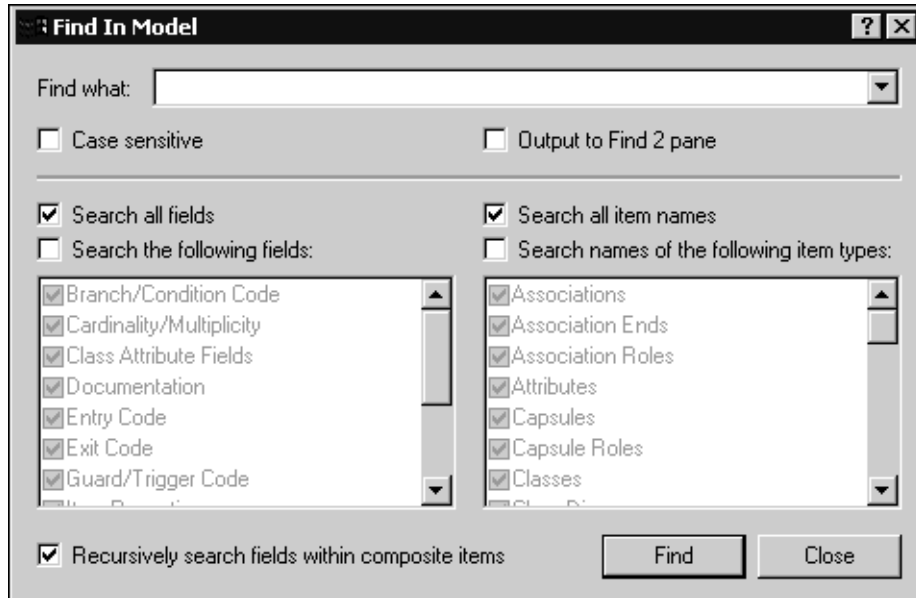
## **Line Attributes...**

Lets you select between rectilinear or oblique line styles, and routing styles. You can undo and redo any changes you make. Select **Line Attributes > Edit...** to edit the properties of line attributes.

## **Find...**

Finds all references to a specified item in the model.

**Figure 12 Find in Model Dialog**



The **Find in Model** dialog provides a drop-down list of your previous search strings, if any. Select an existing search string or specify the search string to find. The search results appear in the **Find** tab of the **Output** window (click **View > Output**)

**Note:** You can select **Output to Find 2 pane** on the **Find in Model** dialog to display the search results in the **Find 2** tab.

**To search for groups of items or to search code, you can use the wildcard character (\*):**

- **A\*** matches any name beginning with the letter A
- **\*A** matches any name ending with the letter A
- **\*A\*** matches any name containing the letter A

This command lists only the first 250 items that match a name containing a wildcard character. To search for **""**, use **"\\*"**.

The **\*** wildcard is especially useful for finding any classes or packages that were automatically renamed in a model that was upgraded from a previous release. For example, you can search for every model item that was renamed by typing: **\*#\*** (star pound star). Every model item that has a **#** in its name is found.

This command searches for the named item and displays a list of diagrams in which that item appears. You can double-click on an entry to display that diagram.

## Replace...

Use this command to find and replace any item in the model. This command displays the Replace Dialog so that you can type the search string and the replace with string.

The dialog provides drop-down lists of previous search and replace strings. Select an existing search or replace string or type the names of the item to find and replace. The search result is displayed in the Find tab of the Output window (**View > Output**) in a three-column list, including the name, type, and location. Optionally, you can choose to have the results displayed in the Find 2 tab.

When you click Replace, a secondary dialog appears with options to Find Next, Replace, Replace All, and Cancel. After the replace operation has taken place, you can query the Find tab in the Output window to view the results of the search and replace.

## Reassign...

Each icon in a diagram represents an element in the current model. Use this command to make a selected icon represent a model element other than the one it now represents. For example, you can assign an existing class to a different diagram element.

This feature is useful if you want to assign an element to use the same named item from a different name space.

To make an icon represent another element, select the icon and then click Reassign from the **Edit** menu. The dialog lists the packages in the model on the left and a list of the valid elements to choose from on the right. Choose the model element that the selected icon will represent. This affects only the selected icon. Other icons representing the original model element (on all diagrams) maintain their original representation.

Here is an example: Assume you have three classes named car, buggy and wagon. With buggy selected, click **Edit > Reassign**. In the dialog select wagon. You are changing the underlying model of the diagram element buggy to use wagon instead of buggy. (Buggy may still exist in the model but you are given the option of deleting it.) Additionally, if buggy had an inheritance relation drawn to car, then wagon adopts that inheritance relation.

The Reassign command does not work within Capsule Collaboration (Structure) diagrams.

## Compartment...

Opens the Edits Compartments dialog. This dialog allows you to arrange how attributes and operations are displayed within a class or package icon on a diagram.

## Change Into

Changes the selected model element into another (related) kind of modeling element.

In the process of refining your model, you may find it necessary to change a model element from one kind to another. For example, you may want to change a class into a capsule once you have decided that the class has a state machine and a logical thread of control.

You can use the commands on the **Change Into** submenu to change a model element from one type to another. You can transform

- A class into another type of class
- A relationship into a different type of relationship

You can also transform an element as follows:

- 1 Choose the icon on the diagram toolbox.
- 2 Press the ALT or META key.
- 3 Click on the element you want to change.

This command changes the model element and updates all diagrams containing this element.

When a relationship type is changed, this command removes the original relationship from all diagrams, but does not automatically add the new relationship to these diagrams. Use the Filter Relationships command to display the new relationship in specific diagrams.

## Parts Menu

**Note:** The **Parts** menu is only available on capsule structure and state diagrams.

### Edit Inside

Shows the internal composition of a contained state or capsule role's class. Allows you to perform edits (with some limitations) on elements contained inside the selected state or capsule role without having to open a new editor. This is most useful for seeing the internal connections of transitions to substates and of relay ports to other contained capsule roles.

## **Remove/Exclude**

Removes a local or exclude an inherited element from the current class. For example, if you are creating a subclass of an existing capsule class, and the superclass defines a state that is not applicable in the subclass, you may remove it in the subclass diagram using the Remove/Exclude command. Any excluded elements still appear in the navigator area of the structure or behavior editor, but have the symbol x beside them to indicate that they have been removed.

## **Inherit**

Causes a previously excluded element to be reinherited. Reinherited elements are added back to the diagram.

## **Aggregate**

Applies only to states or capsule roles. For states, the aggregate command creates a new composite state containing the selected states to be aggregated. For capsule roles, the aggregate command creates a new capsule class to hold the capsule roles that are being aggregated. This command also replaces the aggregated capsule roles in the structure diagram where the command was executed with a single capsule role of the newly-created capsule class.

## **Decompose**

Applies only to composite states or capsule roles that are aggregates. Breaks the selected state into its immediate substates, or breaks the selected capsule role into the capsule roles contained within the aggregate capsule class.

## **Promote**

Moves the selected element up in the class hierarchy. The element is moved into the immediate superclass and is inherited by the subclasses (including the current class). If there is any name conflict between this element and another element in the superclass or in any of its subclasses, the promote command fails.

## **Demote**

Moves the selected element down in the class hierarchy. The element is removed from the current class, and is pushed down into all immediate subclasses, as if it had been defined locally on the subclasses.

### **Lock Position(s)**

Locks the element in position on the diagram. Once the element is locked it cannot be moved around the diagram unless it is unlocked.

### **Unlock Position(s)**

Allows the element to be moved around within the diagram.

## **View Menu**

### **Toolbars**

Toggles the display of the The Toolbar and Toolboxes.

### **Status Bar**

Toggles the display of the status bar at the bottom of the window, which provides textual information about selected items and current operations.

### **Browsers**

Toggles the display of all application browsers, as well as create a new one.

### **Description**

Toggles the display of the Description Window, which contains the Documentation Tab and the Code Tab.

### **Output**

Toggles the display of the output window.

### **Specification History**

Records Specification dialogs opened from within toolset, allows you to easily navigate between the dialogs, and provides a mechanism to quickly close these dialogs.

### **Filter**

Filters label information on diagrams.

### **Zoom**

Zooms in on the current diagram. Select the zoom level.



## Scale to Window

Scales the current diagram down to fit entirely within the current diagram window border. Scales according to the outer boundaries of the diagram - for example, the outer state border of the state diagram - and not simply the area around visible diagram elements.

## Page Breaks

Toggles the visual indication of where page breaks appear on diagrams when printed. The printer specified in the Printer Setup determines the exact location of page breaks. You can also change this setting through the rose.ini file.

## Refresh

Redraws the current diagram.

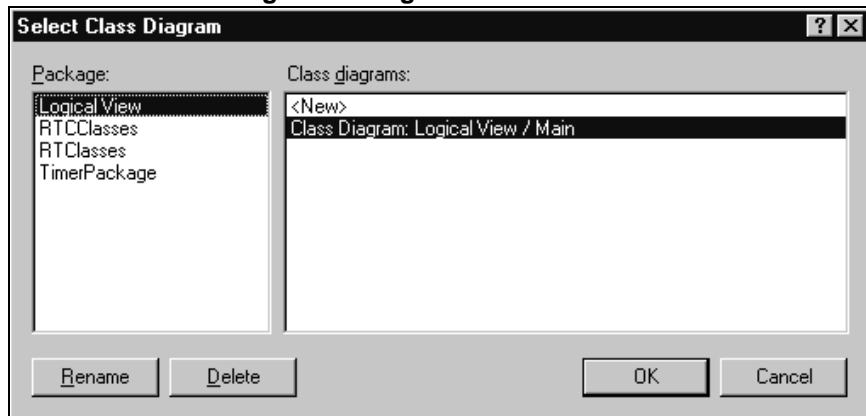
## Browse Menu

Use commands on the **Browse** menu to navigate through the diagrams and specifications that represent your model.

## Select Diagram Dialog

When you select a diagram type from the **Browse** menu, a dialog appears for that type of diagram. For instance, when you select **Browse > Class Diagram...** the Select Class Diagram dialog appears (see Figure 13).

**Figure 13** Select Class Diagram Dialog



Using commands from the dialog, you can display, rename, create, and delete diagrams.

## **Browse Menu Operations**

### **Class Diagram...**

Opens a Select Class Diagram dialog, allowing you to select a diagram to open, or to create a new diagram. The dialog also allows you to rename or delete diagrams.

### **Use Case Diagram...**

Opens a Select Use Case Diagram dialog, allowing you to select a diagram to open, or to create a new diagram. The dialog also allows you to rename or delete diagrams.

### **Collaboration Diagram...**

Opens a Select Collaboration Diagram dialog, allowing you to select a diagram to open, or to create a new diagram. The dialog also allows you to rename or delete diagrams.

### **Sequence Diagram...**

Opens a Select Sequence Diagram dialog, allowing you to select a diagram to open, or to create a new diagram. The dialog also allows you to rename or delete diagrams.

### **Component Diagram...**

Opens a Select Component Diagram dialog, allowing you to select a diagram to open, or to create a new diagram. The dialog also allows you to rename or delete diagrams.

### **Deployment Diagram...**

Opens a Select Deployment Diagram dialog, allowing you to select a diagram to open, or to create a new diagram. The dialog also allows you to rename or delete diagrams.

### **State Diagram**

This menu item is only activated when you have selected a capsule, class, or protocol on a diagram. Use this command to display the state diagrams associated with the selected class or protocol in a class diagram.

### **Structure Diagram**

This menu item is only activated when you have selected a capsule on a diagram. This operation opens a structure diagram for the selected capsule.

## Open Superclass

Opens the corresponding diagram for the immediate superclass. Only applies when the current diagram is a capsule collaboration diagram or a state diagram (capsule, protocol, or data class).

## Open Subclasses

Displays a list of subclasses in a Choose Capsule Role Dialog. Only applies to capsules. Selecting a capsule from the displayed subclass list and clicking **OK** opens the corresponding diagram for the selected capsule. Only applies when the current diagram is a capsule structure diagram or a state diagram (capsule, protocol, or data class).

## Go Inside

Replaces the current diagram window contents with the corresponding diagram for the selected capsule role (for capsule structure diagrams) or substate (for capsule state diagrams). For example, selecting a substate on a capsule state diagram and choosing **Browse > Go Inside** causes the substate's state diagram to replace the current state diagram in the same window. Only applies when the current diagram is a capsule structure diagram or a state diagram (capsule, protocol, or data class).

## Go Outside

Replaces the current diagram window contents with the corresponding diagram for the selected capsule role (for capsule collaboration diagrams) or substate (for capsule state diagrams). For example, selecting a substate on a capsule state diagram and choosing **Browse > Go Outside** causes the substate's state diagram to replace the current state diagram in the same window. Only applies when the current diagram is a capsule structure diagram or a state diagram (capsule, protocol, or data class).

## Expand

Opens the subdiagram associated with an item. Packages/subsystems have a default main diagram that you can expand to if you select a package or a subsystem in another diagram.

## Parent

Selecting **Browse > Parent** or clicking the Browse Parent icon on the toolbar displays the “parent” of the selected diagram or specification. If you have a specification selected, the specification for the parent of the “named” item is displayed. For example, if you select a substate selected in a state diagram, choosing **Browse >Parent** opens a state diagram on the parent state from a substate.

## Specification...

Opens the specification dialog for selected item(s) on the current diagram.

## Top Level

Use this command to display:

- the top level main class diagram
- the top level main component diagram
- the deployment diagram for your model

### Current Diagram

### New Current Diagram

Class diagram	the main top level class diagram
State diagram	the main top level class diagram
Collaboration diagram	the main top level class diagram
Sequence diagram	the main top level class diagram
Component diagram	the main top level component diagram
Package specification	the main top level class diagram
Class specification	the main top level class diagram
Object specification	the main top level class diagram
Component package specification	the main top level component diagram
Component specification	the main top level component diagram
Process specification	the main top level deployment diagram
Device specification	the main top level deployment diagram
Connection specification	the main top level deployment diagram

## **Referenced Item**

Displays a diagram or specification referenced by the selected item. In particular, you can

- Display a diagram showing the class of which the selected object is an instance
- Display the diagram where the class, use case, or package is actually defined
- Display the operation specification for a message, provided that the message is tied to an operation; note that you must select the message label before executing this command

If the selected icon represents an object, this command finds the object's parent class and displays a diagram in which that class appears. If the class appears in multiple diagrams, the diagram in which the class was created is displayed.

If the selected icon represents a class that was created in a different logical package, this command displays a diagram from the logical package in which the class appears. If the class appears in multiple diagrams from that logical package, the diagram in which the class was created is displayed.

## **Previous Diagram**

Brings to front or opens the last diagram that was current.

## **Build Menu**

### **Build**

Opens the Build dialog from which you can choose the Build Level.

### **Quick Build**

Builds the component using the options you specified the last time you built this component.

### **Rebuild**

Forces a complete build of a component. All classes references by the component will be verified, regenerated, compiled, and linked.

### **Clean**

Removes all files from the output directory.

## **Stop Build**

Stops the build in progress.

## **Run (F5)**

Loads the component instances specified in the Build Settings Dialog. The component must be successfully built before it can run.

If the Attach Target observability flag was set on the Component Instance Specification dialog and a Target observability Port number filled in, the Target observability interface is displayed allowing you to control the execution of the model.

**Note:** The following four items only apply when a Target Observability session is running.

## **Start (F5)**

Starts the execution of the component instances. If the component instances are in the reset state, execution begins with all fixed capsules being initialized (initial transitions fired). If the component instances are in the stop state, execution resumes.

## **Stop (Shift+F5)**

Stops the execution of the component instances at the current point of execution and remembers the state of all capsules. Execution is stopped as soon as each currently running transition is finished. The stop button does not halt execution in the middle of a transition action.

## **Step (F10)**

Steps through the next deliverable message. Pressing the step button while in the stopped state causes the next message of the highest available priority to be delivered. Any associated transitions are executed. Execution stops again as soon as the last transition segment for that message has finished executing.

## **Restart (Ctrl+Shift+F5)**

Resets the component instances, resetting all fixed and destroying all dynamic capsule instances. The running component instance is terminated and a new one is run.

## **Load**

Loads the components instances specified in the Build Settings dialog. The component must be successfully built before it can run. The Load command spawns an external process in which the model executable runs. You will likely see an external command window appear.

The Attach Target observability flag must be set on the Component Instance Specification dialog, and a Target Observability Port number filled in for the model to be loaded within the tool.

The execution interface will be displayed allowing you to control the execution of the model. See Execution basics for more information on the execution tools.

## **Reload**

Kills the existing model process and runs the model again. The execution interface stays open.

## **Shutdown**

Kills the existing model process and closes the execution interface.

## **Settings...**

Displays the Build Settings Dialog. You must use this dialog to specify the active component before you can build the component.

## **Add Class Dependencies...**

Runs a script that checks for any missing dependencies between model elements and adds them. The script checks dependencies found in attributes or operations. It does not check for code-level dependencies.

## **Component Wizard...**

Activates the **Component Wizard** to help you through the steps of creating and deploying a component.

## Report Menu

Generates lists of diagrams in which the selected class is a supplier in a relationship, or in which instances of the selected class appear. These lists can be used to navigate to the diagrams they contain.

### **Show Usage...**

Obtains a list of all the locations where the selected item is used (a supplier in a relationship).

This command displays a list of diagrams in the Show Usage dialog. Double-click on a diagram from the list to display the diagram.

### **Show Access Violations...**

Obtains a list of access violations in the model. An access violation occurs when an element in one package references an element in another package that is not visible to it.

The rules for determining if an element B in package P2 is visible to an element A in package P1 are as follows:

- P1 and P2 are the same package OR
- B has its visibility set to Public AND
- there is a dependency from P1 to P2 OR
- there is a dependency from P1 to a package that contains P2 OR
- there is a dependency from a package that contains P1 to P2 OR
- there is a dependency from a package that contains P1 to a package that contains P2 OR
- P2 is marked as global (in the Detail tab of the Specification dialog)



The **Show Access Violations** menu item is available for class diagrams and component diagrams.

**To check for access violations in the Logical View:**

- 1 Open a class diagram.
- 2 With nothing selected in the diagram, choose the **Show Access Violations** menu item.
- 3 If there are any access violations, they are listed in a dialog. You can open an editor that shows the cause of a violation by selecting it in the dialog and clicking Browse (or by double-clicking on the violation). The list of violations can be sorted by clicking on either the Violator or the Supplier column headings.

To check for access violations in a specific set of classes, select those classes on the class diagram before choosing the **Show Access Violations** menu item. Selecting a package on a class diagram is equivalent to selecting each class in that package.

**To check for access violations in the Component View:**

- 1 Open a component diagram.
- 2 With nothing selected in the diagram, choose the **Show Access Violations** menu item.
- 3 If there are any access violations, they are listed in a dialog.

To check for access violations in a specific set of components, select those components on the component diagram before choosing the **Show Access Violations** menu item. Selecting a package on a component diagram is equivalent to selecting each component in that package.

The access violations calculation examines the existing class or component relationships in the model. For this reason you should ensure that the relationships are complete by building the model.

**Show Code Occurrences...**

Identifies where code is specified within the selected classifier.

**Show References...**

From a Use Case, Class or Structure diagram, will find any roles on collaboration diagrams that reference the selected classifier.

## **Documentation Report**

Generates a data dictionary from the model.

## **Show Part Of Ancestors**

This option is only available when your current diagram is a structure diagram. Opens a dialog listing all the capsules that contain this capsule as a capsule role.

## **Show Part Of Descendants**

This option is only available when your current diagram is a structure diagram. Opens a dialog listing all the contained capsule roles of this capsule.

## **Query Menu**

The **Query** menu provides commands that control which model elements appear in the current diagram. Use case diagrams, class diagrams and component diagrams support the **Query** menu functionality.

### **Add <element> Commands**

Some menu items are only available when certain diagrams are active.

Use these commands to populate the current diagram with icons representing one or more of the selected elements from the model. You can use this command to populate a new (empty diagram) or to add elements to an existing diagram. In either case, you must create or display the diagram first.

If relationships exist among the elements you are adding, or if relationships exist between added elements and any elements already appearing in the diagram, icons representing these relationships and their adornments will also appear in the diagram. Use the Filter Relationships command to directly control which kinds of relationships appear in the diagram.

### **Add Classes...**

Adds classes from the browser to the current diagram. Brings up a dialog with a list of available classes to choose from. This menu item is only visible when a Use Case or Class Diagram is open.

### **Add Capsules...**

Adds capsule classes from the browser to the current diagram. Brings up a dialog with a list of available capsule classes to choose from. This menu item is only visible when a Use Case or Class Diagram is open.

### **Add Protocols...**

Adds protocol classes from the browser to the current diagram. Brings up a dialog with a list of available protocol classes to choose from. This menu item is only visible when a Use Case or Class Diagram is open.

### **Add Use Cases...**

Adds use cases from the browser to the current diagram. Brings up a dialog with a list of available use cases to choose from. This menu item is only visible when a Use Case or Class Diagram is open.

### **Add Components...**

Adds components from the browser to the current diagram. Brings up a dialog with a list of available components to choose from. This menu item is only visible when a Component Diagram is open.

### **Add Interfaces...**

Adds classes from the browser to the current diagram. Brings up a dialog with a list of available classes to choose from. This menu item is only visible when a Component Diagram is open.

### **Expand Selected Elements...**

Allows you to specify relationship level and client/supplier criteria for choosing additional elements. All settings in the dialog are remembered from the previous use.

Use this command to show additional model elements in the current diagram. This command enables you to add icons to the current diagram elements having a specified relationship with a selected element or set of elements. For example:

- All classes that inherit from a selected class
- The classes from which a selected class inherits
- The classes that a selected class associates to
- The classes that directly use a set of selected classes
- All components that a component uses
- All packages that a component uses
- All interfaces that a component uses
- All components that a package uses
- All packages that a package uses
- All interfaces that a package uses
- All interfaces that a component realizes

From the Expand Selected Elements dialog, you can display the *Class Specification - Relations Tab* on page 344 for class diagrams or for component diagrams to specify relationship-kind and access-kind criteria for choosing additional elements.

**Note:** The level cannot be changed if you select Expand indefinitely.

### **Hide Selected Elements...**

Specifies the elements whose icons are to be removed from the current diagram.

Use this command to remove icons representing components from the current diagram. The components represented by these icons are not deleted from the model.

By default, this command removes only the components whose icons are selected. You can optionally remove icons representing components that are clients or suppliers of the selected components.

### **Filter Relationships**

Displays the Relations tab from the class and use case diagrams, and the Visibility Relations dialog from the component diagram. Both enable you to specify which kinds of relationships can appear. The filter relationship dialog remembers the last set of filter settings used. Use this command to control which kinds of relationships appear in the current diagram.

## **Tools Menu**

### **Layout**

Opens a submenu of options for rearranging the diagram:

#### **Layout Diagram**

Analyzes the location of all icons in the current diagram, determines the optimal location for the icons, and redraws the diagram.

#### **Align/Distribute...**

Opens the Align and Distribute dialog. The selected objects are arranged according to the choices made in the dialog. Alignment and distribution operations can be performed in both horizontal and vertical arrangements.

## Change View Spread

Creates space in a diagram for adding new views or creates a cleaner appearance. There are a number of different ways the views can be spread out by specifying the Spread Technique.

- **Uniform** - Indicates that the views are spread out across the diagram uniformly by the percentage. If a view is at location (100,100) and they specified a horizontal percentage of 10% and a vertical percentage of -10%, the new location of the view would be (110, 90). This affects all views in the diagram the same way.
- **Constant Radial** - Indicates that the views spread outward/inward from a central point. The preview displays a crosshair that specifies where the spread starts. It can be moved around interactively in the preview window with the mouse. The views spread out a constant distance based on the diagram size.
- **Decreasing Radial** - Is similar to Constant Radial except that the views spread progressively less far the farther away from the central point they are.
- **Increasing Radial** - Is similar to the Constant Radial except that the views spread progressively more the farther away from the central point they are.

The preview allows the user to play with the settings until the desired spread is achieved.

## Autosize All

Resizes all the objects in the diagram to fit their labels. The size of the objects increase or decrease to the minimum size required for the label to appear.

## Make Same Size

Makes two or more node views the same size in either height or width, or both. You can choose from smallest, average, or largest of all the selected views to make the new width and height. The dialog provides a preview screen.

## Size Border from View

Adjusts the black border to fit within the size of the window.

## Reposition all from Superclass

Modifies the Structure and State diagrams for a capsule by repositioning the selected item views according to their position in the superclass. Only the following items can be repositioned:

- State
- Port
- Capsule Role
- Choice Point
- Entry and Exit Point
- Port Role
- State Perimeter
- Collaboration Perimeter

If any other items are selected, they will be ignored.

**Note:** This option attempts to position (from the superclass) all item views on a diagram, including those that do not fully support this operation, such as non-linear transitions.

## Create

Opens a submenu of options for creating elements to place on the current diagram. Use the commands on the **Create** menu to place the icons in the active diagram.

When you choose an item from the **Create** menu, the corresponding diagram toolbox tool becomes active and the pointer changes to a cross (for a node) or an arrow (for a relationship). You can then use the mouse to position the pointer and place the new item.

The contents of the **Create** menu change to match the current diagram's toolbox.

## Check Model

Provides a way of re-executing the model validation that happens at open time.

Check Model is designed to be used when you are saving your model to multiple controlled units to ensure that all the units are consistent with one another. This is especially useful when parallel development is going on in multiple controlled units, since it is possible for different units to get out of sync with one another.

In a model, where one item holds a reference to another item, it is possible that a reference exists, but there isn't an item in the model of the right kind or with the right name. In that instance, the reference is unresolved.

Check Model checks the reference:

- To the supplier of any kind of relationship, uses, instantiation, metaclass, logical package import, module visibility, connection, and so forth
- From a view on a diagram to an item in the model

### **Import Code...**

Opens a file browser allowing an external code file to be selected and imported. See *Importing Rational Rose Generated Code* on page 136.

### **Model Properties**

Use the commands on the **Model Properties** submenu to display or modify the model properties associated with the model and its elements, or to display or modify model property sets.

#### **Edit**

Opens the Options Dialog with the C++ Tab if nothing is selected. This tab is used to display or modify model property values or model property sets.

#### **Replace**

Loads model property sets from the specified model property (.pty or .rtpty) file into the current model. This command deletes all model property sets in the current model, replacing them with the imported model property sets. A model component attached to a model property set that is replaced becomes attached to the replacement model property set. A model element attached to a model property set which is not replaced becomes attached to its default model property set. To make the replaced model properties a permanent part of the current model, you must save the model.

#### **Export**

Saves the current model's model property sets to a specified model property file (.pty file). When you export model properties, all of the model property sets stored with the model are written to a file that can be imported into another model.

This command displays a dialog in which you can specify the location and name of the model property file to be exported.

#### **Add**

Adds new model properties from a model property set contained in a model property file.

## Update

Modifies the existing model properties in the current model by adding and/or changing them to include the model properties in the update model property set. A model element attached to a model property set that is updated becomes attached to the updated model property set. A model element attached to a model property set that is not updated becomes attached to its default model property set.

This command opens a File Browser so you can specify the location and name of the model property file to be used to update the existing model properties.

To make the updated model properties a permanent part of the current model, you must choose the Save command from the **File** menu.

## Options...

Opens the Options Dialog, which provides control over many general model properties. (See *Toolset Options* on page 550.)

## Source Control

Opens a submenu of operations for interacting with a source control/configuration management (CM) system. For information on Source Control options, see Source Control Fundamentals in the *Guide to Team Development* for Rational Rose RealTime.

## Configure...

Opens the Model Specification dialog on the Source Control tab, which allows you to specify options relating to source control, to specify the source control system to use, and to generate unique identifiers for all elements of the model.

**Note: Generating unique identifiers affects the entire model.** Review *Unique Ids* on page 125 **before** setting this option.

## Get Entire Model

Requests a given version of all files from the CM tool, and then loads the new files.

## Synchronize Entire Model

Synchronizes the status of the model elements with the current source control tool and reloads any files that have been changed outside of the toolset.

## Refresh Status of Model

Synchronizes the status of model elements displayed in the model browser with the status as reported by the CM tool.



### **Select Checked out Units in Browser**

Selects all the units in the browser that are currently checked out.

### **Show Unit Versions**

Shows a dialog containing a list box, which displays the version of each unit.

### **Submit All Changes to Source Control**

Determines what changes have not yet been submitted to source control, then prompts you to add/checkin these changes as appropriate.

### **Synchronize Model with File System...**

Throws away any unsaved edits and reloads all files from the file system.

### **Open Script**

Opens a file browser to select a Rose REI or RRTEI script to open for editing. See The Script Editor Window in the *Rational Rose RealTime Extensibility Interface Reference*.

### **New script**

Opens the script editor to create a new Rose REI or RRTEI script. For more information, see The Script Editor Window in the *Rational Rose RealTime Extensibility Interface Reference*.

From the script editor, you can invoke several dialogs.

### **Add Watch**

Use the **Add Watch** dialog to add a variable to the Script Editor's watch variable list. For more information, see Adding Watch Variables in the *Rational Rose RealTime Extensibility Interface Reference*.

### **Modify Variable**

Use the Modify Variable dialog to change the value of a selected watch variable. For more, information, see Adding Watch Variables in the *Rational Rose RealTime Extensibility Interface Reference*.

## **Find**

Use the Find dialog to locate instances of specified text quickly anywhere within your script. For more information, see Finding Specified Text in the *Rational Rose RealTime Extensibility Interface Reference*.

## **Replace**

Use the Replace dialog to automatically replace either all instances or selected instances of specified text. For more information, see Replacing Specified Text in the *Rational Rose RealTime Extensibility Interface Reference*.

## **Calls**

Use the Calls dialog to determine the procedure calls by which you arrived at a point in your script when you are stepping through a subroutine. For more information, see Displaying the Calls dialog in the *Rational Rose RealTime Extensibility Interface Reference*.

## **Go To Line...**

Use the Go To Line dialog to jump directly to a specified line in your script. For more information, see Moving the Insertion Point to a Specified Line in Your Script in the *Rational Rose RealTime Extensibility Interface Reference*.

## **Dialog Editor**

Use the Dialog Editor to insert or edit a dialog in your script. For more information, see Working with the Dialog Editor in the *Rational Rose RealTime Extensibility Interface Reference*.

## **Add External Java**

Allows you to add external .class files and the .class files within .jar files into your existing model.

## **TargetRTS Wizard**

Simplifies the activities of building, configuring, managing and customizing the TargetRTS libraries and build environment.

## **Connexis**

Provides connectivity for Unified Modeling Language (UML) models.

### **Move Model Elements**

Provides an simple method for moving classes between packages **Logical View**.

### **Rational Quality Architect - RealTime Edition**

Activates RQA-RT to automatically verify designs against Sequence diagram specifications both analytically and during execution. Application generation and automatic testing of fully or partially complete designs, plus animated visual and symbolic debuggers, encourages early and continuous design refinement and validation.

### **Aggregation Tool ...**

Activates the **Aggregation Tool** to help you create and modify attributes. For more information, see "Aggreagation Tool" in the *Addin, Tool And Wizard Guide, Rational Rose RealTime*.

### **Model Integrator**

Opens the Model Integrator tool.

### **Web Publisher**

Opens the Web Publisher tool.

### **C++ Analyzer**

Opens the C++ Analyzer tool.

## **Add-Ins Menu**

### **Add-In Manager**

Opens the Add-In Manager dialog to activate or deactivate add-ins.

Several add-ins are shipped with the Rational Rose RealTime product, including:

- C++ language code generators
- Component Wizard
- Add Dependencies
- Generate Documentation
- C language code generators

Other add-ins will be released through Rational RoseLink partners. See the Rational Rose RealTime web site for links to RoseLink partner add-ins.

## Window Menu

The **Window** menu has commands for manipulating the windows within the Rational Rose RealTime environment, and a list of all the currently open windows. Use commands from the **Window** menu to control the automatic placement of multiple diagram and specification windows within the application window. You can also use commands on the **Window** menu to redisplay windows that have been covered by other windows or iconified.

To quickly bring a particular window to the forefront, select the name of the window from the menu.

### Cascade

Arranges all windows in an even-stepped arrangement. The windows are all sized to a standard size, and the title bar of each window is visible while the body of each window is covered by the next window in front. The most recently-viewed window is completely visible.

### Tile Horizontally

Arranges all windows horizontally within the Rational Rose RealTime window. The application window is divided into equal-size areas, with one diagram or specification window in each area. The most recently visited window is placed in the upper left corner. Less recently-visited windows are placed to the right of more recently-visited windows.

### Tile Vertically

Arranges all windows horizontally within the Rational Rose RealTime window. The application window is divided into equal-size areas, with one diagram or specification window in each area. The most recently visited window is placed in the upper left corner. Less recently-visited windows are placed below more recently-visited windows.

### Arrange Icons

Arranges collapsed windows evenly along the bottom of the Rational Rose RealTime window.

**Close**

Closes the currently active window.

**Close All**

Closes all currently open windows.

**Window Selectors**

The open windows within your application are listed on the **Window** menu with a set of numerical selectors. The windows are listed by title. Selecting one of these items from the menu opens the specified window and brings it to the forefront.

## Help Menu

**What's This?**

Opens the context-sensitive Help.

**Contents**

Opens the Help Table of Contents.

**Search...**

Opens the Help Search.

**Index...**

Opens the Help index.

**Using Help**

Opens a Help topic explaining how the Help system works.

**Tutorials**

Opens the Tutorials book from which you can choose tutorials based on your skill level and background.

**Example Models**

Opens the Example Models book.

## Keyboard Shortcuts

Opens a Help topic on keyboard shortcuts.

## Welcome to Rational Rose RealTime

Opens the Startup Screen.

## About Rational Rose RealTime

Opens the About Rational Rose RealTime dialog, which shows information on the product version, add-ins, support contacts, and so forth.

## Browsers

---

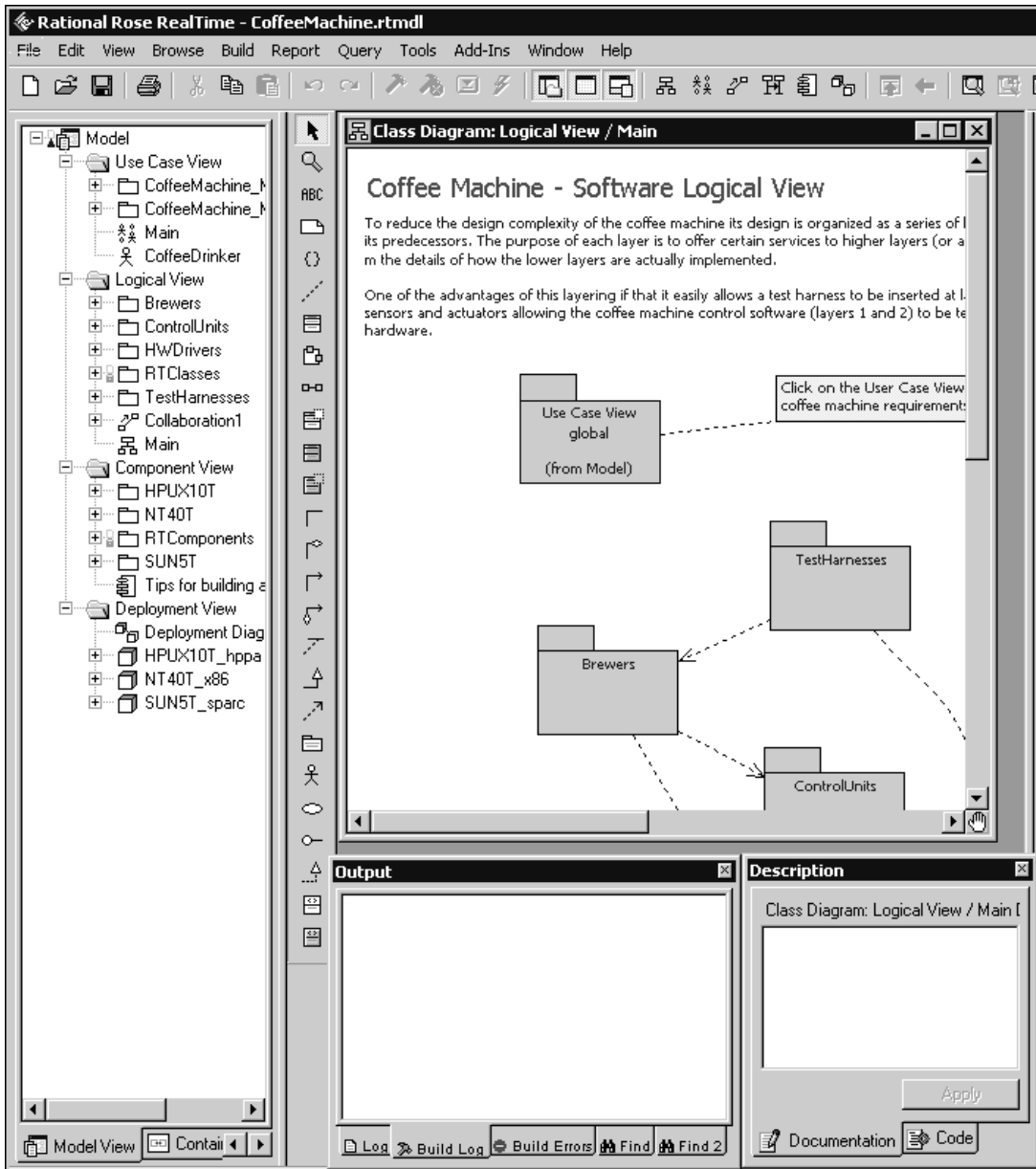
Model elements are created and viewed through Browsers. The primary tab in the browser is the **Model View** tab, which provides access to all elements of the current model. Browsers list the model elements - usually in a hierarchical way - allowing elements to be expanded to show additional information. Also, most browser lists can be filtered in various ways.

The browser is an easy-to-use alternative to menus and toolbars for visualizing, navigating, and manipulating items within your model.

The browser is a hierarchical navigational tool that lets you view the names and icons representing use case, collaboration, deployment and class diagrams, as well as model elements such as logical packages, classes, interfaces, associations and component packages associated with the model.

Figure 14 shows the main application window with the **Model View** tab in the browser.

Figure 14 Rational Rose RealTime Application Window with Browser



Each of the views within the model displays as a separate folder in the **Model View** tab in the browser. All model elements created within a view are displayed as sub-elements of that view folder.

## Tabs

The tabs in the Browser are:

- Model View Tab
- Containment View Tab
- Inheritance Tab
- RTS Tab

### Model View Tab

The **Model View** tab shows all the packages, classes and diagrams in the model. The **Model View** tab in the browser displays all of the elements of the model, organized into the four main views: **Use Case View**, **Logical View**, **Component View**, and **Deployment View**.

### Containment View Tab

The **Containment View** tab shows the containment hierarchy of the capsule classes in the model.

### Inheritance Tab

The **Inheritance View** tab shows the inheritance hierarchy of the capsule classes, data classes, and protocols in the model.

### RTS Tab

The **RTS** tab appears only when a component instance is run with Target observability enabled. This tab provides a run-time view of the model, showing the list of capsule incarnations, and providing buttons to control the model execution (see *Rational Rose RealTime Execution Interface* on page 482).

### Navigating

The plus sign (+) sign next to an icon indicates the item is collapsed, and additional information is located under the entry. Click on the + sign and the tree is expanded. Conversely, a minus (-) sign indicates the entry is fully expanded.

Double-clicking on the diagram name or icon displays the diagram. Double-clicking on any other item displays the associated specification.



## Displaying the Browser

When the Browser is first displayed, it is docked along the left edge of the frame. To move the window, click and drag on the border. The window outline indicates the window state: a thin, crisp line indicates the window is docked, while a thicker, hashmark-type border indicates it is floating.

To disable a browser, select it from **View > Browsers**. The check mark is removed along with the display of the browser.

Characteristics unique to the browser state (docked or floating) are discussed below.

### Docked:

- The window can be moved within the dockable region of the frame, but it remains positioned along the border.
- The size remains fixed. The free side is resizable.
- A ToolTip displays the icon title when partially covered by the browser border.
- The window can be docked on any border.

### Floating:

- The window can be moved to any location, and is always displayed on top of the diagram.
- Size can be changed via click and drag along the border in a vertical or horizontal direction.

## Refreshing the Browser

With the mouse positioned inside the browser, click **Refresh** from the shortcut menu.

## Multiple Browsers

You can have multiple versions of the same browser, and apply filtering that is different between them. For instance, you could open up a second browser and set its filter to show only protocols and their signals.

## Filtering

You can filter various packages, diagrams, and model elements using the **Filter** dialog.

## Diagram Editors

---

There are several different kinds of diagrams that can be created and edited through Rational Rose RealTime. Each diagram allows you to specify or document a different aspect of the model. Some diagrams are accessible in only one view, while other diagrams are found in more than one view. Each icon on a diagram represents an element in the model. Since diagrams are used to illustrate multiple views of a model, each model element can appear in none, one, or several of a model's diagrams. This means you can control which components and properties appear on each diagram.

The following is the complete list of diagrams available in Rational Rose RealTime:

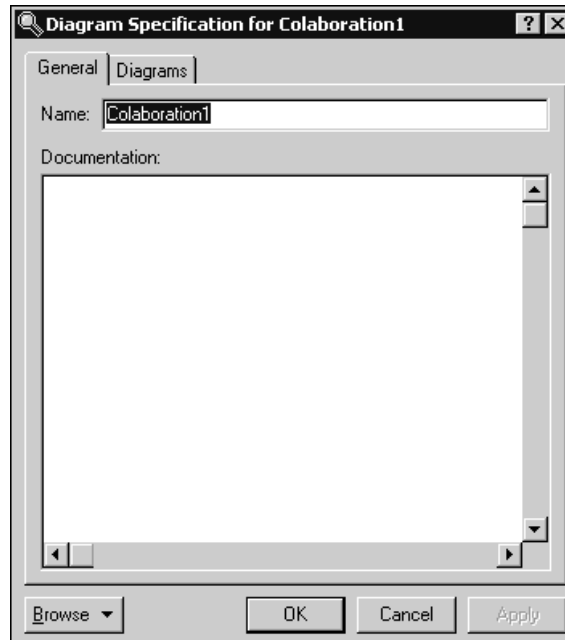
- Activity Diagrams
- Class diagram
- Use case diagram
- Collaboration diagram
- Sequence diagram
- Structure diagram
- State diagram
- Component diagram
- Deployment diagram
- Structure Monitor diagram
- State Monitor diagram

See the individual diagram topics for information on creating or modifying that particular diagram.

Each editor is displayed in a separate window. The diagram editors all have associated toolboxes.

## Diagram Specification - General Tab

Figure 15 Diagram Specification - General Tab



## Diagram Specification - Diagrams Tab

The **Diagrams** area lists the diagrams for the selected collaboration. For collaborations, you can add three types of diagrams: Collaboration, Sequence, and State diagrams.

The first column contains the icon that corresponds to the diagram's type. The **Title** column contains the title for the diagram. You can modify the name of the title.

To add a new diagram, use the shortcut menu and select the desired **Insert** option.

## Adding Icons to a Diagram

To create or add icons to a diagram, you can:

- use tools on the toolbox
- use the drag and drop capabilities of the Browsers, which you can undo and redo from the Edit Menu
- invoke commands from the Query Menu, which add icons representing specific model elements

You can cut, copy, and paste icons between different diagram windows using commands on the Edit Menu. The **Edit** menu also provides commands that enable you to select, find, and rename icons. The Browse Menu provides commands to navigate among diagrams, as well as create, rename, and delete them. Commands on the Report Menu provide additional diagram navigation capabilities. You can print diagrams using the Print command (see *Print Specifications* on page 119).

## Opening Specifications

Clicking the Specification command from the Browse Menu displays the specification for the model component represented by the selected icon.

## Shortcut Menu

Clicking the right mouse button on an icon activates the popup menu, which enables you to modify properties (for icons that represent relationships) or select properties to be displayed within the icon. Select the Open Specification command from the context menu to open the specification dialog.

## Background Shortcut Menu

The Background shortcut menu changes according to the diagram in which you click. Following are some basic menu items:

### Zoom

Zooms in on the current diagram. Select the zoom level.

### Scale to Window

Enables the automatic resizing of icons to accommodate text.

### Layout

Opens a submenu of options for rearranging the diagram:

#### Layout Diagram

Analyzes the location of all icons in the current diagram, determines the optimal location for the icons, and redraws the diagram.

#### Align/Distribute

The selected objects are arranged according to the choices made in the dialog. Alignment and distribution operations can be performed in both horizontal and vertical arrangements.

## Change View Spread

Creates space in a diagram for adding new views or creates a cleaner appearance. There are a number of different ways the views can be spread out by specifying the Spread Technique.

- **Uniform** - The views are spread out across the diagram uniformly by the percentage. If a view is at location (100,100) and they specified a horizontal percentage of 10% and a vertical percentage of -10%, the new location of the view would be (110, 90). This affects all views in the diagram the same way.
- **Constant Radial** - The views spread outward/inward from a central point. The preview displays a crosshair that specifies where the spread starts. It can be moved around interactively in the preview window with the mouse. The views spread out a constant distance based on the diagram size.
- **Decreasing Radial** - Similar to Constant Radial except that the views spread progressively less far the farther away from the central point they are.
- **Increasing Radial** - Similar to the Constant Radial except that the views spread progressively more the farther away from the central point they are.

The preview allows the user to play with the settings until the desired spread is achieved.

## Autosize All

Resizes all node views in the diagram to fit their labels.

## Make Same Size

Makes two or more node views the same size in either height or width, or both. You can choose from smallest, average, or largest of all the selected views to make the new width and height. The dialog provides a preview screen.

## Select in Browser

Selects an element from the diagram in the browser.

## Filter

Use these options to filter information on diagrams.

## Scroll Bars

Diagram windows provide vertical and horizontal scroll bars to pan across diagrams larger than the window.

# Overview Navigator and Toolset Buttons, and Class, Capsule, and Protocol Specification Context Menus

## Overview Navigator Button

Use the **Overview Navigator** (the hand symbol located in the bottom right-hand corner of the diagram dialog) to navigate around a diagram.

## Toolset Buttons

For descriptions of the buttons in the Toolbar, see *The Toolbar* on page 39.

## Context Options for Specification Dialogs

**1 Language Details** - Shows a submenu for the Attribute, Operation, and Aggregation tools.

**Browse Code** - Lets you view the code associated with the selected object.

**Build** - Lets you specify a build type:

**Build** - Opens the Build dialog from which you can choose the Build Level.

**Quick Build** - Builds the component using the options you specified the last time you built this component.

**Rebuild** - Forces a complete build of a component. All classes references by the component will be verified, regenerated, compiled, and linked.

**Clean** - Removes all files from the output directory.

**Code Sync** - Captures user changes, made to generated code, back into the model.

**Configure Capsule for Connexis** - Shows the Configure Connexis Capsule dialog where you can select options specific for Connexis.

**Connexis** - Provides connectivity for Unified Modeling Language (UML) models.

**Duplicate** - Creates a new object with a new name that is identical to the currently selected object.

**New Attribute** - Creates a new attribute for the selected object.

**New** - Creates a new object for the selected control.

**New In Signal** - Creates a new in-signal for the selected object.

**New Operation** - Creates a new operation for the selected object.

**New Out Signal** - Creates a new out-signal for the selected object.

**New Port** - Creates a new port for the selected object.

**Open Specification** - Opens the **Specification** dialog for the selected object.

**Open State Diagram** - Opens the **State** diagram for the selected object.

**Open Structure Diagram** - Open the **Structure** diagram for the selected object.

**Open Subclass** - Opens the **Specification** dialog for the subclass of the selected object.

**Open Superclass** - Opens the **Specification** dialog for the superclass of the selected object.

**Options** - Shows a submenu of options for the selected object.

**Rational RequisitePro Trace Tool** - Lets you maintain and establish traceability between your design requirements and your Rational Rose RealTime elements.

**Relocate** - Lets you relocate a model element to a new package.

**Select in Browser** - Makes current the active object on the Model View tab in the browser.

**Set As Active** - Sets the current component to be the active component. Select this option if you build and run the same component and component instances often.

**Source Control** - Opens a submenu of operations for interacting with a source control or configuration management (CM) system.

## **Context Options for Other Controls**

**Allow Docking** - Toggles whether to allow docking of the current window.

**Attach Console** - Attaches a console window to the executing target model to interact with the command line model debugger, and so forth.

**Attach Target** - Enabled only if a component instance has been run without observability at startup. You can attach observability to a running process at any time, if that the process was started with observability enabled. This menu item can only be used with **Detach Target**.

**AutoSave** - Automatically captures all of the build output to a log file so that you can process the output later.

**Clear** - Deletes all information on the selected tab or window.

**Copy** - Copies the currently selected item or items to the clipboard. From the clipboard, you can paste items into other diagrams, or paste items into documents you create with any standard word-processing software.

**Cut** - Removes the selected item and places it in the buffer.

**Delete** - Removes the selected element from the diagram.

**Detach Target** - Detaches observability, meaning that the toolset no longer communicates with the running component instance. This menu item can only be used with Attach Target.

**Hide** - Toggles the display of the **Output** window.

**Load** - Loads or downloads a component instance to a target platform. The load does not start the execution of the loaded component instance. Use Run once it is loaded. This is only used with target platforms that require loading of modules before they are run. For platforms that do not require loading of modules, this menu item is disabled.

**Open Breakpoint Diagram** - Displays the **Breakpoint Diagram** dialog from which you can set breakpoints on a state machine.

**Reload** - Reloads a components instance. Used only with target platforms that require loading of modules before they are run. For platforms that do not require loading of modules, this menu item is disabled. This option unloads, then loads the component without resetting the target board.

**Rename** - Changes the name of the currently selected element.

**Restart** - Kills the running component instance and runs another instance. If the instance is running on an target board, the component is reloaded before a new instance is run.

**Run** - Starts the execution of the component instance. If observability is configured to attach at start-up the RTS browser appears. When observability is attached at start-up the component instance is paused, or does not start processing messages, until you click the **Start** button on the **RTS Browser**.

**Save As** - Saves the current output to a log file so that you can process the information later.

**Select All** - Selects all the data in the current tab or window.

**Shutdown** - Kills the running component instance, closing the RTS Browser if necessary.

**Time Stamp** - Prefixes messages posted to the log with a time stamp.



**Unload** - Unloads a components. Use only with target platforms that require loading of modules before they are run. For platforms that do not require loading of modules, this menu item is disabled.

**View Breakpoints** - Shows a list of all of the current breakpoints you specified.

## Sequence Diagram Context Menu

### Scale to Window

Scales the current diagram down to fit entirely within the current diagram window border. Scales according to the outer boundaries of the diagram - for example, the outer state border of the state diagram - and not simply the area around visible diagram elements.

### Zoom

Zooms in on the current diagram.

### Layout

Opens a submenu of options for rearranging the diagram:

- **Layout Diagram**

Analyzes the location of all icons in the current diagram, determines the optimal location for the icons, and redraws the diagram.

- **Align/Distribute**

Opens the Align and Distribute dialog. The selected objects are arranged according to the choices made in the dialog. Alignment and distribution operations can be performed in both horizontal and vertical arrangements.

- **Change View Spread**

Creates space in a diagram for adding new views or creates a cleaner appearance. There are a number of different ways the views can be spread out by specifying the Spread Technique.

**Uniform** - Indicates that the views are spread out across the diagram uniformly by the percentage. If a view is at location (100,100) and they specified a horizontal percentage of 10% and a vertical percentage of -10%, the new location of the view would be (110, 90). This affects all views in the diagram the same way.

**Constant Radial** - Indicates that the views spread outward/inward from a central point. The preview displays a crosshair that specifies where the spread starts. It can be moved around interactively in the preview window with the mouse. The views spread out a constant distance based on the diagram size.

**Decreasing Radial** - Is similar to Constant Radial except that the views spread progressively less far the farther away from the central point they are.

**Increasing Radial** - Is similar to the Constant Radial except that the views spread progressively more the farther away from the central point they are.

The preview allows the user to play with the settings until the desired spread is achieved.

- **Autosize All**

Resizes all the objects in the diagram to fit their labels. The size of the objects increase or decrease to the minimum size required for the label to appear.

- **Make Same Size**

Makes two or more node views the same size in either height or width, or both. You can choose from smallest, average, or largest of all the selected views to make the new width and height. The dialog provides a preview screen.

- **Size Border from View**

Adjusts the black border to fit within the size of the window.

### **Open Interaction Specification**

Opens the **Interaction Specification** dialog for the selected interaction.

### **Open Collaboration Diagram**

Opens the **Structure** diagram for the collaboration.

### **Validate**

Allows you to check the Sequence diagram specification for missing elements. It provides control over what aspects of the Sequence diagram should be checked for completeness.

### **Auto-Create FOC's**

Determines whether any new send or call messages automatically get an FOC (Focus of Control) - and a return message, if appropriate - when they are created.

### **Select In Browser**

Identifies the location of the selected item on the **Model View** tab in the browser.

Compares previous runs and actual production traces with your Specification diagrams. You can specify more precise filtering for the differencing to obtain a more accurate representation of the differences.

### **Select For Difference**

Sets the current Sequence Diagram for differencing (for RQA-RT). You can select only one diagram, either from the **Model View** tab in the browser or the diagram itself.

### **Difference**

Initiates the differencing process for one or two selected sequence diagrams.

### **Select Race Conditions**

Displays a list of all of the pairs of messages in the selected Sequence Diagram which are in a race condition. A race condition occurs between pairs of events; when events appear in one order in the sequence diagram, but occur in either the same or an opposite order when the system runs.

## **Toolboxes**

Every diagram has an associated toolbox, which contains icons of tools that can be applied to that diagram. If the current diagram is write-protected, the diagram toolbox is not displayed. The diagram toolbox is also only available when a diagram is displayed.

There are several tools that are common to every toolbox:

### **Selector Tool**

Selects objects for moving, resizing, and so forth.

### **Zoom Tool**

Zooms in on a portion of the diagram. Click on the tool and then click on the part of the diagram you want to zoom in on.

### **Text Tool**

Adds text anywhere in the structure diagram.

### **Note Tool**

Annotates the diagram with textual notes. This is useful for marking up the diagram, for example, with explanations and review comments. You can also drag and drop a diagram or external document from the browser onto a note. When the name of the diagram or external document is underlined, the name is a hyperlink to a diagram or URL. If you double-click on the note, the diagram or external document opens.

### **Constraint Tool**

Adds UML constraints to any diagram. A constraint can be anchored to a view element by using the anchor tool. Currently, constraints do not have any semantic meaning to the tool. There are RRTEI APIs to add or remove, and enumerate constraints in a diagram.

### **Note anchor Tool**

Anchors a note to a particular element on the diagram. Allows notes to be moved with the element they are anchored to.

### **Lock Selection Tool**

Use to make the tool selections stay locked. That is, if the lock is on, then the next tool you select will stay selected after you've completed the operation. This allows you to perform a number of operations with a particular tool without having to reselect the tool after each operation.

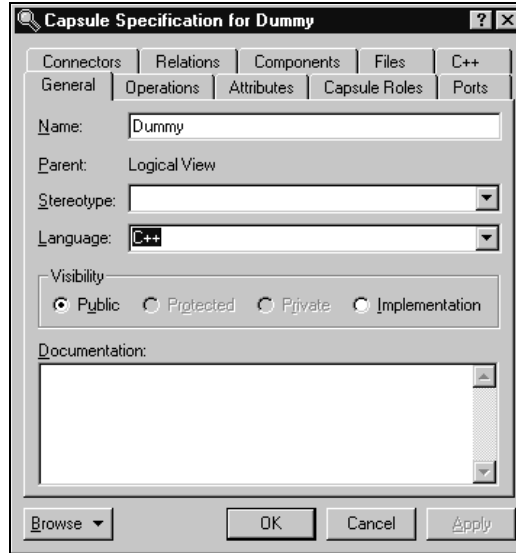
You may also hold down the shift key to keep a tool selected. The last selected tool stays active until you release the shift key.

## **Specification Dialogs**

---

Specification dialogs are used to edit the properties of any element in the model. All specification dialogs contain at least a Name and Documentation field. Some specification dialogs contain many fields split across different tabs. The example below shows a specification dialog with multiple tabs. All properties of a modeling element are accessible through the specification dialog for that element. Many of the properties on the specification dialog are also visible/editable through one or more diagram editors.

**Figure 16 Sample Specification Dialog for a Capsule**



Specification dialogs are resizable. The tab you are in is remembered so that the next time you open the specification dialog you go to the same tab. The position and size of specification dialogs are saved with the workspace.

## **Spreadsheet-type Functionality for List Controls within a Specification Dialog**

When the list control has focus, the following applies:

- F2 or ENTER key puts the field in inline edit or drop down combination mode. Press ENTER again to accept the data and move to the next row in the column.
- The TAB key also accepts data (when editing a cell) and goes to the next column of the same row. If you are in the last column, it moves to the first column of the next row. If you are in the last column of the last row, it inserts a new row and begins inline editing.
- When not editing a cell, the SHIFT + TAB combination works as the reverse of TAB; that is, it moves to the previous column in the same row, or if you are in the first column it moves to the last column of the previous row. If you are in the top-leftmost cell, it does nothing.

## **Browse**

Clicking **Browse** displays the following context menu options:

- **Select in Browser** - Highlights the selected element in the browser.
- **Open Diagram** - Opens the diagram associated with the object.
- **Browse Parent** - Opens the **Specification** dialog for the parent of the selected element.
- **Browse Selection** - Opens the **Specification** dialog for the currently selected element.
- **Browse Superclass** - Opens the **Specification** dialog for the superclass of the selected object.
- **Browse Subclasses** - Opens the **Specification** dialog for the subclasses of the selected object.
- **Show Usage** - Displays a list of all diagrams in which the currently selected element is the supplier, or in the case of a collaboration diagram, a list which shows the usage of a message.
- **Find References** - Finds a specified item in the model by searching all fields - excluding Documentation - and searching all objects - excluding Diagrams, Component Instances, devices, Instances, Interactions, Messages, Packages, Probes, and Processors. This command displays the **Find In Model Dialog** so that you can type the search string.

## **OK**

Accepts changes and closes the dialog.

## **Cancel**

Ignores changes that were made and closes the dialog.

## **Apply**

Commits any changes that were made.

## **Help (?)**

Opens the Online Help for the current **Specification** dialog.

## **Exit (X)**

Closes the specification dialog.

## Tabs

Many dialogs include a number of tabs across the top for grouping different specification information. The Files Tab, Relations tab, Components tab, Attributes tab, Operations tab, and Unit Information Tab may be displayed for many different model elements. **Unit Information** tab only appears if the model is being controlled as units.

## Actions Tab

### Type

Displays the action specified in the **Action Specification** dialog:

- **Action** - A simple action may be the invocation of a method, or the starting or stopping of an activity.
- **Send Event** - Send events are actions that trigger another event.

The type of action determines what options are available in the dialog box.

Double-click on an action to open the **Action Specification** dialog for the selected action. If there are no actions listed, right-click on the **Actions** tab and click **Insert**.

### Action Expression

Displays the timing option that specifies when to carry out an action and the types of actions that are carried out. Actions on activities can occur:

- **on entry** - The task is performed when the object enters the activity
- **on exit** - The task is performed when the object exits the activity
- **do** - The task is performed while in the activity and must continue until exiting the activity
- **on event** - The task triggers an event only when a specific event is received

You can modify the action settings through the **Detail** tab of the **Action Specification** dialog.

## Attributes Tab

The UML asserts that attributes are data values (string or integer) held by objects in a class. Thus, the Attributes tab lists attributes defined for the class. The attribute definition can be modified through the Attribute Specification dialog.

**Note:** Attributes and relationships created using this technique are added to the model, but do not automatically appear in any diagrams. That is, adding an attribute affects the code generation for the class and a compilation dependency between the class of the container and the class of the attribute, but these relationships are not graphically visible in the model.

The descriptions for each field follow:

- **Visibility Adornment** (Unlabeled):
  - **Public** - The attribute is publicly visible, and is accessible to all clients.
  - **Protected** - The attribute may be accessed only by subclasses, friends, or by operations of this class.
  - **Private** - The attribute is accessible only by the class itself or by its friends.
  - **Implementation** - The attribute is accessible only by other operations in this class.
- **Stereotype** - Displays the name of the stereotype.
- **Name** - Displays the name of the attribute.
- **Class** - Identifies where the attribute is defined.
- **Type** - This can be a class or a traditional type, such as **int**.
- **Initial** - Displays the initial value of an object.

The Attribute tab is active for all class types.

### Show Inherited

Click this option to see attributes inherited from other classes. If there is no check mark in this field, you can view only attributes associated with the selected class.

**Note:** Rational Rose RealTime allows you to directly modify any attribute shown in the attributes list by displaying the attribute specification dialog. You should be careful when modifying base class attributes for it may have implications on other elements in your model which reference or are subclassed from the base class.



## Creating New attributes

You can add an attribute relationship by selecting **Insert** on the popup menu or by pressing the insert key. A new attribute with a default name is added.

## Moving and copying attributes

To move an attribute from one Specification sheet to another, drag and drop it. From the **Edit** menu of the main window, you can select **Undo** and **Redo**.

To copy an attribute from one Specification sheet to another, drag and drop it while holding down the CTRL key. From the **Edit** menu of the main window, you can select **Undo** and **Redo**.

## Components Tab

The components list displays a list of components to which this class has been assigned. Components can be inserted, deleted, and moved up and down in the list. Each component has a corresponding Component Specification for editing the component attributes.

A check-box provides filtering control over which components are displayed:

**Show all components** displays the list of all components in the model.

Right-clicking on a component brings up the Components popup menu.

## Detail Tab

### When

Specifies a timing option to carry out for the selected action.

### On Event

The **On Event** parameters are only enabled when you set the **On Event** timing parameter in the **When** box.

- **Event** - In an **Activity Diagram**, an event is an occurrence that can trigger a state transition. Type the name of the event that will trigger the action.
- **Arguments** - Specifies any optional arguments associated with the event.
- **Condition** - Specifies a conditional **Boolean** expression.

You can use an **On Event** action rather than a self-transition because self-transitions trigger all the actions associated with a state, whereas state and activity actions handle internal state and activity transitions. This means that you can process an internal event without triggering the **entry** and **exit** actions.

## Type

Specifies the type for the action.

- **Action** - A simple action may be the invocation of a method, or the starting or stopping of an activity.
- **Send Event** - Send events are actions that trigger another event.

The type of action determines what options are available in the dialog box.

## Name

Specifies a name of the **Action** or **Send Event**. This name appears on the state or activity on the Activity Diagram.

## Send arguments

Specifies any arguments for a send event. One or more arguments can accompany a send event.

## Send target

Specifies any targets for the send event. A target is any object that will receive the transition event.

## Files Tab

A list of referenced files is provided here. The files list popup menu allows you to insert and delete references to files or URLs.

You can link external files to model elements for documentation purposes.

## General Tab

### Name

Specifies the name for the currently selected state.

### Stereotype

Specifies a keyword that further defines the classification of the model element. A stereotype represents the subclassification of a model element. Some stereotypes are already predefined, but you can also define your own to specify new modeling types.

To view stereotypes on the Activity Diagrams, click **Tools > Options**, select the **Diagram** tab, and click **Label**, **Decoration and label**, **Decoration only**, or **Icon** in the **Stereotype** box. **Label** displays the stereotype name in angle brackets (for example, <<stereotype>>). **Decoration** displays a graphic marker such as highlighting an icon or tool. **Icon** displays the graphical representation, if any.

### **Owner**

Specifies the model elements that own the selected state.

### **Context**

Specifies a view for a related set of modeling types.

### **Documentation**

Describes model elements or relationships. The description can include such information as the constraints, purpose, and essential behavior of the element. The information you type in this field is not displayed in the Activity Diagram.

### **State/activity history**

Specifies whether to return the most recently visited state or activity when transitioning directly to a state or activity with sub-states or sub-activities. Set this option to apply history at the state or activity level.

History provides a mechanism to return to the most recently visited state when transitioning directly to a state with sub-states. History applies to the level in which it appears. It may also be applied to the lowest depth of nested states.

### **Sub state/activity history**

Specifies history for all depths for nested states or activities within the state or activity level. Set this option to apply history to all the depths of nested states or activities within the state or activity level.

History provides a mechanism to return to the most recently visited state when transitioning directly to a state with sub-states. History applies to the level in which it appears. It may also be applied to the lowest depth of nested states.

## **Operations Tab**

Operations denote services provided by the class. Operations are methods for accessing and modifying Class fields or methods that implement characteristic behaviors of a class.

The **Operations** tab lists the operations that are members of this class. The actual definition of the operation is accessible from the **Operation Specification**.

The operations are listed with the following fields:

- **Visibility Adornment** (Unlabeled) - The visibility of the operation is indicated with an icon. Following are the visibility options:
  - **Public** - The operation is accessible to all clients.
  - **Protected** - The operation is accessible only to subclasses, friends, or to the class itself.
  - **Private** - The operation is accessible only to the class itself or to its friends.
  - **Implementation** - The operation is accessible only by operations of this class.
- **Stereotype** - Displays the name of the stereotype.
- **Signature** - Displays the name of the operation.
- **Class** - Identifies which class defines the operation.
- **Return Type** - Identifies the type of value returned from the operation.

The **Operation** tab is active for all class types. In the class diagram, you can display operation names in the class compartment.

### **Show Inherited**

Click this option to see operations inherited from other classes. If there is no check mark in this field, you can view only operations associated with the selected class.

Note: Rational Rose RealTime allows you to directly modify any operation shown in the operations list by displaying the operations specification dialog. You should be careful when modifying base class operations for it may have implications on other elements in your model which reference or are subclassed from the base class.

### **Creating New Operations**

To enter an operation in the **Class Specification** dialog, select **Insert** from the popup menu. A new operation with a default name is added to the operations list.

### **Moving and Copying Operations**

To move an operation from one Specification dialog to another, drag and drop it. From the **Edit** menu of the main window, you can select **Undo** and **Redo**.

To copy an operation from one Specification dialog to another, drag and drop it while holding down the CTRL key. From the **Edit** menu of the main window, you can select **Undo** and **Redo**.

## Relations Tab

The relations list displays relations between classes as specified in diagrams. Relations can be inserted, deleted, and moved up and down in the list. Each relation has a corresponding Association specification for editing the relation attributes.

A check-box provides filtering control over which relations are displayed:

**Show Inherited** shows any relations inherited from a superclass.

## Swimlanes Tab

### Name

Specifies the swimlane name where the enclosed state or activity resides.

## Transitions Tab

### Event

Specifies the names of all the events for transitions associated with the selected element.

An event causes a state transition. You do not have to uniquely label events because one event can cause a transition to many different states or activities.

### End

Specifies the target state or activity for transitions.

## Unit Information Tab

The **Specification** dialog for a controlled element includes a **Unit Information** tab.

### Descriptions

#### Owned by model

Indicates whether the unit is owned by this model or whether it is owned by another model and shared into this model. This setting is not directly editable.

### **Under source control**

Indicates whether this element has been added to source control. This setting is not directly editable.

### **Control new child units**

Controls whether newly created controllable elements in this package will be individually controlled by default. This check box is only displayed in the Unit Information tab for a package.

### **Disallow model-relative pathnames**

Informs Rational Rose RealTime to not use the implicit \$@ virtual pathmap symbol when saving units located anywhere within this package. This check box is only displayed in the Unit Information tab for a package.

### **Scratchpad**

Indicates that the package is a scratch pad. This check box is only enabled in the **Unit Information** tab for a package that is not under source control.

### **Filename**

Displays the name of the file that is used to save this controllable unit. This field is not directly editable.

### **Save As**

For controlled units, this option saves the selected unit, and if specified, its child units (if any exist), to a new location.

If the model is under source control, Rational Rose RealTime prompts you to check out the parent unit before proceeding with the **Save As** operation.

If you cancel the **Save As** operation, the unit file names change back to the names previous to clicking the **Save As** option; however, any files already saved are not deleted from your hard drive.

You cannot undo the **Save As** operation.

**Note:** The containing unit of the unit you click **Saved As** for is identified as being modified. It is essential to save the model to update the containing unit. If the parent unit is not saved, it will point to the original file location.

## Version

Displays the version identifier for this controlled unit. If this information is not known, then '<unknown>' is displayed. The ability to extract this version information depends on the source control tool being used. If a unit is not under source control, then this field is not displayed.

## Scratch Pad Packages

When working on a model in a team environment, it is common for a developer to create temporary model elements that are not intended to be shared with the rest of the team. For example, a developer may create a temporary component when unit testing a change to a capsule class. If the model is under source control, then the developer will also not want these temporary elements to be checked in with the other changes they are making.

In order to support temporary work within a controlled model, Rational Rose RealTime supports scratch pad packages. A scratch pad package is a package that will never be added to source control. Also, in a model that is under source control, changes can be made to a scratch pad package without the toolset requiring that package be checked out. This allows multiple team members to make temporary changes within the scratch pad without encountering any contention issues.

Elements can be moved into or out of a scratch pad package by dragging them to another package in the browser. Elements can also be copied into (or out of) a scratch pad package using control-drag.

The controllable elements within a scratch pad package cannot be individually controlled. If a controlled unit is moved into a scratch pad package, then it will no longer be controlled.

### To create a scratch pad package:

- 1 Create a package and give it a descriptive name, for example, ScratchPad.
- 2 Select the package in the browser and choose **File > Control Unit**.
- 3 Open the Specification dialog for this package and change to the Unit Information tab.
- 4 Select the Scratchpad and click **OK**.
- 5 Save the package containing the scratch pad. Optionally you can also save the scratch pad. If the containing package is under source control, it should be checked out and checked in.

# Searching and Sorting

---

This topic is organized as follows:

- *Using Sort* on page 104
- *Find In Model Dialog* on page 105
- *Replace Dialog* on page 106

## Using Sort

### Sorting in the Browser

Sorting in the browser enables you to arrange classes, attributes, operations, and packages in alphabetical order.

Follow these steps to place items alphabetically in the browser:

- 1 Highlight a class or package icon.
- 2 Right-click on the icon.
- 3 Click Sort in the shortcut menu.

**Note:** The arranged items in the browser are not saved when you close the application.

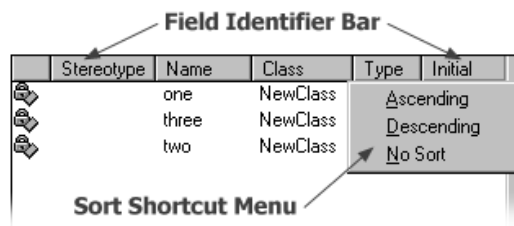
Right-clicking in an application browser displays the popup menu, in which Sort is an item. Choose between Alphabetical Order and Internal Order. Sort settings are saved for each browser in your workspace.

### Sorting in the Class Specification

Sorting in class specification enables you to arrange attributes and operations three ways.

**To arrange items in class specification:**

- 1 Highlight an attribute or operation.
- 2 Right-click on any Field Identifier bar located in class specification.





- 3 Click either **Ascending**, **Descending**, or **No Sort** from the shortcut menu options.
  - **Ascending** - lists attributes and operations in case-sensitive alphabetical order.
  - **Descending** - lists attributes and operations in reverse, case-sensitive alphabetical order.
  - **No Sort** - lists attributes and operations in the order they are specified in the model.

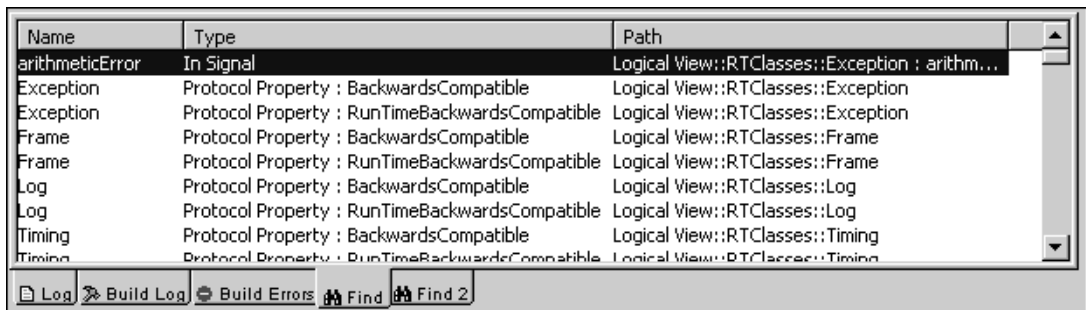
## Find In Model Dialog

Use the **Find** and **Find In Model** dialogs to search for model elements by name, or to search detail code segments.

You can narrow your search by selecting specific fields and items types from the list. This enables you to specify which objects are included in the search.

By default, the results of a search display in the **Find** tab of the **Output** window. To view the **Output** window, click **View > Output**.

**Figure 17 Find Tab in the Output Window**



If you select **Output to Find 2 pane** in the **Find In Model** dialog, the results display in the the **Find 2** tab of the **Output** window.

The results of searching displays a list of diagrams in which that item appears. You can double-click on an entry in the **Find** tab or click **Browse** to go directly to the **Specification** dialog or diagram.

## Using a Wildcard

To search for groups of items or to search code, you can use the wildcard character (\*):

- **A\*** matches any name beginning with the letter A
- **\*A** matches any name ending with the letter A
- **\*A\*** matches any name containing the letter A

**Note:** To search for the wildcard character "\*", use "\\*".

The wildcard is particularly useful for finding any classes or packages that were automatically renamed in a model that was upgraded from a previous release. For example, you can search for every model item that was renamed by typing the following:

```
*##*
```

Every model item that has a pound symbol (#) in its name will be found.

## Searching Code

The **Find in Model** dialog searches for specified text in all detail level code in a model: transitions, entry and exit code, choice points, guards, and operations. It also searches the dimension and type property of all attributes, the data type of signals, documentation, and language tab properties.

## Searching for Model Elements by Name

You can search the model for elements whose name matches the string specified in the **Find What** box. Selective searching based on the type of element is also supported, for example, search only capsules or state diagrams.

## Replace Dialog

You can use the Replace dialog to search and replace detail code segments or to search and replace for model elements by name. Results are displayed in the Find 2 tab of the Output window (**View > Output**).

## Searching Code

Use the Replace dialog to find and replace all uses of a class in detail code or which is a property of another class. For example, you could find and replace attributes of a specific type or signal data types of a specific name.

The Replace dialog searches and replaces for the specified text in all detail level code in a model: transitions, entry/exit code, choice points, guards, and operations. It also searches the dimension and type property of all attributes, the data type of signals, documentation, and language tab properties.

### Searching for Model Elements by Name

You can also search and replace the model for elements whose name matches the string specified as the Item to find. Selective searching and replacing based on the type of element is also supported, for example, search and replace only capsules or state diagrams.

When you click **Replace**, a secondary dialog (Figure 18) appears with options to *Find Next*, **Replace**, **Replace All**, and **Cancel**. After the replace operation has taken place, you can query the **Find** tab in the **Output** window to view the results of the search and replace.

**Figure 18** Replace Fields dialog



### Selective searching

You can toggle several items at once by pressing the SHIFT key, then selecting a set of elements types from the list, then using the SPACEBAR to change the selection of the options.



## Contents

This **chapter** is organized as follows:

- *Description Window* on page 109
- *Adding Documentation to Model Elements* on page 111
- *Adding Code to Model Elements* on page 112
- *Output Window* on page 112
- *Specification History Window* on page 115

This chapter describes other application windows, including the **Description Window**, which contains the following:

- **Documentation Tab**
- **Code Tab**
- **Output Window**

## Description Window

---

The Description window contains the **Documentation Tab** and the **Code Tab**. You can toggle between the two by clicking their tabs.

### Displaying the Description Window

By default, the **Description** window is closed. To view the window, select **View > Description**.

Only one **Description** window can be open at a time, but as you select different items, the window updates accordingly. If you select an item that has no documentation or code associated with it, you select multiple items, or you do not have an item selected, you are notified.

When the window is first displayed, it is docked to the bottom left corner. To move the window, click and drag on the border. The window outline indicates the window state: a thin, crisp line indicates the window is docked, while a thicker, hashmark-type border indicates that it is floating.

Characteristics unique to the window state (docked or floating) are:

- *Docked* on page 110
- *Floating* on page 110

### **Docked**

- The window can be moved within the dockable region of the application.
- The size can be changed using the splitter bars.
- The title can be displayed through a tool tip (simply place your cursor anywhere in the window). There is no title available when the window is docked.
- The window can be docked at any time.

### **Floating**

- The window can be moved to any location, and is always displayed on top of the diagram.
- Size can be changed using click and drag along the border in a vertical or horizontal direction.

The window title displays the description. The static text displays the name of the element who's code or documentation you are viewing.

## **Documentation Tab**

You can use the **Documentation** tab to edit or view the documentation associated with the currently selected model element. Scroll bars are added when necessary and word wrap is employed.

## **Code Tab**

You can use the **Code** tab to edit or view the code associated with the currently selected model element. Scroll bars appear when necessary and word wrap is employed.

## **Word Wrap**

To enable or disable word wrap in a text box, such as a **Documentation** box or **Code** box, right-click in **Code** box and select **Word Wrap**. The keyboard shortcut to toggle **Word Wrap** is CTRL + O.

**Note:** Word wrapping only affects the display of text. When saving or exporting, Word Wrap is set, it does not affect the output format for the text. The **Word Wrap** menu option is only enabled when the edit window contains text.

## Pull-down Menu

Using the pull-down menu, you can move between different sections of code, for example, between entry and exit code for a state. You can also add a trigger to a transition.

## Popup Menu

Using the popup menu, you can

- import and export files containing code
- print the code
- select individual words, lines, or all of the code you are viewing
- specify the font
- use the Search feature to Find and Replace
- launch an external editor

**Note:** If you use an external editor that requires a console terminal, you must specify an application, such as **xterm**, that provides the terminal, followed by the editor command itself.

**Note:** Example on Solaris: `/usr/openwin/bin/xterm -e /bin/vi`

## Adding Documentation to Model Elements

---

All model elements can have documentation associated with them.

**To add documentation to a model element, you can use the Documentation window or follow these steps:**

- 1 Right-click on the model element in the model browser or in a diagram.
- 2 Select **Open Specification** from the selected item's menu.
- 3 Click the **General** tab if it is currently displayed.
- 4 Enter the documentation for the element in the documentation area.
- 5 Close the **Specification** dialog by clicking **OK**.

For long, complex, or formatted documentation, you may want to link an external file (such as an Microsoft Word document) to a model element. See *Inserting a Diagram into an MS Word Document* on page 541.

**Note:** If you add documentation from the **Documentation** tab, you must click **Apply** for to save the information.

## Adding Code to Model Elements

---

All model elements can have code associated with them.

**To add code to a model element, you can use the code window or**

- 1 Right-click on the model element in the model browser or in a diagram.
- 2 Select **Open Specification** from the selected item's menu.
- 3 Click on the language-specific tab if it is not the tab currently displayed.
- 4 Enter the code.
- 5 Close the Specification dialog by clicking **OK**.

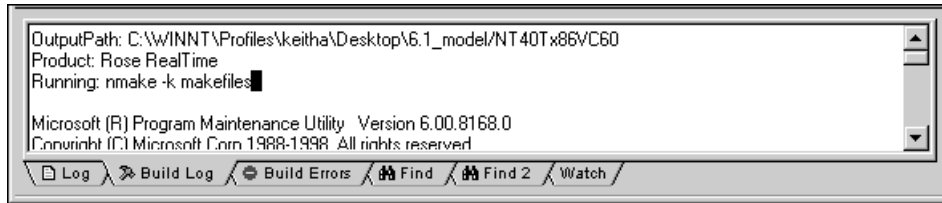
## Output Window

---

The **Output** window (see Figure 19) is a dockable window that contains the following tabs:

- **Log Tab**
- **Build Log Tab**
- **Build Errors Tab**
- **Find Tab**
- **Watch Tab** (RTS only)

**Figure 19 Output window**



### Log Tab

The **Log** tab is used by several commands to report progress, results, and errors. Messages posted to the log are usually prefixed with a time stamp.

To display the **Log** tab, select **View > Output**. The application posts the messages to the log window regardless of whether it is displayed.



You can save the contents of the log window to a file or you can choose to automatically save messages to a file as they are posted. Both options are available from the popup menu.

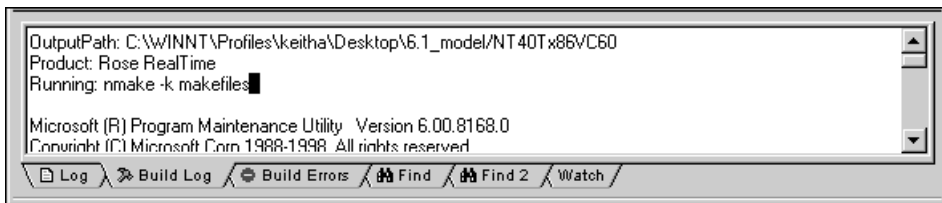
Double-clicking usually brings you to the error source.

## Build Log Tab

The **Build Log** tab stores the contents of the compilation and code generation log. Select **View > Output** and click the **Build Log** tab to open it. Compilation or code generation messages are posted to the **Build Log** tab regardless of whether it is visible.

You can save the contents of the **Build Log** tab to a file. You can also choose to automatically save messages to a file as they are posted.

**Figure 20 Build Log Tab**



The **Build Log** tab contains the raw output stream from the build. You can examine the contents of this window to get a context on any error message displayed in the build messages list.

## Saving Build Output to a Log File

You can capture all of the build output to a log file so that you can process the output later.

**To automatically save build results to a file:**

- 1 On the **Build Log** tab in the **Output** window, right-click and select **AutoSave**.
- 2 In the **AutoSave Log** dialog, specify a name for the log file and select a location.
- 3 Click **OK**.

#### 4 Build your model.

**Note:** If you attempt to open the build log file, you may encounter a Sharing Violation message. To view the contents of the **Build Log** output file, select **AutoSave** again so that it is not set, then open the log file.

## Build Errors Tab

The **Build Errors** tab contains a parsed version of the output stream. It is important to review the **Build Log** tab because some errors cannot be parsed by the error parser.

The **Build Errors** tab contains a **Location** column that gives the class/code segment name pair. The **Context** column provides the context of the problem. The **Message** column gives a description of the problem. These messages are taken directly from the compiler error stream and therefore reflect the accuracy of the compiler that you are using. Further, errors within your code segments may lead to errors being reported in system-generated files.

Double-clicking on an error or warning in the **Build Error** tab takes you to the location in the model that caused the error or warning. See Common build errors for a short summary of common generic build errors.

## Filtering Build Results

To focus on the error results only, you can filter out all warning results on the **Build Errors** tab. Right-click in the **Build Errors** tab area and select **Hide Warnings**.

## Sorting Build Results

You can sort the information in the **Build Errors** tab by clicking on a column name. You can sort based on the **Location**, **Context**, or **Message** data.

## Unknown Compiler Message Stream

It is possible that the compiler being used reports errors in ways that are not understood by Rational Rose RealTime. There are no standards for error reporting by compilers and linkers. Hence, the error parser is often targeted for a particular compiler and linker. If you are using an unsupported compiler, Rose RealTime will probably not be able to understand the error output from the parser and may inaccurately report errors. You have to rely on the raw output stream to see the direct output of the compiler, rather than going by the errors reported by the **Build Errors** tab.

## Find Tab

The **Find** tab works in conjunction with the **Find** dialog. See *Find In Model Dialog* on page 105.

## Watch Tab

Capsule instance attributes can be inspected at run-time and modified from the **watch** tab of the **Output** window. The **Watch** tab has two columns: the name of the attribute and its value.

To add an attribute instance, or variable, to the watch window, open a state monitor and drag-and-drop the attribute from the **Attributes** folder into the watch window.

You can also edit the value of a variable by selecting the **Value** field then entering another value for the variable.

## Refreshing the Watch Values

The watch values are refreshed when a message is received by the state of the capsule instance. If the state monitor the watch was created is closed, the watch value stops being updated. If the state monitor is closed, you can manually force an update of a watch value by right-clicking on the watch item and selecting **Refresh** from the popup menu.

## Specification History Window

---

Use the Specification History window to record Specification dialogs opened (up to the last 1000) from within the toolset. You can easily navigate between the opened Specification dialogs, and open and close them.

To open the **Specification History** window, click **View > Specification History**.

Specifications opened by the user will be recorded and will form the specification history. The **Specification History** list can contain up to 1000 entries; however, you can change the value of the **History level** option on the **General** tab to modify the length of the list. The default value is 25; the minimum value is 0 and the maximum is 1000.

The **Specification History** window can be docked or undocked. By default, this bar is hidden and docked to the right side of the Rational Rose RealTime frame window. The current position in the **Specification History** list is always highlighted. If multiple items are selected in the list, first item in the list is the current item.

## Locking Specification Dialogs

The Specification History window displays an icon opposite each entry in the list to indicate the type of element for which Specification dialog was opened. The display shows the element's short name and its status (locked or unlocked).

A locked entry in the **Specification History** list means that the Specification dialog will not be "pushed" out of the list if the list exceeds the maximum length (1000 entries), and that it will be saved and loaded with the workspace.

## ToolTips

The ToolTips for the elements whose Specification dialogs appear in the list display the fully qualified name for the element and, if it has any context, the short name of the context displays on the second line of the ToolTip.

## Keyboard Shortcuts

For a printable list of shortcut keys, see *Keyboard Shortcuts* on page 565.

By default, the current Specification dialog will close before the previous or next Specification dialog will display. If you open a Specification dialog using the toolset (and not using the previous and next shortcut key commands, that specification appears on the top of the Specification History list. If that Specification dialog already exists in the Specification History list, the earlier entry is removed from the list.

**Note:** The shortcut keys for the **Specification History** window, [Shift+]Alt+{PgDn | PgUp}, are not available in the Add-ins.

The previous and next shortcut key commands "wrap" around the list. If specification corresponding to the current item is not open, it will be opened first.

Most of the operations on the elements of Specification dialogs in the list will be performed using the shortcut menu. Operations are performed on the selected elements only, and multi-selection is permitted.

## Specification History Shortcut Menu

The shortcut menu for the **Specification History** window contains the following options:

- **Open** - Opens the selected **Specification** dialog. Alternatively, you can double-click on the Specification dialog name in the list.
- **Close** - Closes the selected **Specification** dialog.
  - Note:** To close multiple dialogs, press CTRL and click on the names in the list to select the dialogs, then click **Close**. To delete all Specification dialogs in the **Specification History** list, right-click in the list, select **Select All**, right-click in the list again, then click **Close**.
- **Lock** - Toggles the "Locked" status for the selected **Specification** dialog. If you have multiple Specification dialogs selected that are not currently locked, selecting this option locks them. If all selected Specification dialogs are currently locked, selecting this option unlocks them.
- **Refresh** - Removes elements from the list that can no longer be found in the model.
- **Delete from history** - Deletes the selected **Specification** dialog from the **Specification History** list.
- **Select All** - Selects all items in the **Specification History** list
- **Select in Browser** - select the corresponding element from the **Model View** tab in the browser. This option is enabled only when one element is selected.



## Contents

This chapter is organized as follows:

- *Print Specifications* on page 119
- *Print Setup* on page 123

This chapter describes how to print from the application using the **Print Specifications** and **Print Setup** dialogs.

## Print Specifications

---

The **Print Specifications** dialog lets you print diagrams. As well, you can adjust the parameters of diagrams you want to print, including size, orientation, and layout.

The dialog has four tabs:

- **General Tab**
- **Diagrams Tab**
- **Specifications Tab**
- **Layout Tab**

All tabs contain the **Print Preview** button, which you can click to see how your diagram will appear before you route it to a printer.

### General Tab

The **General** tab contains three fields:

- **Printer Area**
- **Print Range Area**
- **Copies Area**

### Printer Area

In the **Printer** area, you can select the name of the printer you want to use from a drop-down menu. Select the **Print to File** option to print a diagram to a file, instead of routing it directly to a printer. You are prompted to specify a filename and location.

As well, there is a **Properties** button.

## Properties Dialog

Clicking **Properties** opens the **Properties** dialog, which contains two tabs: **Layout** and **Paper/Quality**.

Click the **Advanced** tab to fine-tune your printing parameters.

### Print Range Area

Use the **Print Range** area to select the **Current Diagram**, **Selected Diagrams**, and **Selected Specifications**. **Current Diagram** is the default. Clicking **Selected Diagrams** and then **Diagram Options** opens the **Diagrams Tab**. Clicking **Selected Specifications** and then **Specification Options** opens the **Specifications Tab**.

### Copies Area

The **Copies** area lets you specify the number of copies you want to print, and whether you want multiple copies collated.

## Diagrams Tab

The **Diagrams** tab contains the options that you can choose from to generate a printout of one or more diagrams.

**Note:** If all the options on the **Diagrams** tab are grayed out, click the **General** tab, then in the **Print range** box, select **Selected diagrams** and click **Diagram Options**.

The **Diagrams** tab has the following areas: **Use case diagrams**, **Class diagrams**, **Component diagrams**, **Deployment diagrams**, and **Interaction diagrams**.

The first four fields contain the following buttons:

- **Top Level** - Prints only the diagrams at the top level of the model.
- **Entire Structure** - Prints all the diagrams.
- **None** - Prints none of the diagrams.

The **Include State Diagrams** check box is not applicable unless you have chosen **Top Level** or **Entire Structure** in the **Use Case** or **Class Diagrams** fields.

The **Interaction Diagrams List Control** lets you select each object message or message trace diagram containing objects whose specifications you want to print. The **All** button selects all object and interaction diagrams in the list. The **None** button deselects all object and interaction diagrams in the list.



## Specifications Tab

The **Specifications** tab contains the options to generate a printed version of one or more specifications.

**Note:** If all the options on the **Specification** tab are grayed out, click the **General** tab, then in the **Print range** box, select **Selected specifications** and click **Specification Options**.

The **Specifications** tab has the following areas: **Use case specifications**, **Class specifications**, **Component specifications**, **Deployment specifications**, **Options**, and **Interaction specifications**.

The first three fields contain the following buttons:

- **Current** - Prints only the specifications for the current diagram.
- **Entire Structure** - Prints all the diagrams.
- **None** - Prints none of the diagrams.

The **Options** field contains the following options:

- **Selected classifiers only** - This option is available only when you select **Current** or **Entire structure** from the **Class specifications** box. If you select **Selected classifiers only**, only the specifications for the currently selected classes, capsules, and protocols in the model will print.

**Note:** To print associated code with the specification, select **Selected classifiers only** and **Operation specifications**.

- **Operation specifications** - This option is available only when you select **Current** or **Entire structure** from the **Class specifications** box. If you select **Operation specifications**, only the operation specifications associated with the classes in the indicated diagrams are printed.

**Note:** To print associated code with the specification, select **Selected classifiers only** and **Operation specifications**.

- **State specifications** - is only applicable when you have chosen **Current** or **Entire structure** in the **Class diagrams** field. When you check this box, all the state-transition specifications for all the state diagrams that are associated with the classes in the indicated diagrams are printed.

**Note:** This option controls only the printing of use case specifications. You can print the associated operation and state-transition specifications by checking the **Operations Specifications** and **State Transitions** boxes.

- **Selected associations only** - is only applicable when you have chosen **Current** or **Entire** structure in the **Class diagrams** field. When you check this box, only the operation specifications for those associations with the classes in the indicated diagrams are printed.
- **Selected components only** - is applicable only when you have chosen **Current** or **Entire** structure in this field. When you check this box, only the specifications for those components that are currently surrounded by selection handles in the indicated component diagrams are printed.
- **Selected devices only** - is only applicable when you have chosen **Current** or **Entire** structure in the **Deployment specifications** field. When you check this box, only the specifications for the selected devices are printed.
- **Selected processors only** - is only applicable when you have chosen **Current** or **Entire** structure in the **Deployment specifications** field. When you check this box, only the specifications for the selected processors are printed.

The *Interaction Diagrams List Control* lets you select each object message or message trace diagram containing objects whose specifications you want to print. The **All** button selects all object and interaction diagrams in the list. The **None** button deselects all object and interaction diagrams in the list.

## Layout Tab

The **Layout** tab contains the options that you can choose from to change the position and size of the diagrams you want to print. If your print job is larger than the available paper, you can tile your work so that it is spread across several pieces of paper. Assemble the separate pages to create the whole image.

The **Layout** tab contains two areas: **Positioning** and **Options**.

### Positioning area

The Positioning field contains the options that you can choose from to change the size of the diagrams you want to print.

- **As In Diagram** - Prints diagram as you see it on screen.
- **Fit To Page** - Resizes each diagram to a single page.
- **Tile** - Enables the **Options** field.

## Options Area

The **Options** field contains the following options:

- **Overlap** - Lets you set the percentage of the images on each tile overlap on adjacent tiles.
- **Print Crop Marks** - Lets you align tiled printouts.
- **Preserve Aspect Ratio** - Lets you maintain the diagram's proportions.

## Scale label printing font to 90% of display font

You can specify a value by which you can scale the size of the font used for printing and for the Print Preview. For example, labels that look good on your screen may exceed compartment size when printed. The valid range of values to specify is between 75 and 115. The default value, 90%, is sufficient for most printing activities.

## Print Setup

---

The **Print Setup** dialog lets you set up print options, generally. The dialog contains three areas: **Printer**, **Paper**, and **Orientation**. As well, there is a **Network** button. Clicking this button opens the **Connect to Printer** dialog, which lists shared printers on the network. Click on a particular printer to route your print jobs to it.

### Printer Area

The **Printer** area lets you select the name of the printer you want to use from a drop-down menu. You can also click **Properties** to open the **Properties** dialog.

### Paper Area

The **Paper** area lets you specify the size and source of the paper you want to use to print.

### Orientation Area

The **Orientation** area lets you choose between **Portrait** and **Landscape** orientations.



## Contents

This chapter is organized as follows:

- *Unique Ids* on page 125
- *Opening Models* on page 129
- *Opening Models from ObjecTime Developer 5.2.1* on page 132
- *Opening Rational Rose Models* on page 134
- *Importing Rational Rose Generated Code* on page 136

## Unique Ids

---

Unique ids are unique internal names associated with model elements. They are used internally by Rational Rose RealTime, and not all model elements require unique ids. Rational Rose RealTime includes a feature that helps Model Integrator by generating unique ids for those model elements that would otherwise not require them, for internal use. For Model Integrator, an element with a unique id is easier to merge.

RRTEI users will find traceability easier when they set this option. Unique ids improve the traceability of model elements of other tool integrations that use RRTEI.

It is necessary to plan and choose when to incorporate the new unique ids into the project model since virtually all controlled units will be modified implicitly. Additionally, the generated new ids are dependent on time and location. For example, generating unique ids for a given model at different times, or on different machines, produces different ids.

The following model elements do not have unique ids, unless you set this option:

- Protocol In Signals ()
- Protocol Out Signals ()
- States (CompositeState)
- Capsule Roles (CapsuleRole)
- Ports (Port)
- Port Roles (PortRole)
- Capsule Structure diagram (CapsuleStructure)
- Classifier Role (ClassifierRole)

- Transitions (Transition)
- Junction Point (JunctionPoint)
- Choice Point (ChoicePoint)
- Connectors (Connector)
- (Guards)
- (Events)
- (EventGuards)
- Parameters ()
- Element hyperlinks (ExternalDocument)

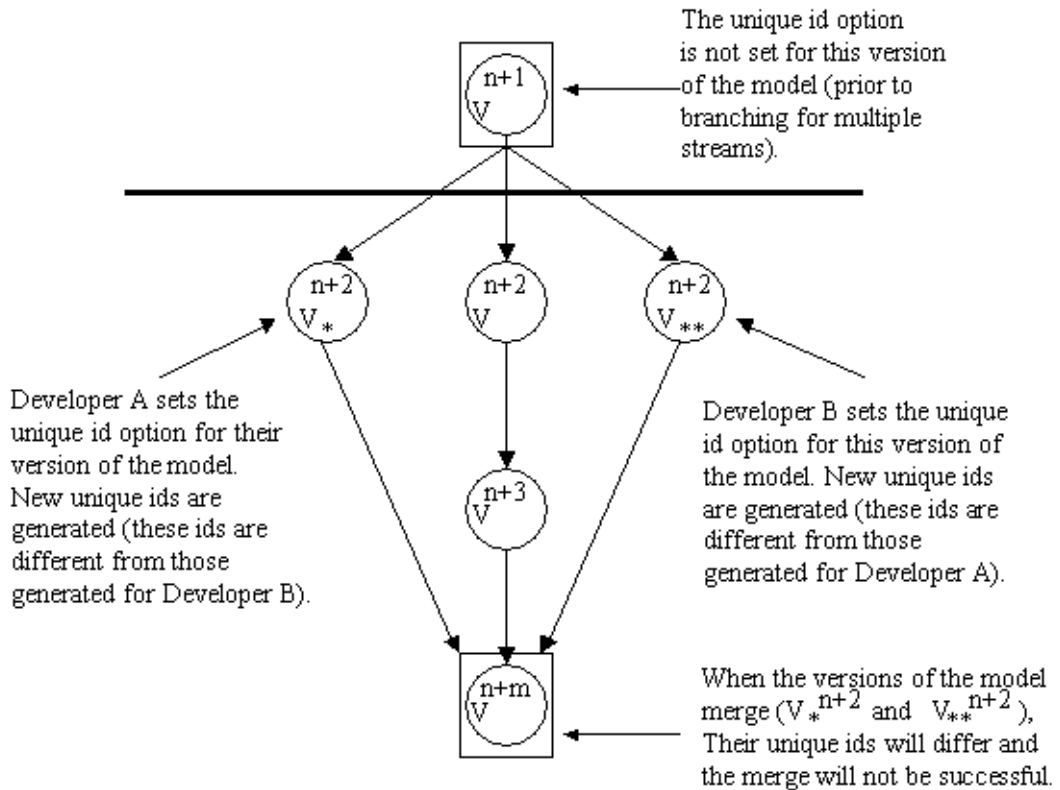
**Caution:** We strongly recommend any team involved in parallel development use this option.

**Note:** Setting this option creates unique ids for model elements that currently do not have them. This typically affects most of the model, so you will be prompted to check out those parts when setting this option.

When saving the model, the size of the affected file increases by approximately 20%, and the time to load the model also increases.

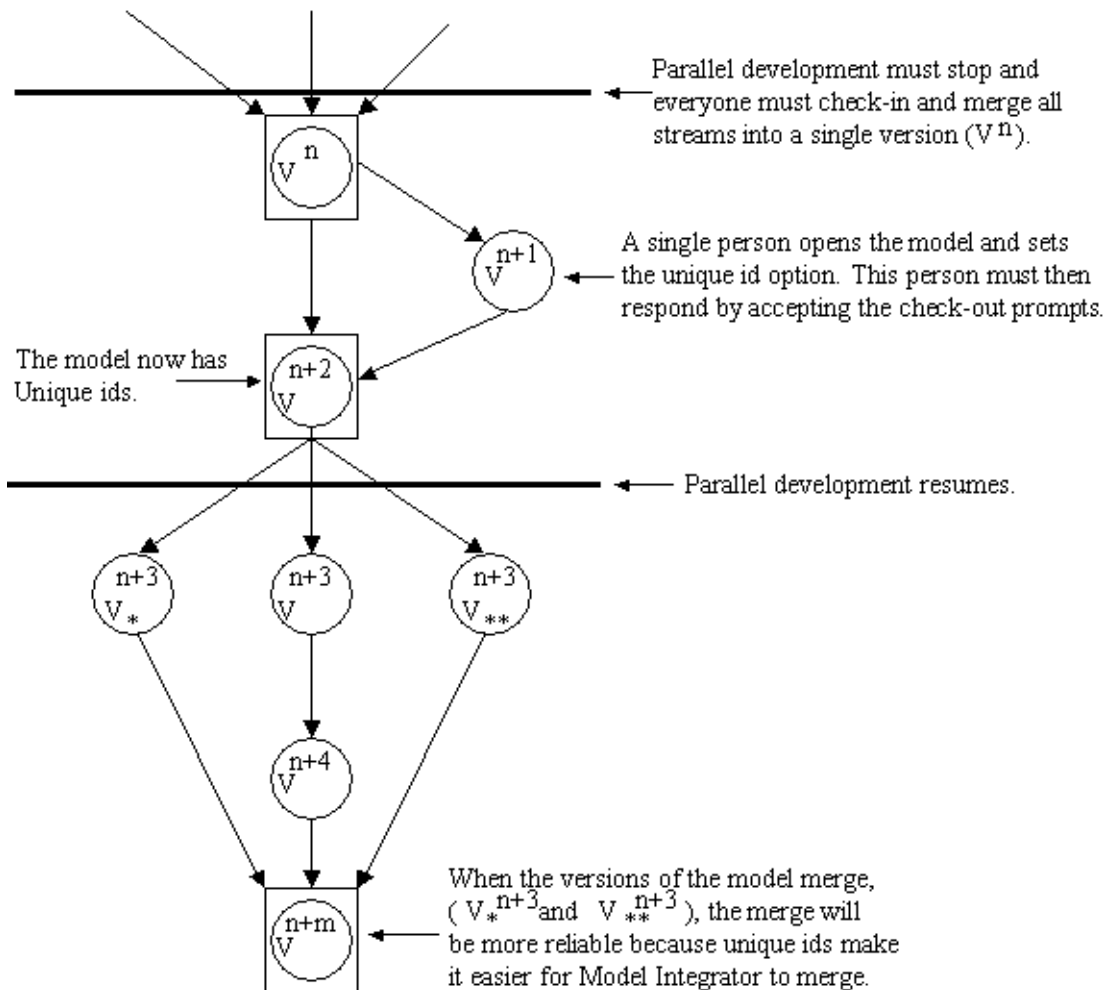
**Caution:** Do not set this option for multiple streams as shown in Figure 21; otherwise, objects with similar characteristics will be treated differently since their unique id's will differ.

Figure 21 Incorrect Merge Scenario



An example of when to set this option is shown in Figure 22.

**Figure 22 A Correct Merge Scenario**  
**Parallel Development**



**Note:** This option must be set prior to branching.

For information on how to enable the Unique ids, see *Model Specification* on page 129.


To clear the unique id option, follow the same procedure in Figure 22.

**Note:** If you clear this option, your merge results will not be as reliable.



## Opening Models

---

To open an existing Rational Rose RealTime, Rational Rose, or ObjecTime Developer model, click the **Open Existing Model** icon  on the toolbar or select **File > Open**.

A dialog appears prompting for the model file name. You can select from among different types of models to open through this dialog, including: Rational Rose RealTime models (.rtmdl), Rational Rose models (.mdl) and ObjecTime Developer models stored as linear form (.lf).

**Note:** Opening a model discards any existing model that you currently have open. The tool prompts you to save changes first.

**Note:** In the Windows version of the toolset, typing %ROBERT\_HOME% in the file name takes you to the directory that the environment variable contains. Use the same % notation on UNIX to specify environment variables.

### Model Specification

A Model Specification enables you to display and modify the properties of the top level element.

To display a Model Specification, right-click on the top level element and choose **Open Specification**.

#### Specification Content

The **Model Specification** contains the following tabs:

- **General Tab**
- **Source Control Tab**
- **Files Tab**
- **Unit Information Tab**

#### General Tab

##### Name

Identifies the name of the model.

##### Generate unique identifiers for all elements

Specifies that unique identifiers are generated for all elements in the model. By default, this option is selected.

**Note:** *Before clearing this option, ensure that you review the information on Unique Ids on page 125.*

## **Documentation**

Contains information about this model.

## **Source Control Tab**

Provides options for interacting with a source control / Configuration Management (CM) system.

## **Files Tab**

Provides a list of referenced files. The files list pop-up menu allows you to insert and delete references to files or URLs.

**Note:** You can link external files to models for documentation purposes.

## **Unit Information Tab**

### **Owned by model**

For units, it indicates that the selected unit is owned by the model.

For models, it indicates that this mode is owned by another model.

For collaborations, it indicates whether the unit is owned by this collaboration, or whether it is owned by another model and shared into this model. When selected, it indicates that this collaboration is owned by another model.

### **Under source control**

Indicates whether this model has been added to source control.

### **Control new child units**

Controls whether newly created units will be individually controlled, by default.

When selected, any new child units for this element will be controlled.

### **Disallow model-relative pathnames**

Informs Rational Rose RealTime not to use the implicit and virtual pathmap symbol when saving units.

### **Scratchpad**

Indicates that the package is a scratch pad. This option is only enabled in the **Unit Information** tab for a package that is not under source control.

## Save As

For controlled units, this option saves the selected unit, and if specified, its child units (if any exist), to a new location.

If the model is under source control, Rational Rose RealTime prompts you to check out the parent unit before proceeding with the **Save As** operation.

If you cancel the **Save As** operation, the unit file names change back to the names previous to clicking the **Save As** option; however, any files already saved are not deleted from your hard drive.

You cannot undo the **Save As** operation.

Note: The containing unit of the unit you click Saved As for is identified as being modified. It is essential to save the model to update the containing unit. If the parent unit is not saved, it will point to the original file location.

## Version

Displays the version identifier for this controlled unit. If this information is not known, then "<unknown>" displays. The ability to extract this version information depends on the source control tool being used. If a unit is not under source control, then this field is not displayed.

## File Name

Displays the file name of the model, or the name of the file used to save this controllable unit. This field is not directly editable..

## A Workspace

A workspace contains basic configuration information for working with a model. This information includes the name of open model, whether source control is enabled, and any settings related to how source control and file management behave when editing the model.

The workspace information is stored in a separate file (.rtwks). When a model opens and a workspace file of the same name exists in the directory, the toolset prompts you to open the workspace instead. If you regularly work on a particular model, open the workspace corresponding to that model rather than just opening the model itself.

The following settings are stored in a workspace:

- the model being worked on
- the source control settings for this model as specified in the **Specification** dialog for the model
- file management settings

If a model is renamed, a workspace file that refers to the old model name will not open correctly. You can edit the workspace file directly and change the path name information, or open the model file without the workspace and then save the model to create a new workspace.

## User-specific Working Environment Settings (.rtusr, .rtto and .rtwks)

Rational Rose RealTime preserves user-specific working environment settings between toolset sessions. The user-specific working environment consists of:

- options specified in the **Tools > Options** dialog (for example, font size, default label filtering)
- open windows, including their size and position
- active component
- active component instances
- target observability settings such as probes, monitors, inject messages and watch variables

All of these settings are user-specific and should not be shared between users. Settings not related to target observability are saved in a .rtusr file with the same root name as the current workspace. Target observability settings are saved in a .rtto file with the same root name as the current workspace.

These files are saved whenever the current workspace is either saved or closed.

## Opening Models from ObjecTime Developer 5.2.1

---

Rational Rose RealTime can only import Linear Form files from ObjecTime Developer 5.2.1. Other kinds of files, such as binary .update or .context files cannot be imported directly into Rational Rose RealTime.

**Note:** ObjecTime 5.2.1 users must apply a patch to their toolset to export models from ObjecTime so that they can be read by Rational Rose RealTime. See Upgrades and Patches from the Download Center on the Rational Web site.

### To open an ObjecTime Developer 5.2.1 model:

- 1 The ObjecTime Developer project file must be saved as a Linear Form file (.lf)
- 2 To open an ObjecTime Developer model from Rose RealTime, select **File > Open** and choose **Linear Form (.lf)** from the **Files of Type** drop-down menu.
- 3 Select the file to open and click **Open**.

Files from versions of ObjecTime older than ObjecTime Developer 5.2 will have to be opened in ObjecTime Developer 5.2 and saved as project files first.

**Note:** Opening a new model discards any existing model that you have. The toolset will prompt you to save changes.

### Importing requirements

Requirements captured in ObjecTime Developer Models can be converted through a requirements-specific patch for 5.2 and 5.2.1. An HTML file is generated that contains the actual requirements from the OTD models. Links to these requirements are converted when the actual model is imported into Rose RT. The HTML requirements file is stored outside of the Rose RealTime toolset. Place the file in your configuration management library for storage purposes.

See the *ObjecTime Developer Conversion Guide* for information on converting from ObjecTime Developer.

## Limitations and Restrictions

When an ObjecTime Developer model is opened in Rational Rose RealTime, the following elements may not be converted:

- **Dependencies** - The dependencies list for classes in ObjecTime Developer is not converted. Dependencies must be recreated using the **Build > Add Class Dependencies** command. This runs a script that checks the model elements for dependencies and adds them. It does not, however, find references that exist only in detailed code.

# Opening Rational Rose Models

---

## Before Starting

Rational Rose RealTime can open files (.mdl files) saved with Rational Rose 4 and on.

## Fixing a Model

When importing a model from Rose into Rose RealTime, you are encouraged to resolve any model errors in Rose (**Tools > Check Model**) before trying to import the model. In particular it is important to fix unresolved references. In general, Rose is not concerned as much about unresolved references; however, they are very important in Rational Rose RealTime as they can result in incomplete code generation and compilation errors.

## To open a Rational Rose model:

- 1 To open a Rational Rose model from Rational Rose RealTime, select **File > Open** and choose **Rose Model (\*.mdl)** from the **Files of type** drop-down menu.
- 2 Select the file to open and click **Open**.

Files from Rational Rose versions older than Rose 98 must first be opened in Rose 98 and saved.

**Note:** Opening a new model discards any existing model that you have. The Rational Rose RealTime will prompt you to save changes.

## Import Log Messages

The following messages may appear in the **Log** after a Rose model has been imported.

**Message:** Warning: Renamed elementClass "oldElementName" to "newElementName".

**Description:** A loaded model element has been renamed to conform with Rational Rose RealTime naming requirements. Double-clicking on the warning in the log may (or may not) display the renamed element.

**Message:** Error: Unresolved reference from... to... by...

**Description:** The toolset was unable to resolve a reference between two model elements. This is usually the result of loading an incomplete model, for instance when the user has updated only part of a model from CM. The rest of the model needs to be loaded in order for the reference to be resolved. However, in some

instances (where toolset stability is an issue) the unresolved model element is removed from the model. If this is the case, the deletion is also recorded in the log window.

**Message:** Error: Error reading file fileName at line lineNumber or Error message detail.

**Description:** The error message detail may contain validation errors originating from the internal meta-model, which are not covered here. Possible error message details that originate from the petal reader are listed below.

**Message:** Invalid syntax.

**Description:** The file contents cannot be read by the toolset. The user should send the file to customer support with a description of what they were doing when the file was created.

### Example

Imported a Rational Rose model, made some changes to the **Component View**, now the file will not reload in Rational Rose RealTime.

## Limitations and Restrictions

When a Rational Rose model is opened in Rational Rose RealTime, the following elements are not converted:

- Importing Rational Rose models containing controllable units is not supported  
Load the model with controllable units in Rational Rose. Export the model into a single .ptl petal file. Import the .ptl file into Rose. Save the model as a .mdl file in Rational Rose. Open the .mdl file in Rational Rose RealTime.
- Three-tier class diagrams are not supported in Rational Rose RealTime.  
Rational Rose RealTime skips over three-tier class diagram making it unnecessary to remove them before importing.
- Rational Rose elements that are not supported are written to the Documentation field in Rational Rose RealTime.

## Importing Rational Rose Generated Code

---

Source code generated from a Rational Rose model and has been edited within the preserved regions may be imported.

### To import Rational Rose generated code:

- 1 Verify that the Rational Rose .mdl file is not newer than the generated code. If so, regenerate the code.
- 2 Open the Rational Rose model (see *Opening Rational Rose Models* on page 134).
- 3 Select **Tools > Import Code**.

If code was generated from this model using Rational Rose and the model was saved after the code generation was performed, a "Rational Rose Code Import" window displays. Otherwise, the "There are no cpp or h files available for import" message displays.

The **Rational Rose Code Import Window** lists all the .cpp and .h files generated from the model, and lets you select all or a subset of the files. It also displays the classes that will be affected by each file that is selected. After a file has been imported it will not be listed if code importation is repeated.

- 4 After you have completed importation and are satisfied with the results, save the model.

### Limitations and Restrictions

- No action is taken on empty preserved regions. As a result, constructors, destructors, and operators that are generated by Rational Rose, which have empty preserved regions, are be added to the model.
- Use of the **Code Name** properties for classes and operations may cause inconsistent naming in the generated code. The inconsistencies may cause compile time errors that can be resolved manually.



## Contents

This chapter is organized as follows:

- *Creating a Use Case Diagram* on page 137
- *Using the Use Case Diagram Editor* on page 138

## Creating a Use Case Diagram

---

You create use case diagrams in the **Use Case View** of the model browser. A **Main** use case diagram is always present in the Use Case view. Use the **Main** use case diagram to describe the relationships between the primary actors and use cases in the system. You can create other use case diagrams, as required.

### To edit the Main use case diagram:

- 1 Double-click the **Main** diagram in the **Use Case View** package in the Model View Tab.

The **Use Case** diagram editor appears (see *Using the Use Case Diagram Editor* on page 138).

- 2 Place actors and use cases in the diagram by dragging them from the model browser, or by using the tools in the **Use Case Diagram Toolbox**.
- 3 Draw relationships among actors and use cases using the toolbox.

### To create a new use case diagram:

- 1 Right-click on the **Use Case View** package (or any sub-package) in the model browser.
- 2 Select **New > Use Case Diagram** from the popup menu.

Enter the name of the use case diagram.

## Using the Use Case Diagram Editor

---

Use case diagrams present a high-level view of how a system is used as seen from an outsider's (or actor's) perspective. These diagrams depict system behavior (also known as use cases). A use case diagram may depict all or some of the use cases of a system.

A use case diagram can contain:

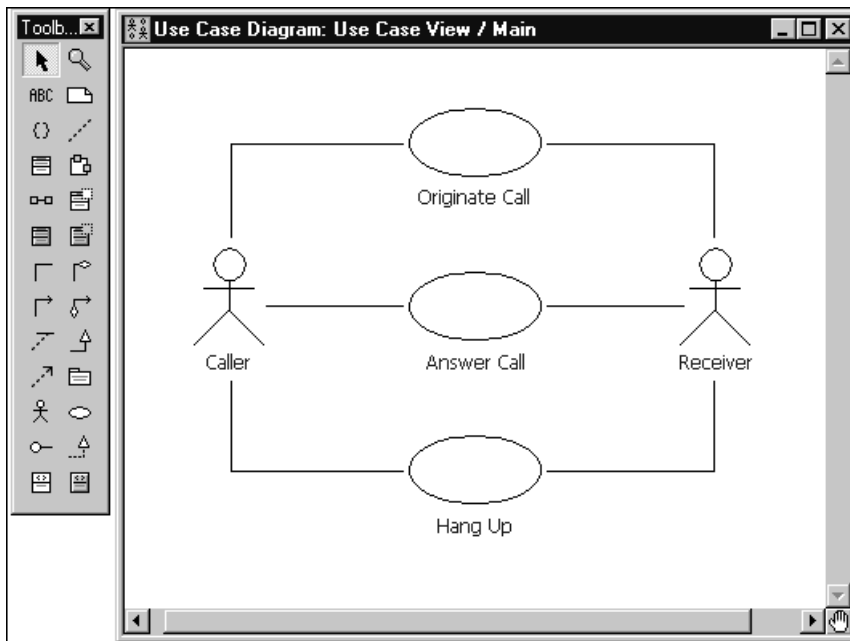
- actors ("things" outside the system)
- use cases (system boundaries identifying what the system should do)
- interactions or relationships between actors and use cases in the system including associations and generalizations

Use case diagrams can be used during analysis to capture the system requirements and understand how the system should work.

The use case diagram editor is used to create a diagram showing use cases and the relationships among use cases, actors and classes. The use case diagram consists of two parts: the diagram area and the Use Case Diagram Toolbox.

The Title bar shows the full name of the Class diagram.

**Figure 23 Use Case Diagram Editor**



## Usage Tips

Typically, you add a set of related use cases and actors to the diagram. Then, draw the relationships among use cases and actors by selecting one of the relationship tools in the toolbox, selecting one of the related elements and dragging on to the other related element.

Although you have the full set of class diagram tools at your disposal in the **Class Diagram Toolbox**, there are a limited number of relationships that should be applied to use cases. Valid relationships between use cases are: includes, extends and generalizes. However, there are no specific tools for includes and extends relationships. These should be modeled as unidirectional associations with stereotypes or stereotyped generalizations (UML 1.1).

Relationships between actors and use cases should be modeled as associations or directional associations.

**Note:** When naming actors, be aware that actors are stereotyped classes, and there is only one name space for all classes in Rational Rose RealTime. For example, if you name an actor **Server**, you will not be able to create another class named **Server** in the Logical View because there will be a name conflict. We suggest using a naming convention, such as adding an ending like "**\_actor**" to actor names.

## Use Case Diagram Toolbox

The use case diagram toolbox is the same as the *Class Diagram Toolbox* on page 149.



## Contents

This chapter is organized as follows:

- *Creating a Use Case* on page 141
- *Creating an Actor* on page 143

## Creating a Use Case

---

### To create a new use case:

- 1 Right-click on the **Use Case View** in the **Model View** tab in the browser.
- 2 Select the **New > Use Case** menu option.

A new use case is created with a default name of **NewUseCase1**.

- 3 Begin typing to change the name.

You can also create new use cases using the **Use Case** tools for the **Use Case** diagram. The use case can then be filled out using the **Use Case Specification** dialog. To access the specification dialog, double-click on the use case in the model browser.

## Use Case Specification

A **Use Case Specification** enables you to display and modify the properties and relationships of a use case in the current model.

To display a **Use Case Specification**, double-click on any icon representing the use case or right-click on the use case in the model browser and chose **Open Specification** from the model browser.

## Specification Content

The **Use Case Specification** contains the following tabs:

- General Tab
- Diagram Tab
- Relations Tab
- Files Tab

## General Tab

In addition to the elements found in standard **Specification Dialogs**, the **General** tab contains:

### Name

A use case name is often written as an informal text description of the external actors and the sequences of events between elements that make up the transaction. Use-case names often start with a verb. The name can be entered or changed on the specification or directly on the diagram.

### Package

This static field identifies the package to which the components belong.

### Stereotype

A stereotype label. A stereotype represents the subclassification of an element. For example, an actor is a stereotype of a class. Some stereotypes are already predefined, but you can also define your own.

### Rank

The Rank field prioritizes use cases. For example, you can use the rank field to plan what iteration in the development cycle a use case should be implemented.

### Abstract

An abstract notation indicates a use case that exists to capture common functionality between use cases (uses) and to describe extensions to a use case (extends).

### Documentation

Provides a description for this Use Case.

## Diagram Tab

### Diagrams

The **Diagrams** box lists all the diagrams owned by the use case. The diagram list consists of two columns. The first (unlabeled) column displays the diagram icon type for the diagram. The second column displays the diagram name. To insert a new diagram in the list, right-click and select one of the **Insert** options from the shortcut menu that corresponds to the diagram type.

## Relations Tab

### Relations

The **Relations** box lists all the relationships associated with the selected use case. The client and supplier names and type icons display to the right of the relation name. Double-clicking on any column in a row displays the element's specification.

## Files Tab

Contains a list of referenced files. The files list shortcut menu allows you to insert and delete references to files or URLs.

**Note:** You can link external files to model elements for documentation purposes.

## Creating an Actor

---

You can create Actors in the **Use Case View** of the **Model View** tab in the browser.

### To create a new actor:

- 1 Right-click on the **Use Case View** package in the Model View Tab.
- 2 Select the **New >Actor** menu option.  
A new actor is created with a default name of NewClass1.
- 3 Click on the new actor to change its name.

Actors can also be created using the actor tool in the **Use Case Diagram Editor** or **Class Diagram Editor**.

**Note:** An actor is a stereotype of a class. You can define many of the same properties on an actor as you can on any other class. To add to the actor's definition, double-click on the actor to open the Actor Specification dialog.

## Actor Specification

An Actor Specification looks identical to a Class Specification, except that the stereotype field is set to actor. However, some of the fields in the class specification are not applicable to actors and are therefore disabled.



## Contents

This chapter is organized as follows:

- *Creating a Class Diagram* on page 145
- *Creating Relationships* on page 153
- *Creating Association Relationships* on page 154
- *Creating Aggregation Relationships* on page 160
- *Creating an Association Class* on page 162
- *Aggregation Specification* on page 162
- *Creating Inheritance Relationships* on page 162
- *Creating Dependency Relationships* on page 167
- *Creating Reflexive Relationships* on page 170
- *Changing the Directionality of an Association* on page 170
- *Creating Package Relationships* on page 170
- *Creating Realize Relationships* on page 171
- *Inserting Dependencies, Generalizations, and Realizations on the Relations Tab* on page 172
- *Adding and Hiding Classes, and Filtering Class Relationships* on page 178
- *Using State Machine Code Generation for Classes* on page 178
- *Generation of Parameterized and Instantiated Classes* on page 200

## Creating a Class Diagram

---

Class diagrams are created in the Logical View of the Model browser. A **Main** class diagram is always present in the Logical view. The Main class diagram should be used to describe the relationships between the primary packages and a layered system. Other class diagrams can be created to communicate key relationships within portions of the model.

### To edit the Main class diagram:

- 1 Double-click on the Main diagram in the Logical View package in the **Model View** tab in the browser. The **Class Diagram** editor appears.
- 2 Place classes, packages, capsules, and protocols in the diagram by dragging them from the model browser, or by using the tools in the toolbox.
- 3 Draw relationships and associations among the classes, packages, capsules, and protocols using the toolbox.

### To create a new class diagram:

- 1 Right-click on the Logical View package in the **Model View** tab in the browser.
- 2 Select **New > Class Diagram** from the menu.
- 3 Enter the name of the class diagram.

There are several additional topics on creating relationships between model elements in the class diagram:

- Creating Association Relationships
- Creating Aggregation Relationships
- Creating Inheritance Relationships
- Creating Dependency Relationships
- Creating Reflexive Relationships
- Creating Package Relationships
- Defining multiplicity in relationships

## Using the Class Diagram Editor

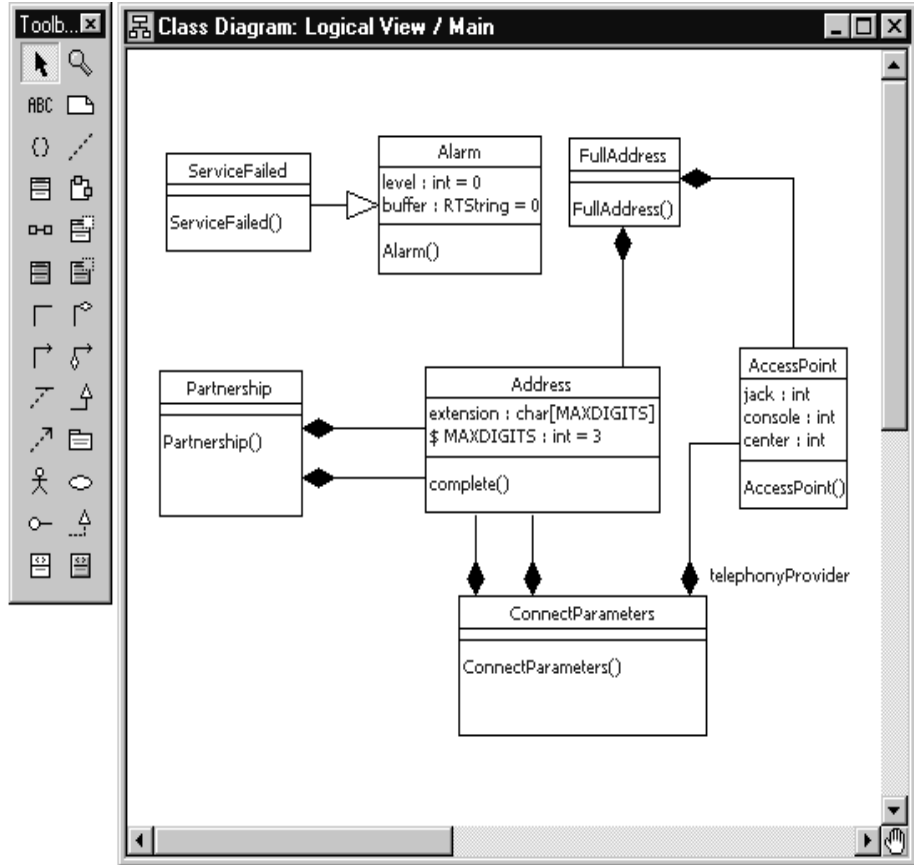
The class diagram editor is used to create a diagram showing classes and associations among the classes. The class diagram consists of two parts:

- the diagram area
- the Class Diagram Toolbox

Elements of the class diagram, such as classes, capsules, use cases and associations, are added using the toolbox.

The window title bar shows the full name of the class diagram.

Figure 24 Class Diagram Editor



Class diagrams contain icons representing classes, capsules, protocols, packages, interfaces, and their relationships. You can create one or more class diagrams to depict the classes at the top level of the current model; such class diagrams are themselves contained by the top level of the current model. You can also create one or more class diagrams to depict classes contained by each package in your model; such class diagrams are themselves contained by the package enclosing the classes they depict, the icons representing logical packages and classes in class diagrams.

Every class is assigned to a logical package. When you create a class using a creation tool from the class diagram toolbox, the class is assigned to the logical package containing the class diagram.

**Note:** If you select a label, the tether to that label displays for a short time only. To view the tether for longer periods, press and hold the left mouse button. The tether to the label will display for as long as you hold the left mouse button. If you start to drag the label, the tether is replaced with a tether and tracking box.

## Diagram Entities

There are four types of entity that you can place on a class diagram:

- Classes
- Capsules
- Protocols
- Packages

## Relationships

There are four basic kinds of relationship you can create through the class diagram. Refer to the following topics:

- *Creating Association Relationships* on page 154
- *Creating Aggregation Relationships* on page 160
- *Creating Dependency Relationships* on page 167
- *Creating Inheritance Relationships* on page 162

## Creating Capsule and Protocol Aggregations on the Class Diagram

There are some things that you can do on both the class diagram and the capsule structure diagram, including adding capsule ends and ports. Defining aggregation between a container capsule and a contained capsule results in the creation of a capsule end inside the container capsule. Defining aggregation between a capsule and a protocol results in the creation of a port as part of the capsule. Capsule structure changes made on the class diagram are automatically reflected in the structure editor. Changes made on the structure editor are only reflected on a class diagram if the model elements involved are placed on a class diagram.

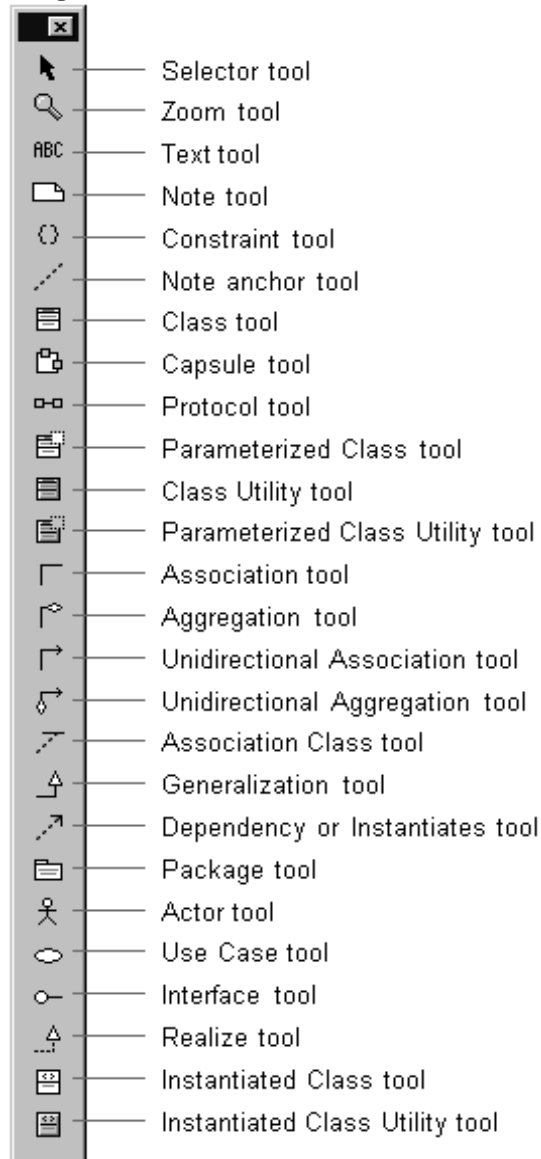
## Using the Class Diagram to Visualize Existing Relationships

You can visualize the existing relationships among these entities. Dragging capsule and protocol classes from a model browser onto a class diagram causes the tool to draw any existing relationships between these elements. For example, if a capsule aggregates another capsule, dragging the two capsule classes on to a class diagram will draw the relationships bases on the filter options chosen in the **Filter Relationship** menu command in the **Query** menu.

## Class Diagram Toolbox

The class diagram toolbox contains the following tools (they are not all displayed by default):

**Figure 25** Class diagram toolbox



**Selector**

Use to select objects for moving, resizing, and so forth.

**Zoom Tool**

Use to zoom in on a portion of the class diagram. Click on the tool and then click on the part of the diagram you want to zoom in on.

**Text Tool**

Use to add text to the class diagram.

**Note Tool**

Use to annotate the diagram with textual notes. This is useful for marking up the diagram with explanations, review comments, and so forth. You can drag and drop a diagram or external document from the browser onto a note. Notice that the name of the diagram or external document is underlined. If you double-click on the note, the diagram or external document is opened. You can undo and redo this command.

**Constraint Tool**

Use to add UML constraints to the class. A constraint can be anchored to a view element by using the anchor tool. Currently, constraints do not have any semantic meaning to the tool. There are RRTEI APIs to add or remove, and enumerate constraints in a diagram.

**Note Anchor Tool**

Use to anchor a note to a particular element on the class diagram. (See *Using the Class Diagram Editor* on page 146.)

**Class Tool**

Use to place a class on the class diagram. (See *Using the Class Diagram Editor* on page 146.) Pops up a pick list allowing you to choose from an existing class or create a new class.

**Capsule Tool**

Use to place a capsule class on the class diagram. (See *Using the Class Diagram Editor* on page 146.) Pops up a pick list allowing you to choose from an existing capsule class or create a new capsule class.

### **Protocol Tool**

Use to place a protocol class on the class diagram. (See *Using the Class Diagram Editor* on page 146.) Pops up a pick list allowing you to choose from an existing protocol class or create a new protocol class.

### **Parameterized Class Tool**

Use to place a parameterized class on the class diagram. (See *Using the Class Diagram Editor* on page 146.) Displays a pick list allowing you to choose from an existing class or create a new class.

### **Class Utility Tool**

Use to place a utility class on the class diagram. (See *Using the Class Diagram Editor* on page 146.) Displays a pick list allowing you to choose from an existing class or create a new class.

### **Parameterized Class Utility Tool**

Use place a Parameterized class utility on the class diagram. (See *Using the Class Diagram Editor* on page 146.) Displays a pick list allowing you to choose from an existing class or create a new class.

### **Association Tool**

Use to draw an association between two classes on the class diagram. (See *Using the Class Diagram Editor* on page 146.) Associations can be created between classes (including class utility, parameterized class, and so on), between capsule classes, and from capsule classes to protocol classes.

### **Aggregation Tool**

Use to draw an aggregation between two classes on the class diagram. (See *Using the Class Diagram Editor* on page 146.) Associations can be created between classes (including class utility, parameterized class, and so on), between capsule classes, and from capsule classes to protocol classes. See *Creating Aggregation Relationships* on page 160 for more information.

### **Unidirectional Association**

Use to draw a unidirectional association between two classes on the class diagram. A unidirectional association is simply an association with navigability limited to one direction. Associations can be created between classes (including class utility, parameterized class, and so on), between capsule classes, and from capsule classes to protocol classes. See *Creating Relationships* on page 153 for more information.

## **Unidirectional Aggregate Association**

Use to create an association that is unidirectional in the direction it was drawn, with an aggregation at the end. Any association between classes can be converted into this through the specification dialog, as well.

## **Association Class**

Use to link a class with an association between two other classes on a class diagram. Use the Association Specification (by double-clicking on the association after it has been drawn) to specify details of the association semantics. Using the link attribute tool automatically sets the Link Element field on the association to be the class joined to the association with the link attribute tool.

## **Generalization**

Use to indicate that one element is a generalization of another. This is primarily used to indicate a superclass/subclass relationship between classes. Draw the relationship from the specializing element to the generalizing element (that is, from subclass to superclass). Use the Generalize Specification (by double-clicking on the generalization after it has been drawn) to specify details of the generalization semantics.

Adding a generalizes relationship between two classes (including capsule and protocol classes) results in one class being generated as a subclass of the other at code generation time.

## **Dependency or Instantiates**

Use to indicate that one element is dependent on another. This is primarily used to indicate a compilation dependency between classes. Draw the relationship from the dependent element to the dependent-upon element. Use the Dependency Specification (by double-clicking on the dependency after it has been drawn) to specify details of the dependency semantics.

Adding a dependency relationship between two classes (including capsule and protocol classes) results in the dependent class including the .h file of the dependent-upon class.

## **Package**

Use to add a package to the diagram. The package is given a default name such as 'NewPackage1'.



### **Actor**

Use to place an actor on a diagram. Displays a pick-list allowing you to select from available classes or create a new class.

### **Use Case**

Use to place a use case on a diagram. This creates a new use case with a default name such as 'UseCase1'.

### **Interface**

Use to place an interface on a diagram. Displays a pick list allowing you to select a class or create a new class.

### **Realize**

Use to indicate that a class realizes an interface or a use case. Draw the relationship from the realizing element to the element being realized.

### **Instantiated Class**

Use to place an Instantiated class on the class diagram. Displays a pick list allowing you to choose from an existing class or create a new class. **There is no code generation support for instantiated classes.**

### **Instantiated Class Utility**

Use to place an Instantiated class utility on the class diagram. Displays a pick list allowing you to choose from an existing class or create a new class. **There is no code generation support for instantiated class utilities.**

## **Creating Relationships**

---

Relationships among modeling elements take many forms. Most relationships imply an interaction or a dependency between two model elements. The term 'class' in the following descriptions includes capsule and protocol classes as well as "data" classes.

See the following topics for the type of relationship you are interested in creating:

- An association is a relationship between two classes (including capsule and protocol classes). An association relationship may have a number of different implications for the generated code, or it may not result in any generated code at all, depending on the specific properties defined on the association. See *Creating Association Relationships* on page 154.
- An aggregation is a more specific form of association that indicates that one class is part of a larger, composite class. That is, one or more instances of one class are considered to be owned by (and are created and destroyed under the control of) an aggregate class. See *Creating Aggregation Relationships* on page 160.
- A dependency relationship indicates that the implementation of one class or package depends on the existence of the definition of another class or package (or some aspect of that class). See *Creating Dependency Relationships* on page 167.
- A generalization relationship indicates that one class inherits properties from (is a subclass of) another class. See *Creating Inheritance Relationships* on page 162.
- A reflexive relationship is one in which an instance of a class may also have associations with other instances of the same class. See *Creating Reflexive Relationships* on page 170.

## Creating Association Relationships

---

Association relationships indicate some form of interaction between two classes. Typically, the association relationship indicates that instances of those classes communicate with each other at run-time.

To create an association relationship in the class diagram editor:

- 1 Click on one of the two association icons in the class diagram toolbox: the bi-directional association or the uni-directional association. (For more on directionality, see *Changing the Directionality of an Association* on page 170).
- 2 Click on one of the two classes involved in the association.
- 3 Drag the association line on top of other class.

An association line appears between the two classes.

## Association Properties

After an association is created between two classes, each of those classes is said to play an end in the association.

There are several properties surrounding the association, including properties of the two ends involved in the association. These properties can be edited by double-clicking on the association to bring up the Association Specification, or by selecting the association and right-clicking. The right-click menu includes properties specific to the end closest to where the mouse was clicked.

The class you terminated the association line on is referred to as End A. The class you clicked on to start drawing the association end is referred to as End B. You can name these ends explicitly through the **Association Specification** dialog or the shortcut menu.

## Association Specification

An association represents a semantic relationship between two classes. To display the association specification, double-click any association in a class diagram.

### Specification Content

The **Association Specification** dialog consists of the following tabs: General Tab, Detail Tab, End A Detail, End B Detail, End A General, End B General, and language-specific tabs.

### General Tab

#### Name

A name for the association. The name label appears on the class diagram.

Effect on generated code: None.

#### Parent

The parent the component belongs to (its package) is displayed in this non-editable field.

## Stereotype

A stereotype represents the subclassification of an element. It represents a class within the UML metamodel itself (that is, a type of modeling element). Some stereotypes are already predefined, but you can also define your own to add new kinds of modeling types.

Effect on generated code: None.

## End A / B

Labels the ends with names that denote the purpose or capacity wherein one class associates with another. This field is the same as the End field on the End A General and Detail and End B General and Detail tabs. See the End Detail tab for more information.

## Element A/B

Specifies the classes of the two elements that this association associates. This field cannot be edited.

**Note:** Click the **Element A** or **Element B** hot link to open the **Specification** dialog for that object.

## Detail Tab

### Derived

Indicates whether the association is computed or implemented directly. The element name for a derived element is adorned by a “/” in front of the name.

Effect on generated code: No code is generated for derived associations.

### Association Class

Lists the attributed associations linked to the association. These attributed associations apply to the association as a whole. It identifies a class representing the association between the two elements.

Effect on generated code: Each of the end classes has a member generated to point to or contain an instance of the link class, depending on the settings of the containment property. The link class has members generated to point to or contain each of the ends of the association.

**Note:** If the **Association class** box contains a class, the corresponding label becomes a hot link to the Specification dialog for that class.

## Name Direction

Defines the direction of an end. There are three options listed in the drop-down menu associated with the field: <non-directional>, End A and End B.

Effect on generated code: None.

**Note:** If the **Name direction** box contains a class, the corresponding label becomes a hot link to the Specification dialog for that class.

## Constraints

The constraint is an expression of some semantic condition that must be preserved while the system is in a steady state. The constraint on the Detail tab applies to the association as a whole, while the constraint on the Detail A or Detail B tab applies to a particular end.

To apply a constraint, click in the Constraint field and enter the text. Constraints are displayed notationally, surrounded by braces under the end for which it applies.

Effect on generated code: None.

## End A and B General Tabs

### End A / B

Labels the end with a name that denotes the purpose or capacity wherein one class associates with another. This field is the same as the End A and End B fields on the General tab.

Effect on generated code: The end name is generated as a member of the class at the other end of the association. That is, if the class at End A is class A and the class at End B is class B, and the name of End A is foo, then class B will have a member named foo of type Class A.

### Element

Describes the two elements that this association associates. This field cannot be edited.

**Note:** Click the **Element** hot link to open the **Specification** dialog for that element.

## Visibility

Specifies the visibility of the data member representing this end in the other class. Visibility options are

- **Public** - Visible to any class.
- **Protected** - Visible to this class, any subclasses of this class, and any designated friend classes.
- **Private** - Visible only to this class and any designated friend classes.
- **Implementation** - Not visible to any other classes.

Effect on code generation: If a data member is generated for this end in the other class, the member will have the visibility specified here. The member is only generated if the other field settings in the End A/B Detail tab are set appropriately.

## End A and B Detail Tabs

### End

A label for the end. This label appears beside the end on the association in the diagram. This field is the same as the End field on the General and End A and End B General tabs. See the field description on the End A/B General tab for more information.

### Element

A non-editable field that specifies the classifier for this end.

**Note:** Click the **Element** hot link to open the **Specification** dialog for that element.

### Constraints

The constraint is an expression of some semantic condition that must be preserved while the system is in a steady state. The constraint on the Detail tab applies to the association as a whole, while the constraint on the Detail A or Detail B tab applies to a particular end.

Effect on generated code: None.

### Multiplicity

The multiplicity field defines the maximum number of instances that can exist in this end of the association at any given time. See Multiplicity options for more information.

Effect on code generation: The data member for this end is declared as an array with its size being the largest possible value declared in the multiplicity. If the range is unspecified (for example, 1..\*), the containment value is forced to 'By reference' and a warning is issued if the containment value was originally set to 'By value'.

## Aggregation

The Aggregation field has three checkboxes: None, Aggregate, and Composite.

- **None** - The end is not an aggregate.
- **Aggregate** - The end is an aggregate; therefore, the other end is a part and must have the aggregation value of none. The part may be contained in other aggregates.
- **Composite** - The end is a composite; therefore, the other end is a part and must have an aggregation value on none. The part is strongly owned by the composite and cannot be part of any other composite.

Use the Aggregation field to set a direction to either all or part of the relationship among instances of these classes. Only one end of the relationship can be aggregate or composite.

To set the aggregate adornment, click on the Aggregate box in the Association Specification or click **Aggregate** through the shortcut menu. The adornment is a diamond on the relationship.

Effect on code generation: This affects how the other end is stored as a member of this class. Checking the aggregate box allows you to select a containment setting to control how the aggregation will actually be generated in code.

## Target Scope

The Target Scope field has two checkboxes: Instance and Classifier.

- **Instance** - Specifies that instances of the client own the supplier class.
- **Classifier** - Specifies that the client class - not the client's instances - owns the supplier class.

You can set this field in the specification or through the shortcut menu.

Effect on code generation: The data member is scoped to the classifier in the other end class.

## Friend

The friend field designates that the supplier class has granted rights to a client class to access its non-public parts.

Effect on code generation: This field currently has no effect on code generation. See Designating friend classes for information on how to specify friends.

## Navigable

The Navigable field indicates in which direction the association is traversed. By default, ends are bidirectional and no navigation notation is provided.

To set an end's navigation, click on the Navigable box in the Association Specification or click **Navigable** through the shortcut menu. The navigable arrowhead points in the direction of the end, unless a containment adornment is displayed. Containment adornments override navigable adornments.

Effect on code generation: If the navigation check box is not checked, it signifies that the class at the other has no visibility of the class at this end of the association; therefore, no member will be generated in the other end class.

## Keys/Qualifiers

A key or qualifier is an attribute that uniquely identifies a single target object. The attributes allow 1..n or n..n associations, and reduce the number of instances. The list box displays all keys or qualifiers currently defined.

To enter a key or qualifier, click **Insert** from the popup menu or press the insert key. An untitled entry is placed in the name and type field. To change the entry, select to highlight and type in a new name.

Effect on generated code: The Keys/Qualifiers entries currently have no effect on generated code.

for sending/receiving data.

## Creating Aggregation Relationships

---

Aggregation relationships are a form of association relationship that indicate a class (the contained class) is a part-of another class (the container, or aggregate class).

**Note:** You can click **Tools > Aggregation Tool** to quickly and easily create aggregation relationships.



### To create an aggregation relationship in the class diagram editor:

- 1 Click on the aggregation icon in the class diagram toolbox.
- 2 Click on the container class that will contain the class in the diagram.
- 3 Drag the association line on top of the contained class.

An association line appears between the two classes, and a diamond (aggregate) symbol appears beside the contained class.

Typically, aggregation indicates specific run-time constraints that exist on the relationship.

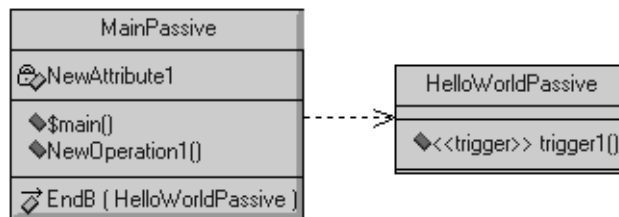
### Considerations

- An instance of the contained class cannot exist outside of an instance of the aggregate class.
- The creation of an instance of the aggregate class usually results in the automatic creation of an instance of the contained class.
- The destruction of an aggregate instance results in the automatic destruction of any contained instances.

Implementation details of the aggregation, such as whether the contained object is referenced by a pointer or embedded, can be specified through the End A and B General Tabs and End A and B Detail Tabs.

Figure 26 shows the aggregation representation in the aggregation compartment on the class diagram.

**Figure 26 Class Diagram with Aggregation**



This compartment only exists for classes and capsules. An aggregation displays in the aggregation compartment only if the aggregation causes the generation of a member variable for the class. This means that if the appropriate end is navigable, the aggregation appears in the compartments for that class or capsule.

## Creating an Association Class

---

A relationship in itself may have state and identity distinct from the instances involved in the relationship. In order to implement a relationship, it may be necessary to define a class representing the relationship.

### To create an association class:

- 1 In the class diagram editor, click on the class icon in the class diagram toolbox.
- 2 Click on the diagram to place a new class.
- 3 Enter the name of the class.
- 4 Click on the link attribute icon in the class diagram toolbox.
- 5 Click on the association class and drag the link attribute line to the association it modifies.

## Aggregation Specification

---

An aggregation represents a special bidirectional semantic relationship between two classes, wherein one or more instances of one class are contained within an instance of the aggregating class.

The aggregation specification dialog is the same as the Association Specification with the **End A aggregate** check-box turned on.

## Creating Inheritance Relationships

---

### To define an inheritance relationship:

- 1 Open the class diagram where you want the inheritance relationship to appear.
- 2 Click on the Generalization icon in the class diagram toolbox.
- 3 Click on the intended subclass.
- 4 Drag the generalization line over the intended relationship.

## Creating an Inheritance Tree

**To add other subclasses to the inheritance relationship to create an inheritance tree:**

- 1 Using the generalization tool, drag a generalization line from each intended subclass to the inheritance triangle by the intended superclass.

Two separate inheritance relationships can be merged into a tree by moving one inheritance triangle symbol on top of another.

## Exclusions

When you create a new generalization between capsules or protocols, the Inheritance Rearrangement dialog may appear prompting you to exclude new superclass properties. This allows the subclass to not inherit certain properties (state machine, capsule structure and protocol signals) defined in the superclass. This is helpful, for example, if your subclass has a state machine and you want to intelligently merge the state machines rather than just blindly inherit the superclass state machine. You can initially exclude the superclass elements, and then gradually re-inherit them as you edit your state machine.

If you select **Copy** or **Cut** from the **Edit** menu, a dialog appears warning you that items whose parents are not being cut or copied will not get pasted. You have the option of checking the box, do not warn anymore this session.

See Inheritance for more information.

## Generalize Specification

A generalize relationship between classes shows that one class shares the structure or behavior defined in one or more other classes.

The Generalize Specification consists of the following tabs: **General** and **Files**.

## General Tab

### Name

A name for the relationship.

### Owner

A non-editable field indicating the name of the subclass.

**Note:** Click the **Owner** hot link to open the **Specification** dialog for that element.

### Stereotype

Specify a stereotype to apply to the relationship.

### Visibility

Specifies the visibility of the generalization. Visibility options are

- **Public** - Visible to any class.
- **Protected** - Visible to this class, any subclasses of this class, and any designated friend classes.
- **Private** - Visible only to this class and any designated friend classes.
- **Implementation** - not visible to any other classes.

### Friendship Required

Specifies that the supplier class has granted rights to the client class to access its non-public members. In the case of a generalization, the subclass is granted friend access right to superclass members.

Effect on code generation: This field currently has no effect on code generation.

### Virtual Inheritance

Specifies that only one copy of the base class will be inherited by descendants of the subclasses.

## Inheritance in Rational Rose RealTime

You can define generalization relationships between classes (including capsule and protocol classes) in Rational Rose RealTime. When a generalization relationship is defined, the specializing class inherits the properties including all attributes, operations, state machine, signals, etc.) of the generalizing class.

For capsule and data classes, all public and protected operations are inherited, as well as all public and protected attributes.

For capsule classes, the structure elements (the ports and capsule roles) are also inherited by the specializing class.

For protocol classes, the signals are inherited as well as the state machine, if defined.

## Promoting and Demoting Elements

Capsule structure elements (ports, capsule roles and bindings), capsule and protocol state machine elements, and protocol signals can all be promoted and demoted in the class hierarchy.

For example, you can select a port from a capsule and demote it, such that it is removed from the generalizing capsule class' structure and moved into each of its subclasses. The port is no longer inherited, it becomes part of the subclass' structure and is removed from the superclass.

As another example of promoting/demoting, you can select a state in a capsule subclass and 'promote it' such that the state is moved into the superclass state machine, and is inherited by all the capsule's subclasses.

To promote an element from a subclass to its immediate superclass, right-click on the element in the browser and then click **Promote to Superclass** or **Demote to Subclass** from the shortcut menu.

## Potential Conflicts Caused by Promote/Demote

A promote or demote operation may fail if there is a name conflict in the subclass or superclass. For example, if you try to promote a state named Ready from a capsule subclass into its superclass, you will get an error if any other subclass of the superclass also has a state named Ready.

## Excluding Elements

In addition to promoting and demoting, you can also exclude certain inherited elements (the same set that can be promoted/demoted) from a capsule subclass or protocol subclass.

An excluded element is removed from the subclass diagram or properties. Note that for structure elements (ports and capsule roles), the excluded element will still be inherited in the code of the subclass, since these elements are generated as members of the superclass and automatically inherited by the subclass. This means, you should not reuse the name of any excluded element or you may cause a name conflict at compile-time.

## Reinheriting Excluded Elements

To exclude an inherited element, right-click on the element in the diagram or properties editor and click **Remove/Exclude**. If this menu entry is not available, the element cannot be excluded. You can reinherit an excluded element by right-clicking on it and selecting **Inherit**. In protocol classes, click the **Show Excluded** check box on the Signals tab to see excluded signals. In a Structure Editor or State Editor right-click on the diagram and select **Filter > Excluded** (turn off the Exclusions filter) to see any excluded elements.

## Rearranging Inheritance Hierarchies

If you choose to make a generalization relationship between two capsules or between two protocols, you will be prompted with a dialog allowing you to exclude the properties of the new superclass. See *Creating Inheritance Relationships* on page 162 for more information.

If you break a generalization relationship between capsule or protocol classes, you will be presented with a dialog option to Absorb all current superclass properties. This allows you to essentially copy the elements that the subclass had previously inherited from the superclass directly into the subclass definition and then break the inheritance relationship between the two classes.

## Inheritance Tab in Browser

Dragging and dropping items within **Inheritance View** tab results in inheritance rearrangements. Dragging an item - that does not inherit - on top of another item results in a generalization relationship between those two items.

Dragging an item that has a "specialize" relationship results in a rearrangement of that relationship. If the item being dragged supports multiple inheritance, the relationship to rearrange is determined by the position of the dragged item in the inheritance tree.

**Note:** There is no establishing of inheritance relationships between packages. If an item having a "specialize" relationship is dropped in a package, that "specialize" relationship will be deleted. For classes, user confirmation is required for deletion. For capsules and protocols, a standard inheritance dialog displays to allow you to cancel deletion.

The **Inheritance** tab supports some location rearrangements. For example, a capsule dropped on a package results in the capsule being moved in to that package, provided that the capsule does not participate in any "specialize" relationships. If it does, this type of relationship is deleted and the capsule is not moved in to the package.

**Note:** Use caution when generalizations exist on more than one diagram. If a generalization has changed its parent, and both the child and the new parent are present on more than one Class diagram, only one diagram will show new generalization. On the other diagrams, you must use **Query > Filter Relationships** and click **OK** to force these diagrams to show the updated relationship.

## Creating Dependency Relationships

---

A dependency relationship is a vague form of relationship between two classes that simply indicates that something in one class depends on the definition of something in the other class.

### To create a dependency relationship:

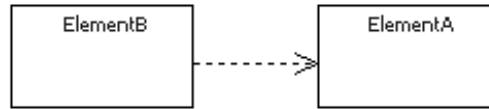
- 1 Click the dependency tool.
- 2 Click on the intended dependent class.
- 3 Drag and drop on to the class that is being depended upon.

Draw a dependency relationship between two classes, or between a class and an interface, to show that the client class depends on the supplier class/interface to provide certain services, such as:

- The client class accesses a value (constant or variable) defined in the supplier class/interface.
- Operations of the client class invoke operations of the supplier class/interface.
- Operations of the client class have signatures whose return class or arguments are instances of the supplier class/interface.

## Graphical Notation

A dependency relationship is a dotted line with an arrowhead at one end:



The arrowhead points to the supplier class. In this example, class A is dependent on class B.

## Naming

Use the relationship name to identify the type or purpose of the relationship.

## Valid Applications

You can draw a dependency relationship between logical packages.

## Add Class Dependencies Wizard

A wizard is supplied to automate the creation of dependencies between a large number of classes (for example, after loading a Rose or ObjecTime Developer model).

See Add Class Dependencies.

## Dependency Specification

The dependency relationship indicates that the client class depends on the supplier class to provide certain services. One class may use another class in a variety of ways. Typically, a dependency relationship indicates that the operations of the client access members (operations or attributes) of the supplier. Dependencies can also be drawn between packages.

You can change properties or relationships by modifying the icon on the diagram or by editing the specification.

You can also view the specification by double-clicking on the name of the dependency relationship in the Relations tab of the Class Specification.

The associated diagrams or specification are automatically updated.

The Dependency Specification contains the following tabs: General, Files.



## General Tab

### Name

A name for the dependency relationship.

### Class

A non-editable field listing the client class.

**Note:** Click the **Class** hot link to open the **Specification** dialog for that element.

### Stereotype

Specifies a stereotype to attach to the dependency.

### Friendship Required

A check box indicating whether the client class should be generated as a friend of the supplier to provide access to non-public members on the supplier.

Effect on code generation: This field currently has no effect on code generation.

### Export Control

Specifies the visibility of the dependency. Visibility options are

- **Public** - Visible to any class.
- **Protected** - Visible to this class, any subclasses of this class, and any designated friend classes.
- **Private** - Visible only to this class and any designated friend classes.
- **Implementation** - Not visible to any other classes.

Effect on code generation: None.

### Multiplicity from

Describe the multiplicity of the client side of the relationship.

Effect on code generation: None.

### Multiplicity to

Describe the multiplicity of the supplier side of the relationship.

Effect on code generation: None.

## Creating Reflexive Relationships

---

An object may sometimes need to communicate with other objects of the same class. In the class diagram, this appears as a class having a relationship with itself. This is called a *reflexive relationship*.

### To create a reflexive relationship in the class diagram editor:

- 1 Click on the association icon in the class diagram toolbox.
- 2 Click on the class with the intended reflexive relationship.
- 3 Drag the association line outside of the class border and then back over the class.

An association line appears drawn from the class back onto itself.

## Changing the Directionality of an Association

---

There are two forms of association that can be created: bi-directional and uni-directional. Bi-directional associations are highly unusual in practice in the development of applications, as a bi-directional association suggests that communication can be initiated in either direction. Most associations between classes in an application are fundamentally uni-directional; that is, an instance of one class always initiates communication to one or more instances of the other class.

### To change the directionality of an association after it has been created:

- 1 Open the association specification dialog by double-clicking on the association in the diagram.
- 2 Select the **Navigable** check box on the **End A General** or **End B General** tab to change the directionality.

## Creating Package Relationships

---

Relationships can be defined between packages. A relationship between two packages indicates that one package is dependent on another. A dependency between packages exists when one or more classes in one package initiates communication with a class or classes in another package. The first package is dependent on the second package.

**To create a dependency relationship between two packages in the class diagram editor:**

- 1 Click on the dependency icon in the class diagram toolbox.
- 2 Click on the package that will be the dependent package in the diagram.
- 3 Drag the dependency line on top of the package being depended on.

A dependency association appears between the two packages, with an arrowhead pointing from the dependent package to the package it depends upon.

## Creating Realize Relationships

---

A realize relationship between classes and interfaces and between components and interfaces shows that the class realizes the operations offered by the interface.

### Naming

Use the relationship name to identify the type or purpose of the relationship.

### Valid Applications

You can draw a realize relationship between a ClassInterface and a Component Interface. The relationship between a component and an interface can not be drawn explicitly. It is created when an interface is assigned to a component through the browser or a specification editor.

## Realize Relationship Specification

### General Tab

#### Name

A name for the Realize relationship.

#### Documentation

Use to describe the Realize relationship.

# Inserting Dependencies, Generalizations, and Realizations on the Relations Tab

---

In the **Relations** tab for capsule, class, protocol, and actor, you can now use the Context menu to insert relationships. For capsules (**Class Diagram - Capsule Specification** dialog box), you can insert dependencies and generalizations. For classes and protocols (**Class Diagram - Class Specification** dialog box), you can insert dependencies, generalizations, and realizations. For actors (**Use Case Diagram - Class Specification** dialog box), you can insert dependencies, generalizations, and realizations.

**Note:** Inserting any relation in the **Relations** tab on the **Class Specification** dialog box for a class does not update the **Class** diagram. Inserting any relation in the **Relations** tab on the **Class Specification** dialog box for an actor does not update the **Use Case** diagram.

## Inserting Dependencies

A dependency relationship specifies that a change in the specification of one element may affect another element that uses it, but not necessarily the reverse. A dependency relationship models dependencies that have not been implicitly captured by the other types of relationships in your model.

From the Context menu in the **Relations** tab, you can insert dependencies for the following:

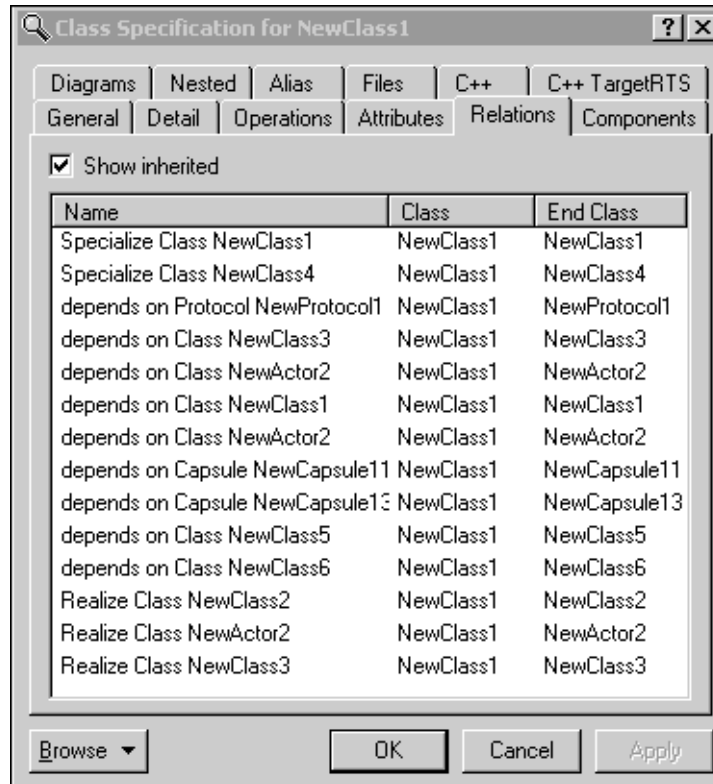
- Capsule
- Class
- Protocol
- Actor

### To insert a dependency:

- 1 In the **Relations** tab for a **Class**, **Capsule**, or **Protocol Specification** dialog box, right-click and select **Insert Dependency**.



- 2 Double-click to select an item from the drop-down list.
- 3 Click OK.



In the **Relations** tab, all dependency relationships begin with **depends on**.

## Inserting Generalizations

Generalizations are significant in that they affect from which object we inherit (such as, attributes and operations). From the Context menu In the **Relations** tab, you can insert generalizations for the following:

- Capsule
- Class
- Protocol
- Actor

You can change the inheritance from the **Class Specification** dialog box, or change the **End Class** for the generalization. This includes not only inheritance changes made in the relation page, but also changes made elsewhere in the model (such as, on diagrams and in the **Inheritance** tab in the browser). For additional information on changing the end class, see *Changing the End Class* on page 176.

Capsules are not allowed multiple inheritance. This means that the **Insert Generalization** option is only enabled for capsules if there is no existing generalization.

Protocols are not allowed multiple inheritance. This means that the **Insert Generalization** option is only enabled for protocols if there is no existing generalization.

In the **Relations** tab, the name of a generalization relationship begins with **Specialize**.

In the **Relations** tab, all generalization relationships begin with **Specialize**.

#### To insert a generalization:

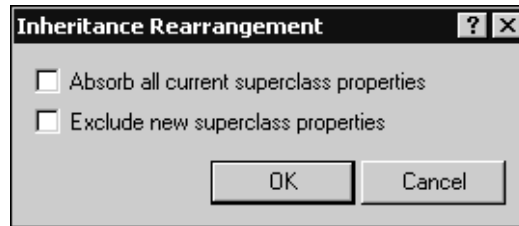
- 1 In the **Relations** tab for a **Class**, **Capsule**, or **Protocol Specification** dialog box, right-click and select **Insert Generalization**.



- 2 Double-click to select an item from the drop-down list.

**Note:** To exit from this pop-up without making a selection, click ESC.

If you create a new dependency for a capsule, class, or protocol, the **Inheritance Rearrangement** dialog box may appear after you specify the name of the new object.



- 3 Select **Absorb all current superclass properties** if you want the subclass to inherit all existing properties of its superclass. Select **Exclude new superclass properties** if you do not want the subclass to inherit certain properties (such as state machine, capsule structure, and protocol signals) defined in the superclass. This is useful for those situations when your subclass has a state machine and you want to merge the state machines rather than blindly inheriting the superclass state machine. You can initially exclude the superclass elements, and then gradually re-inherit them as you edit your state machine.

**Note:** To exit from this pop-up without making a selection, click ESC.

## Inserting Realizations

A realization relationship is a form of generalization in which only behavior is inherited. From the Context menu in the **Relations** tab, you can insert realizations for the following:

- Class
- Protocol
- Actor

In the **Relations** tab, the name of a realization relationship begins with **Realize**.

### To insert a realization:

- 1 In the **Relations** tab for a **Class** or **Protocol Specification** dialog box, right-click and select **Insert Realization**.



- 2 Double-click to select an item from the drop-down list.

**Note:** To exit from this pop-up without making a selection, click ESC.

In the **Relations** tab, all realization relationships begin with **Realize**.

## Changing the End Class

In the **Relations** tab on the **Class Specification** dialog box, you can change the **End Class** of the relation.

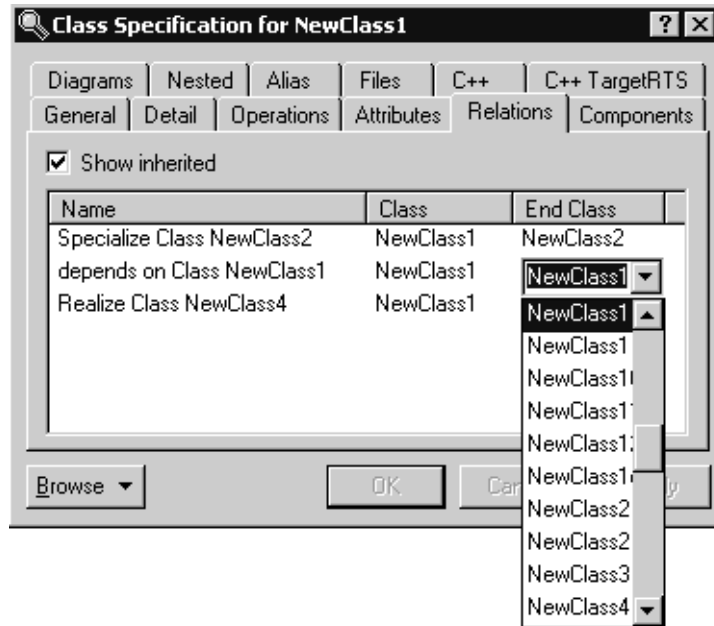
You can only modify the **End Class** for those relations in the **Relations** tab where the class name that appears in the **Class** column is the owner class for the relation. For example, the name that appears in the **Class** column (**NewClass1**) must be the same as the name that appears in the title for the dialog box (**Class Specification for NewClass1**).

You can change the **End Class** to any class-type object including: classes, capsules, protocols, class utilities, and actors.



## To change the End Class:

- 1 From the **Capsule**, **Class**, or **Protocol Specification** dialog box, single-click to select an item in the **End Class** column and wait a short time.



**Note:** The size of the drop-down list is the width of the **End Class** column. If you cannot see the full name of the items in the list, widen the **End Class** column in the dialog box.

When modifying the **End Class**, the pop-up list contains only those objects of the same type as the selected object:

- in a **Class Specification** dialog box for a class, the list only contains classes
- in a **Class Specification** dialog box for an actor, the list only contains actors
- in a **Capsule Specification** dialog box, the list contains capsules
- in a **Protocol Specification** dialog box, the list contains protocols

- 2 Select an item from the list.

**Note:** The diagram will be updated only if the relation already existed in the diagram.

## Adding and Hiding Classes, and Filtering Class Relationships

---

The commands on the Query Menu provide powerful facilities for controlling which model elements are represented by icons in the current diagram.

The options are as follows

- **Add Classes** - Adds classes to the diagram by name.
- **Expand Selected Classes** - Adds classes to the diagram based on their relationships to selected classes.
- **Hide Selected Classes** - Removes selected classes from the diagram and optionally removes their clients or suppliers from the diagram.
- **Filter Relationships** - Controls which kinds of relationships appear in the current diagram.

## Using State Machine Code Generation for Classes

---

State machine code generation on data classes allows you to take advantage of state machine code generation without using the RTS or capsules. You can model a state machine on a class to generate state machine functions and state variables. You can add the generated code into the source files or directly into the model as operations and attributes. If you modify the state machine, you can regenerate the state logic without losing detailed code annotations, such as transitions, entry and exit actions, and guards.

State machine code is entirely contained in the class and does not depend on external classes or the runtime system(s). From an external perspective, classes with state machines on them are no different than classes with operations. They support all of the runtime semantics of classes in a capsule-based model, and if configured as non-RTS classes or placed in a non-RTS component, they become usable in any C or C++ application.

### Configuring a Simple Model

To understand the associated benefits of using state machine code generation for classes, we will look at a simple "Hello World" model. For simplicity, Figure 27 shows a **Class** diagram that has both C and C++ content. Figure 28 shows the **Model View** tab in the browser for the Hello World model.

Figure 27 Class Diagram for the Hello World Example Model (C and C++)

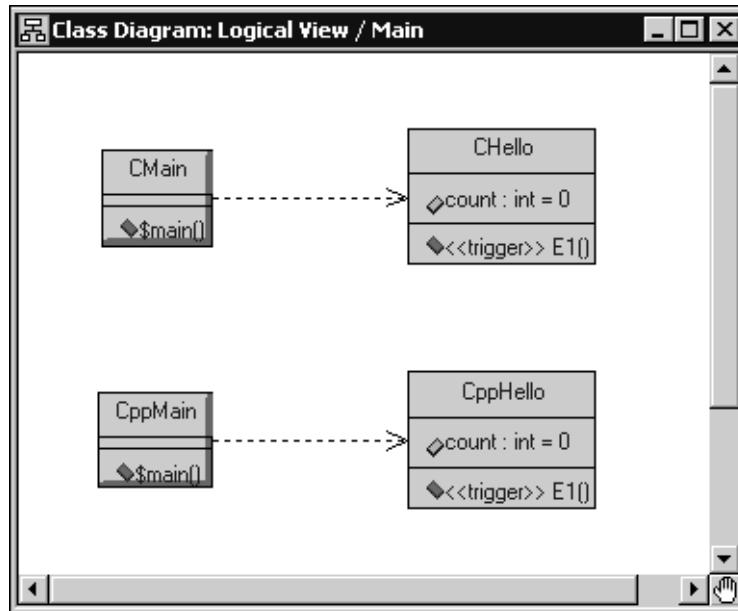
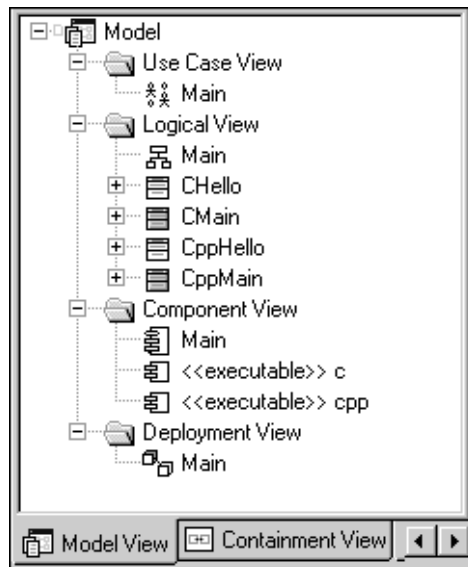


Figure 28 Model View Tab for the Hello World Model



There are two components in the **Component View**: one for C and one for C++. Each produces an executable that yields the same output when run.

When using state machine code generation for classes, the Hello World model is different from other models in the following areas:

- *Generating Component Libraries for Classes without RTS Dependencies* on page 180
- *Configuring the trigger Stereotype for an Operation* on page 183
- *Generating State Machine Code* on page 185
- *Support for Code Sync* on page 187
- *Using Constructors* on page 195
- *Using Return, Break, and Continue Statements* on page 198
- *Specifying History* on page 199
- *No Refinement* on page 200

## Generating Component Libraries for Classes without RTS Dependencies

The Hello World model contains classes in the **Logical View** (C and C++) and maps them to Component libraries. For illustrative purposes, we will use the **cpp** component as shown in Figure 29.

**Figure 29 Model View Tab - C++ Hello World Component**

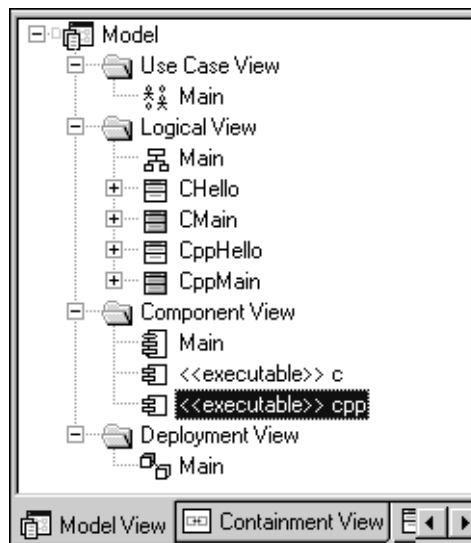
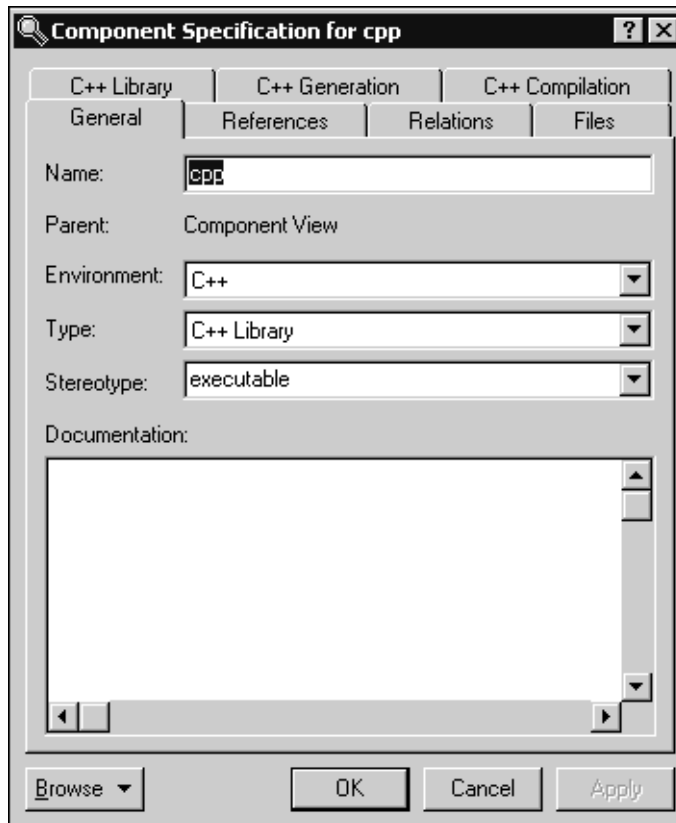


Figure 30 shows the **Component Specification** dialog box for the C++ component. The **Environment** box indicates that this component does not use the C++ and code generator.

Figure 30 Component Specification for C++ Dialog Box



The **Type** box indicates that a library will be built. The Hello World model has no Capsules or Protocols in the Component library and the generated library will contain no dependencies on the runtime system. This is beneficial because:

- it can be used by non-Rational Rose RealTime applications
- it is usable within Rational Rose RealTime applications with some limitations, as if you link with external code.

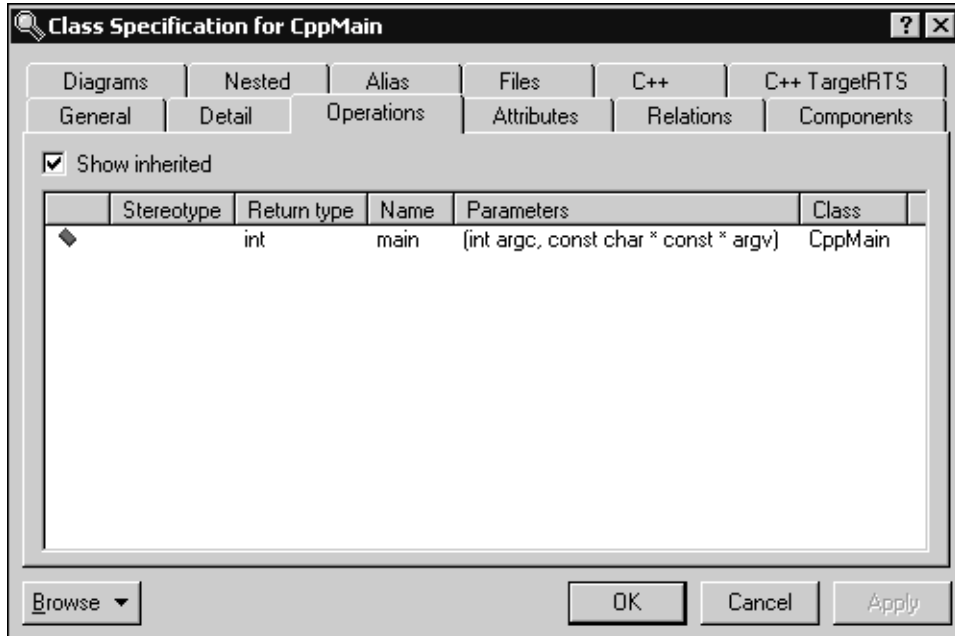
Figure 31 shows the **Operations** tab for the **CppMain** class in the Hello World model. For execution of the application to begin, code must be provided to call:

```
int main (int argc, const char * const * argv)
```

or

```
int main()
```

**Figure 31 CPPMain Class Specification Dialog Box**



**Note:** In C++, a class can have nested classes with state machines, and in turn, they may also have state machines.

## Creating State Machine Trigger Operations

The event editor for triggers on transitions in class state machines now contains a drop-down list.

**Figure 32 Event Drop-Down List**



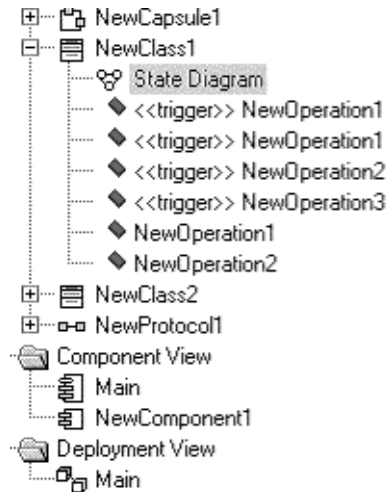
The drop-down list displays all available operations that are stereotyped <<trigger>> for the selected class. The item << **Create a new trigger** >> will create a new operation with a stereotype of trigger.

If there is only one operation stereotyped as trigger and no other operation has the same name, the operation appears in the **Model View** tab without a signature (see **NewOperation3** in Figure 33).

If there is more than one operation stereotyped as trigger with the same name, those operations appear in the **Model View** tab with signatures (see **NewOperation1** in Figure 33).

If there is more than one operation with the same name, but only one of the operations is a stereotyped as trigger, the trigger operation appears in the **Model View** tab without a signature (see **NewOperation2** in Figure 33)

**Figure 33 Trigger Operations in the Model View Tab in the Browser**



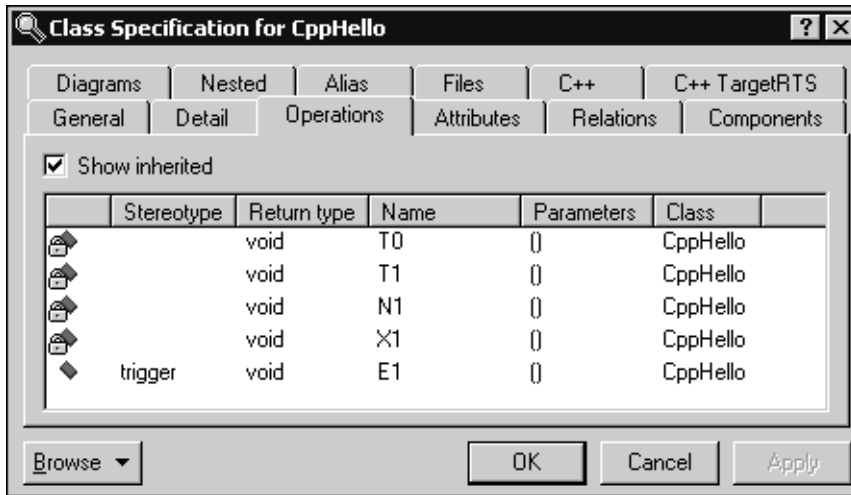
**Note:** You cannot create events that do not have names. You can specify a name from the drop-down list or you can type another name. However, code generation will fail if the name does not match the name of a trigger stereotyped operation.

## Configuring the *trigger* Stereotype for an Operation

Figure 34 shows the **Class Specification** dialog box for **CppHello** that contains an operation called **E1** whose **Stereotype** is set to **trigger**. This means that the **E1** operation for this class can trigger an event to occur. Figure 35 shows the **State Machine** diagram for the **CppHello** Class.

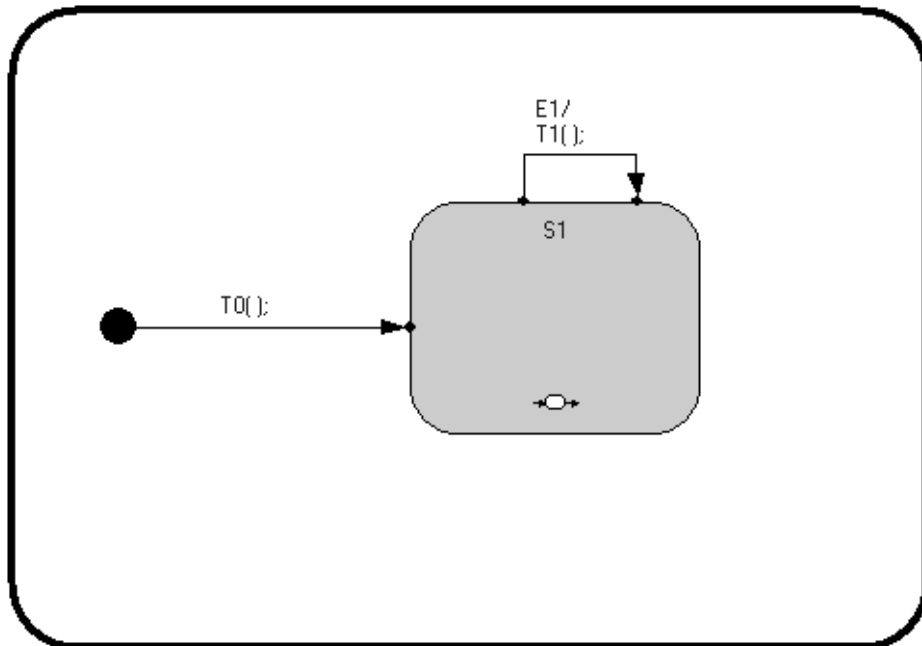
**Note:** Triggers only return void.

**Figure 34 Class Specification Dialog Box for the CppHello Class**



In the Hello World model, the **State Machine** diagram for the **CppHello** (C++) and **CHello** (C) classes are identical (see Figure 35).

**Figure 35 State Machine for the CppHello Class**





The composite state **S1** exists in the Top state of this state diagram. The Initial Transition from the initial state is called **T0**. There is an external self transition on the state **S1** that is triggered by the event **E1**. The action of this transition is called **T1**. The state **S1** has an entry action called **N1** and an exit action called **X1** (see Figure 34).

If you do not specify a trigger for an event on a state, if the event occurs, the default behavior is to do nothing.

## Triggers

A trigger is a stereotyped operation used to trigger a transition in a class state machine. Triggers must have the following properties:

- be public
- have a stereotype of `<<trigger>>`
- return void
- have no arguments
- have no detail code (code generation ignores arguments and detail code for a trigger operation on a data class)

As a result of the properties of a trigger, there is no concept of **rtdata/RTDATA** in the triggered transition. Also, there is no target observability for data class state digrams (for example, no passive class state monitors). Data class state machines are designed to work in a noRTS target environment and do not depend on the toolset or TargetRTS, which precludes the ability to do target observability.

Some functions can be executed in multiple triggers; the code is copied to multiple trigger functions. If there is a lot of code in a transition, and you want to copy this code, create an operation and then call this operation in the transition.

In C, you can have implementation visibility so that you can build a class where the implementation is separate from the state machine.

## Generating State Machine Code

You can generate state machine code to help debug your model, and to modify code outside the Rational Rose RealTime toolset. Generating state machine code updates the header (.h declaration files) and source code files with the latest code based on the information in your model.

To re-capture changes into the model, **Code Sync** must be enabled, and the changes must be made to designated Code Sync areas. For information on Code Sync for state machine code generation for classes, see *Support for Code Sync* on page 187.

To generate State Machine code, ensure that you select the **GenerateStateMachine** option on the **C** or **C++** tab on the **Class Specification** dialog box (see **Note** in Step 4). When selected, you can review the code generated from the state machines in your model.

**To specify state machine code generation in your model:**

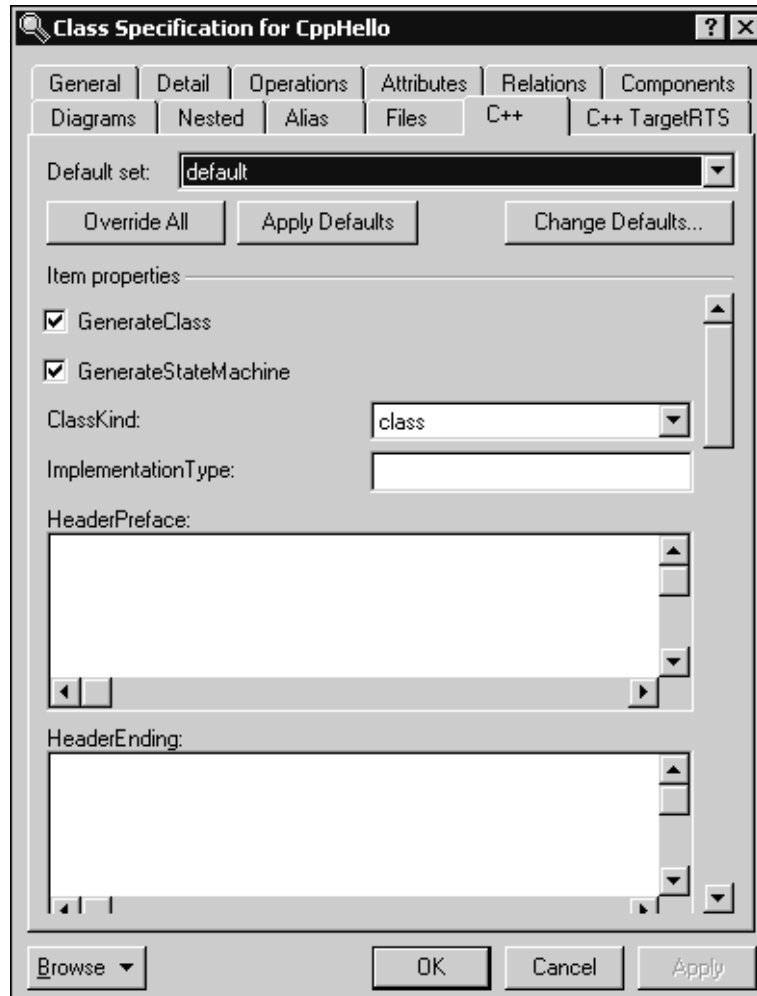
- 1 From the **Class** diagram, select a class.
- 2 Right-click and select **Open Specification**.
- 3 Click the language tab for your model: **C** or **C++**.

**Note:** When the label for a field is bold, this means that the field is overridden.

- 4 In the **Item properties** box, click **GenerateStateMachine**.

**Note:** By default, if you create a new model in Rational Rose RealTime 2002.05.21, the **GenerateStateMachine** option in the **Item properties** box is automatically selected and it appears at the top of the **C** and **C++** tabs (see Figure 36) on the **Class Specification** dialog box. If you open a model created in an earlier version of Rational Rose RealTime, the **GenerateStateMachine** option in the **Item properties** box is not selected.

**Figure 36 Class Specification - C++ Tab**



## Support for Code Sync

The purpose of Code Sync is to provide a facility to capture user modifications to generated code back into the model. This allows you to externally modify and debug the generated code outside of the toolset.

Modifying generated code helps to reduce the debug cycle on some Real Time Operating Systems (RTOS's), and allows you to make changes using a third-party Integrated Development Environment (IDE) or text editor. Using Code Sync, changes to the generated code can be reconciled and re-integrated back into the master copy of the model source files.

If you model classes and develop code in C or C++ without using capsules or the TargetRTS, you can perform debugging using traditional source debuggers. You will want to correct coding errors in the generated code during the debug cycle. Later, you can Code Sync these changes back into the model.

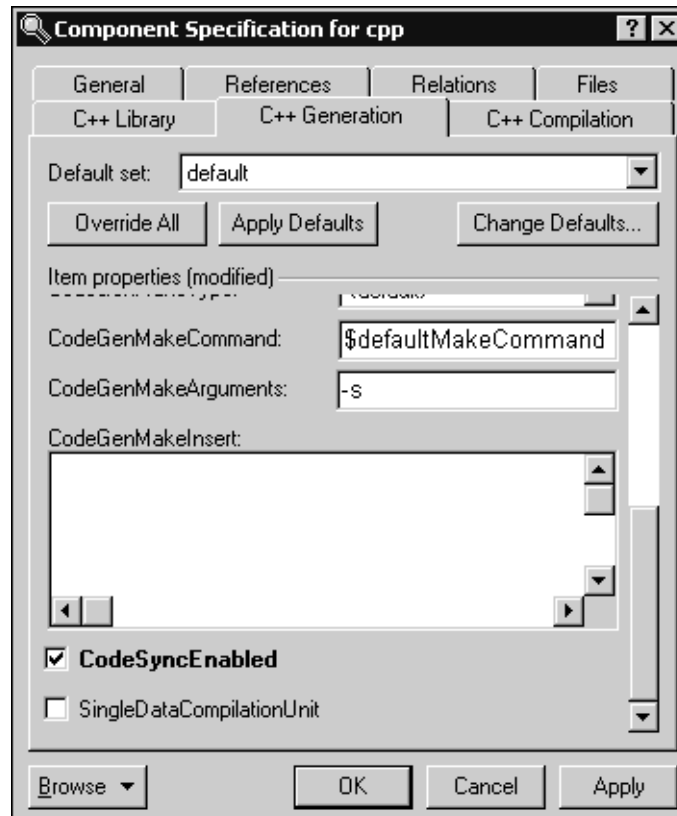
To generate the correct **Makefile** pattern for Code Sync, **Code Sync** must be enabled before the code is initially generated from the toolset.

By default, **Code Sync** is enabled on new components; however, you can disable **Code Sync**, if desired.

### To disable/enable Code Sync:

**Note:** For additional information on using the Hello World models (C++, C, and Java), see *Using the Startup Frameworks* on page 31.

- 1 In the Hello World model, open the **Component Specification** dialog box for the **cpp** component.
- 2 Click the **C++ Generation** tab.



### 3 Select **CodeSyncEnabled**.

**Note:** Components that are dependent on a component with Code Sync enabled do not necessarily need to have Code Sync enabled. To propagate the changes into the model, you must invoke Code Sync, and then determine the changes that you want to accept.

#### **To start Code Sync:**

- 1 From the **Model View** tab in the browser, select the **cpp** component from the **Component View**.
- 2 Right-click and select **Code Sync**. Alternatively, if the component is currently set as the active component, click **Build > Code Sync**.

For additional information on using Code Sync, see *Code Sync* in the Rational Rose RealTime online Help.

### **Considerations**

When using Code Sync, consider the following:

- Code Sync cannot be used to create, delete, or rename model elements, or to otherwise make structural changes to the model. You must make these kinds of modifications using the Rational Rose RealTime toolset.
- If you modify any generated code externally (that is, outside of the Rational Rose RealTime toolset), do not use the toolset to run the externally built executable until all Code Sync changes have been reconciled.
- If you modify the generated code manually, **Clearmake** cannot provide complete traceability back to model files, and it cannot provide wink-in. This means that generated code that has been manually modified is no longer considered a derived object, but rather a view-private file.

- Code Sync only recognizes code delimited by the Code Sync identification tags. You should only modify code that is delimited by the Code Sync identification tags.

- Designated areas for Code Sync are identified in the generated **C++ code** with the following tags:

```
// {{{USR capsuleClass 'NewCapsule1' tool 'OT::Cpp' property 'HeaderPreface'
<insert or modify code here>
// }}}USR capsuleClass 'NewCapsule1' tool 'OT::Cpp' property 'HeaderPreface'
```

- Designated areas for Code Sync are identified in the generated **C code** with the following tags:

```
/* {{{USR capsuleClass 'NewCapsule1' tool 'OT::C' property 'HeaderPreface' */
<insert or modify code here>
/* }}}USR capsuleClass 'NewCapsule1' tool 'OT::C' property 'HeaderPreface' */
```

## Hello World Implementation and Header Files

After compilation of the Hello World model, the `src` directory contains the source files for the model. You can modify the generated code in the source files from outside the toolset within an IDE or text editor of your choice, and update your model with your changes (see *Support for Code Sync* on page 187 and *Considerations* on page 189).

The following files show some of the generated code for the C++ implementation of the Hello World model:

- *CPPHello Class Header File (C++ with Code Sync Disabled)* on page 190
- *CPPHello.cpp (C++)* on page 192

### CPPHello Class Header File (C++ with Code Sync Disabled)

```
// {{{RME classifier 'Logical View::CppHello'

#ifdef CppHello_H
#define CppHello_H

#ifdef PRAGMA
#pragma interface "CppHello.h"
#endif

#include <RTSystem/cpp.h>

class CppHello
{
```

```

public:
    // {{{RME tool 'OT::Cpp' property 'PublicDeclarations'
    // }}}RME

protected:
    // {{{RME tool 'OT::Cpp' property 'ProtectedDeclarations'
    // }}}RME

private:
    // {{{RME tool 'OT::Cpp' property 'PrivateDeclarations'
    // }}}RME
    struct RTState_CppHello
    {
        inline RTState_CppHello( void );
        inline ~RTState_CppHello( void );
        unsigned char state;
    };
    RTState_CppHello rtg_state_CppHello;

public:
    // {{{RME classAttribute 'count'
    int count;
    // }}}RME
    // {{{RME tool 'OT::Cpp' property 'GenerateDefaultConstructor'
    CppHello( void );
    // }}}RME

private:
    // {{{RME operation 'T0()'
    void T0( void );
    // }}}RME
    // {{{RME operation 'T1()'
    void T1( void );
    // }}}RME
    // {{{RME operation 'N1()'
    void N1( void );
    // }}}RME

```

```

// {{{RME operation 'X1()'
void X1( void );
// }}}RME
// {{{RME enter ':TOP:S1'
void rtg_enter2( void );
// }}}RME
// {{{RME exit ':TOP:S1'
void rtg_exit2( void );
// }}}RME
void rtg_init1( void );

public:
    void E1( void );
};

inline CppHello::RTState_CppHello::RTState_CppHello( void )
    : state( 1U )
{
}

inline CppHello::RTState_CppHello::~~RTState_CppHello( void )
{
}

#endif /* CppHello_H */

// }}}RME

```

### **CPPHello.cpp (C++)**

```

#if defined( PRAGMA ) && ! defined( PRAGMA_IMPLEMENTED )
#pragma implementation "CppHello.h"
#endif

#include <RTSystem/cpp.h>
#include <CppHello.h>

```



```

// {{{RME tool 'OT::Cpp' property 'ImplementationPreface'
#include <iostream>
// }}}RME

// {{{RME tool 'OT::Cpp' property 'GenerateDefaultConstructor'
CppClassHello::CppClassHello( void )
    : count( 0 )
{
    rtg_init1();
}
// }}}RME

// {{{RME operation 'T0()'
void CppHello::T0( void )
{
    std::cout << "T0: Hello, world!\n";
}
// }}}RME

// {{{RME operation 'T1()'
void CppHello::T1( void )
{
    std::cout << "T1: count=" << ++count << "\n";
}
// }}}RME

// {{{RME operation 'N1()'
void CppHello::N1( void )
{
    std::cout << "N1: Entering S1\n";
}
// }}}RME

```

```

// {{{RME operation 'X1()'
void CppHello::X1( void )
{
    std::cout << "X1: Exiting S1\n";
}
// }}}RME

// {{{RME enter ':TOP:S1'
void CppHello::rtg_enter2( void )
{
    rtg_state_CppHello.state = 2U;
    {
        N1();
    }
}
// }}}RME

// {{{RME exit ':TOP:S1'
void CppHello::rtg_exit2( void )
{
    X1();
}
// }}}RME

void CppHello::rtg_init1( void )
{
    {
        // {{{RME transition ':TOP:Initial:Initial'
        T0();
        // }}}RME
    }
    rtg_enter2();
}

```

```

void CppHello::E1( void )
{
    unsigned char rtg_state = rtg_state_CppHello.state;
    for(;;)
    {
        switch( rtg_state )
        {
            case 2U:
                // {{{RME state ':TOP:S1'
                rtg_exit2();
                rtg_state_CppHello.state = 1U;
                {
                    // {{{RME transition ':TOP:S1:Junction2:T1'
                    T1();
                    // }}}RME
                }
                rtg_enter2();
                return;
                // }}}RME
            default:
                return;
        }
    }
}

// }}}RME

```

## Using Constructors

A state machine must be initialized before it is used, which may cause one or more initial transitions to fire. Since these transitions may cause operations on the class for its state machine, initialization must occur after the class is constructed.

**Note:** You can specify when that state-machine is initialized. A state machine may also be re-initialized.

## C Language

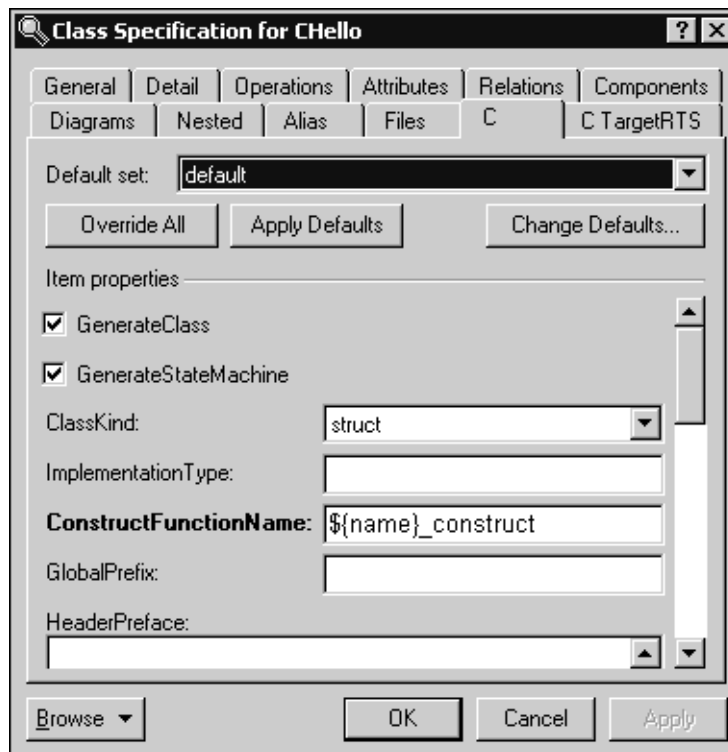
For C, you must set the **ConstructFunctionName** box in the **C** tab on the **Class Specification** dialog box (see Figure 37) to configure the name of the constructor function for the generated class. The default name for the construct function is:

**\$(name)\_construct**

where **\$(name)** is the name of the class. For example, if your class is called **CHello**, then the generated function would be **CHello\_construct**.

**Note:** If the **ConstructFunctionName** box is blank, a constructor function is not generated.

**Figure 37** Class Specification Dialog Box- C Tab - ConstructFunctionName Box



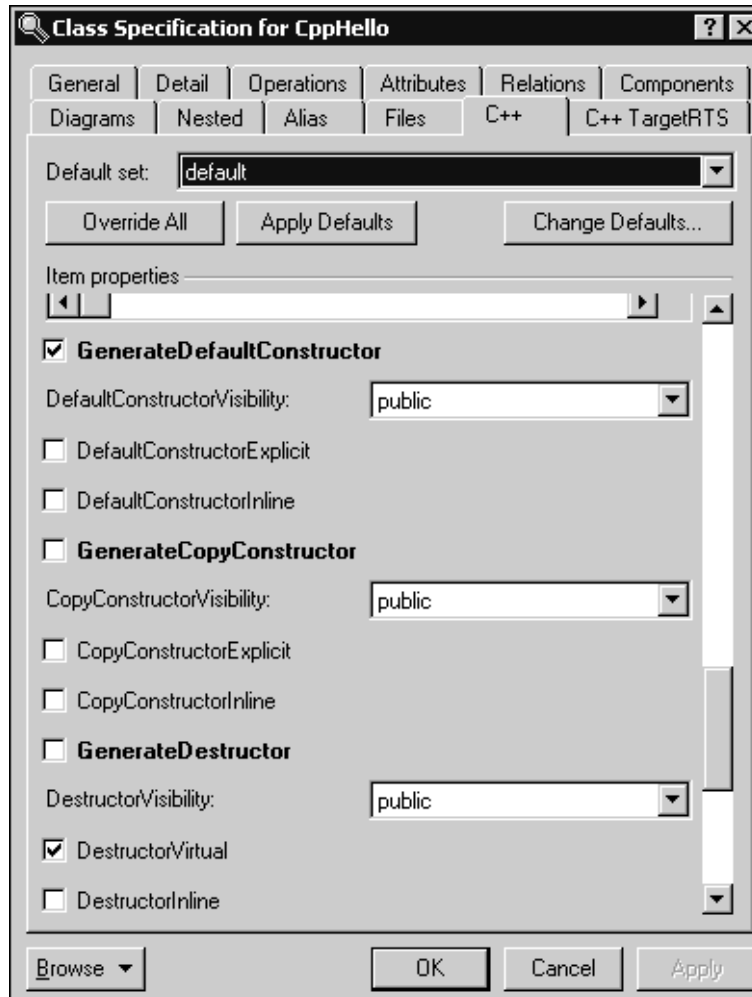
You can create a trigger of your own to initialize the state machine.

**Note:** The **rtg\_init1** reference in the generated code for the Hello World model causes the automatic initialization of state machine in the constructor.

## C++ Language

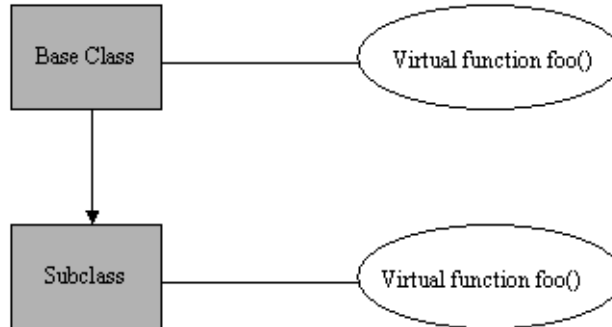
In C++, the code in the default constructor initializes the state machine. The `init` occurs in `GenerateDefaultConstructor` while `GenerateCopyConstructor` copies the state machine. The `rtg_init1` reference in the code for the Hello World model (see *Hello World Implementation and Header Files* on page 190) causes the automatic initialization of state machine in the constructor.

Figure 38 Class Specification for CppHello Dialog Box



## Using Virtual Functions

The state machine is initialized by constructors. Consequently, you must use caution when using virtual functions. For example, if you have a Base Class with the virtual function `foo()`, and the Subclass of the Base Class also has a virtual function named `foo()` (which is different from `foo()` in Base Class), the virtual function in the Base Class will always be used.



## Using Return, Break, and Continue Statements

Use caution when attempting to make use of things that can affect control flow for your code. Code for transitions, choice points, and guards do not get encapsulated in functions; they are used directly in the body of the trigger function.

- **Return** statement - **Return** cannot be used; otherwise you interrupt the chain of state machine code part way through. The code after the **Return** statement never gets executed. Because you cannot use **Return**, you cannot return a value.
- **Break** statement - If you use a **Break** statement in your code, the result is as if you did not take that transition. It is acceptable to use a **Break** statement in the code for a transition (such as inside a **for** loop). However, do not use **Break** outside a loop construct because it will break the generated state machine logic and result in unspecified behavior.
- **Continue** statement - If you use a **Continue** statement in your code, the result is as if you did not take that transition. It is acceptable to use a **Continue** statement in the code for a transition (such as a **for** loop). However, do not use **Continue** outside a loop construct because it will break the generated state machine logic and result in unspecified behavior. If you must use **Continue**, add an operation to the class, then call this operation in the choice point or guard.

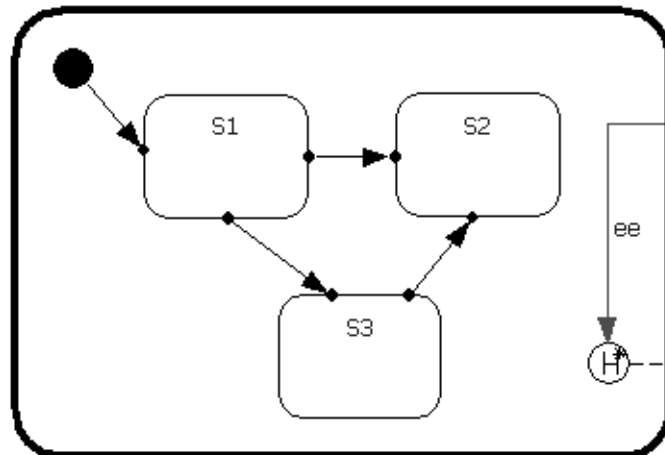
## Specifying History

History is useful when dealing with situations where an event takes control away from the current state and initiates a separate behavior sequence for handling the new event. The new sequence can involve new states and transitions. However, once completed, we often want to resume from the point before the interruption occurred.

### Example

Figure 39 shows an example of a state diagram containing transition **ee**; a self-transition that has a trigger for an event that none of the substates can handle. When that event occurs, the self transition will fire, and then go to **H\*** (deep history) meaning that it will revert to the last active substate. The result is to perform event handling without changing the state of the system.

**Figure 39 A State Machine Diagram Showing History**



### Entry and Exit Functions

**Note:** Ensure that you are aware of how **Entry** and **Exit** actions are called when the **ee** transition is taken. For example, if the current active state is **S2**, when **ee** is triggered, the **Exit** action for **S2** will be taken. Then, the actions for **ee** execute, and finally the **Entry** action for **S2** executes.

## No Refinement

Unlike capsules, the state machines of your base class cannot be refined in your subclass. Inherited state machines have no refinement. However, you can achieve this refinement by using virtual functions, and then in the subclass, you can override the virtual function to do something different.

## Overriding Virtual Operations

**To override an operation defined in a parent class from within a subclass:**

- 1 Ensure that the operation on the parent has the **Polymorphic** option checked.
- 2 Create a new operation on the subclass with the same signature as the operation in the parent.

## Generation of Parameterized and Instantiated Classes

---

A parameterized class is a template for creating any number of instantiated classes that follow its format. A parameterized class declares formal parameters. You can use other classes, types, and constant expressions, as parameters. You cannot use the parameterized class itself as a parameter. You must instantiate a parameterized class before you can create its objects.

In its simplest form, you use parameterized classes to build container classes. You can also use parameterized classes to capture design decisions about the protocol of a class. Use the arguments of the parameterized class to import classes or values that export a specific operation. In this form, a parameterized class denotes a family of classes whose structure and behavior are defined independently of its formal class parameters.

**Note:** Support for parameterized classes and instantiated classes is only available for the C++ language.

**To Create a Parameterized Class or Instantiated Class:**

- 1 Select the **Class** tool from the toolbox and click on the Class Diagram to create a class.
- 2 Right-click on the Class object and click **Open Specification**.

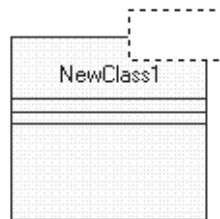


- 3 Click the **General** tab.
- 4 In the **Type** box, specify the type of class you want to create.
- 5 Specify additional options and information you require on the other tabs for this class.
- 6 Click **OK**.

## Parameterized Classes

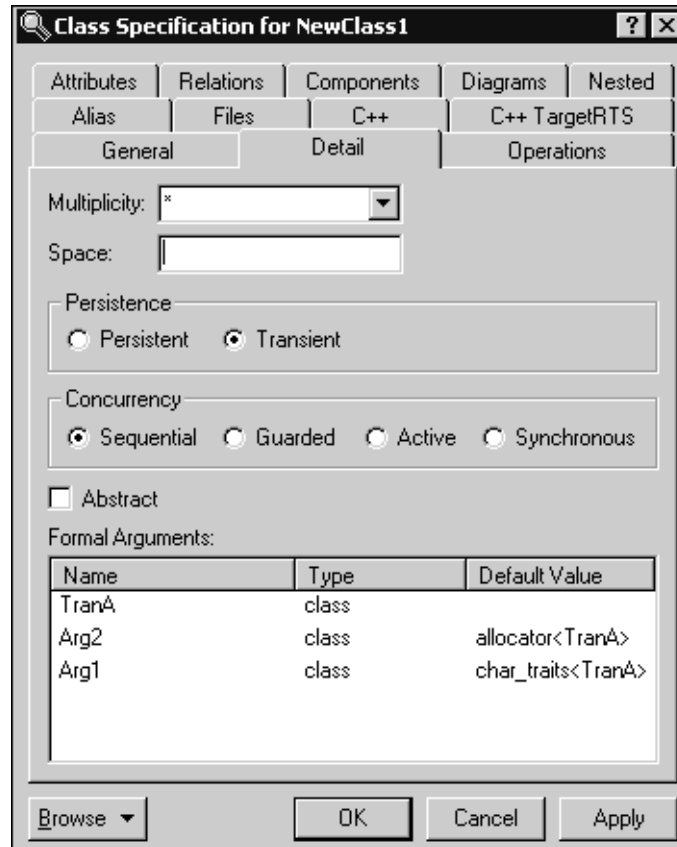
A parameterized class describes a group of classes.

**Figure 40 Parameterized Class**



In Rational Rose RealTime, a parameterized class maps to a template whose parameters are those listed as formal arguments on the **Detail** tab of the **Class Specification** dialog. For example, Figure 41 shows the template parameters associated with a parameterized class

Figure 41 Template Parameters



This corresponds to the preface of the `basic_string` template from the standard template library.

```
template < class Ch, class Tr = char_traits< Ch >, class A = allocator< Ch > >
```

Depending on the information on the `C++` tab, the generator will support templates based on classes where the `ClassKind` property is `class`, `struct` or `union`.

**Note:** Enumerations and `typedefs` are not supported.

Parametrized classes may define attributes, operations and nested classes. They may participate in dependency and generalization relationships. They may not be the target of navigable associations, nor can they be the association class of any association. They can only be the target of instantiation relationships.

The generated header file will contain the declaration and any inline features. The generated implementation file will contain only the **ImplementationPreface** and **ImplementationEnding**. Any other required code is generated in the implementation file of instantiated classes.

## Relationships

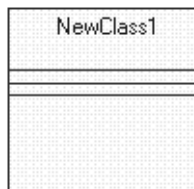
You can draw the following relationships:

Draw:	From a Parameterized Class To:
Generalize Relationship	Class, another parameterized class, instantiated class, interface
Association Relationship	Class, another parameterized class, instantiated class, class utility, parameterized class utility, instantiated class utility, interface
Dependency Relationship	Class, another parameterized class, instantiated class, class utility, parameterized class utility, instantiated class utility, interface

## Instantiated Classes

An instantiated class is a class formed from a parameterized class by supplying actual values for parameters. You use an instantiated class to select a member of the set of classes described by a parameterized class.

**Figure 42 Instantiated Class**



In Rational Rose RealTime, an instantiated class maps to a **typedef** whose definition is the parameterized class referenced by the instantiation relationship using the names of the actual arguments given by the instantiated class.

You create an instantiated class by supplying actual values for the formal parameters of the parameterized class. This instantiation process forms a concrete class in the family of the parameterized class.

**Note:** You must place the instantiated class at the client end of an instantiate relationship that points to the corresponding parameterized class.

An instantiated class whose actual parameters differ from other concrete classes in the parameterized class' family forms a new class in the family.

## Relationships

You can draw the following relationships for instantiated classes:

Draw	From an Instantiated Class To
Association Relationship	Class, parameterized class, another instantiated class, class utility, parameterized class utility, instantiated class utility, interface
Dependency Relationship	Class, parameterized class, another instantiated class, class utility, parameterized class utility, instantiated class utility, interface

Figure 43 shows a string that is an instantiated class participating in an instantiation relationship with the parameterized class **basic\_string**.

**Figure 43 Instantiated Class Participating in an Instantiation Relationship**



When generated, the header file contains the following declaration:

```
typedef basic_string < char > string;
```

Instantiated classes cannot define attributes, operations or nested classes. State machines may not be generated for instantiated classes. They may participate in dependency relationships. They may only be the target of navigable associations and generalization relationships.

**Note:** There must be exactly one instantiation relationship whose target is a parameterized class.

## Limitations

The limitations regarding the support of parameterized and instantiated classes are:

### No Automatic Type Descriptors

The code generator cannot automatically create type descriptors for instantiated classes. However, descriptors will be generated if you specify the five function bodies required to produce a descriptor.

**Note:** Classes nested within parameterized classes are subject to this limitation even if they themselves are not parameterized.

### Toolset Dependencies

You will need to address the following issues:

- Import Generalization Relationships

Rational Rose has a C++ property named **InstanceArgument** that you must import into a property of the same name and associated with the **OT::C++** tool in Rational Rose RealTime.

- Relax Rules For Actual Arguments

Rational Rose uses the contents of **Name** column of the actual arguments of an instantiated class. Because these arguments can be any legal C++ type or value expression, Rational Rose RealTime cannot modify the names when importing a Rational Rose model, nor can it place any restrictions on the names entered.

**Note:** Parameter names are validated whenever the class **Type** changes, or when parameters are copied from an instantiated class to a more restrictive context. The illegal characters are replaced by underscores.

- Allow Deletion of Actual Arguments

If you copy an actual argument from one instantiated class to another, you cannot delete it from the source class.



## Contents

This chapter is organized as follows:

- *Creating Capsule Structure* on page 207
- *Using the Structure Editor* on page 208
- *Structure Diagram Toolbox* on page 211
- *Creating a Port* on page 212
- *Port Specification* on page 213
- *Adding a Capsule Role* on page 219
- *Capsule Role Specification* on page 219
- *Connecting Ports on Capsule Roles Together* on page 221
- *Connector Specification* on page 221
- *Creating a Collaboration Diagram* on page 222
- *Using the Collaboration Diagram Editor* on page 222

## Creating Capsule Structure

---

Capsules are one of the primary modeling elements in Rose RealTime. Complete executable code implementations are generated by the toolset for capsules.

Capsule structure is defined through the Structure Diagram Editor.

There are three kinds of structural element that may be added to a capsule structure diagram:

- Capsule roles
- Ports
- Connectors

None of these elements are required. A capsule does not require any structural elements. To do anything useful, a capsule usually requires at least a port so that it can communicate with other capsules.

Creating a complete capsule structure definition may consist of any of the following basic steps:

- Adding a Capsule Role
- Creating a Port
- Connecting Ports on Capsule Roles Together

## Using the Structure Editor

---

The structure editor is used to define the structure of a capsule class. That is, how instances of that capsule class are composed of other capsule class instances and protocol class instances (ports). The structure editor consists of three parts: the structure diagram area, the structure browser, and the Structure Diagram Toolbox.

Structure elements, such as ports and capsule roles, can be created by dragging a protocol or capsule on to the structure diagram from any browser (usually from the model browser). Structure elements can also be added using the toolbox. The structure browser can be used to navigate to, and open editors and specification dialogs on contained structure elements.

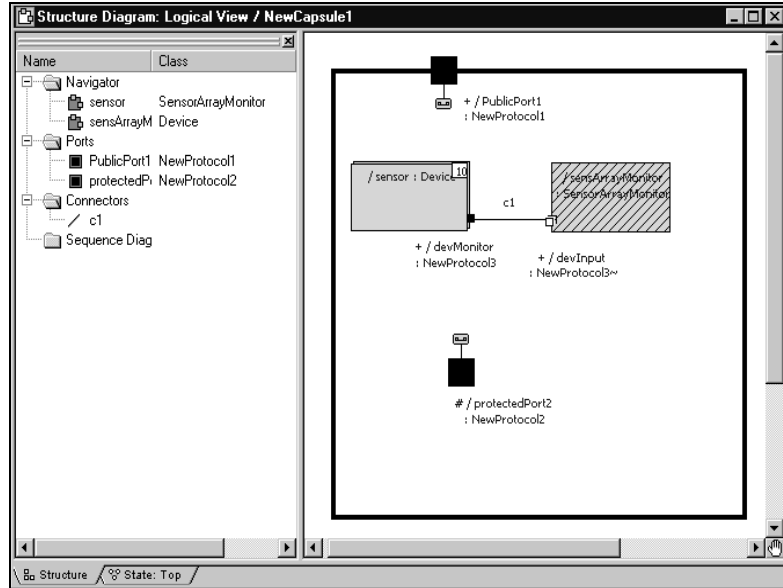
Several standard diagram manipulations can be performed, including resizing, scaling, and filtering.

You can use the popup menu to navigate between structure and state diagrams.

Figure 44 shows a sample structure editor. The window title bar shows the full name of the class. The left side of the window contains the structure browser. The right side contains the structure diagram area. The structure toolbox is not shown. It is usually anchored outside of the diagram window.



**Figure 44 The Structure Editor**



## UML Options

You can use the popup menu to toggle the following UML options:

### Base UML notation

Converts the structure diagram so that it uses base UML notation.

### Show Classifier Name on Roles

Lets you turn off the classifier name portion of a role label.

### Show Protocol Name on Ports

Lets you turn off the classifier name portion on ports.

## Structure Diagram Browser Context Menu Options

**Open Specification** - Displays the Specification dialog for the selected model component.

**Open Capsule Specification** - Displays the Specification dialog for the selected capsule.

**Open Structure Diagram** - Displays the structure diagram for the selected element.

**Open State Diagram** - Dpens the State diagram for the selected object.

**Create Sequence Diagram** - Creates a Sequence diagram under the Structure diagram and populates the sequence diagram with interaction instances containing selected capsule roles. Roles excluded from sequence diagram are not added to the sequence diagram. If only excluded roles are selected, a sequence diagram is not created.

**New** - Allows you to insert references to files or URLs.

**Add New State** - Adds a new state to the browser.

**Delete** - Removes the currently selected object from the browser.

**Rename** - Allows you to change the name of the selected object.

**Promote** - Moves the selected element up in the hierarchy. If there is any name conflict between this element and another element in the previous state, or in any of its substates, the promote command fails.

**Demote** - Moves the selected element down in the class hierarchy. The element is removed from the current State diagram, and is moved down to the next level.

**Filter Folders** - Allows you to specify which folder appear in the State Diagram browser.

**Sort** - Allows you to arrange objects in the browser either in alphabetical order or the internal order of the objects.

**Find In** - Allows you to search for all occurrences of a specified search string.

**Replace In** - Allows you to perform a search to find all occurrences of a search string and replace it with other text.

**Open Another Browser** - Enables you to open another browser in the same window. You can also specify which folders appear in this browser.

**Close Browser** - Closes the browser window for the State Diagram dialog.

**Refresh** - Redraws the current diagram.

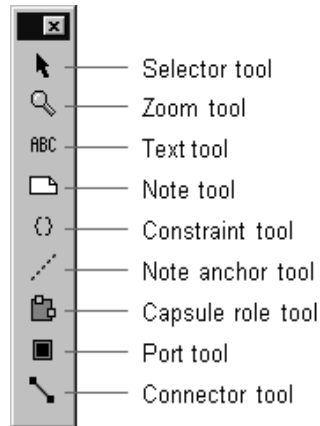
**Rational RequisitePro Trace Tool** - Allows you to maintain and establish traceability between your design requirements and your Rational Rose RealTime elements.

# Structure Diagram Toolbox

---

The structure toolbox contains tools for adding elements to the structure diagram.

**Figure 45 Structure toolbox**



## **Selector Tool**

Selects objects for moving, resizing, and so forth.

## **Zoom Tool**

Zooms in on or out on diagrams.

## **Text Tool**

Adds text anywhere in the structure diagram.

## **Note Tool**

Annotates the diagram with textual notes. This is useful for marking up the diagram with explanations, review comments, and so forth. You can drag and drop a diagram or external document from the browser onto a note. Notice that the name of the diagram or external document is underlined. If you double-click on the note, the diagram or external document is opened. You can undo and redo this command.

## **Constraint Tool**

Adds UML constraints to the diagram. A constraint can be anchored to a view element by using the anchor tool. Currently, constraints do not have any semantic meaning to the tool. There are RRTEI APIs to add or remove, and enumerate constraints in a diagram.

### **Note Anchor Tool**

Anchors a note to a particular element on the diagram.

### **Capsule role Tool**

Adds a capsule role to a capsule collaboration diagram. A pick-list is displayed allowing you to select the class of the capsule role from the list of capsule classes. The first entry in the pick-list menu is **Create a New Capsule** which creates a new capsule class with a default name such as 'NewCapsule1'.

### **Port Tool**

Adds new ports to the capsule class structure. Ports can be placed either in the internal structure (inside the black interface boundary, which makes them protected, or on the structure interface, which makes them public.

A popup menu appears on the capsule role allowing you to select the protocol class of the port from the list of available protocol classes (that is, all protocol classes in the model). The first entry in the popup menu is **Create a New Protocol** which creates a new protocol class called 'NewProtocol1'.

Protected ports are automatically created as end ports. Public ports are created as end ports by default.

### **Connector Tool**

The connector tool is used to wire ports together. Usually connectors bind ports on different contained capsule roles together within the container capsule class. Connectors can also bind internal end ports of the container class to other ports. Interface end ports can only be bound within the context of a container class.

Only compatible ports can be connected together. Compatible ports are usually two ports of the same protocol class, one of which is conjugated.

## **Creating a Port**

---

There are four different ways to add a port to a capsule:

- Drag and drop a protocol class name from the model browser onto the capsule structure diagram.
- Draw an aggregation between a capsule class and a protocol class on a class diagram.

- Use the **Port** tool on the capsule structure diagram toolbox. Select the tool and click on the capsule boundary to add a public port. Click inside the boundary to add a protected port.
- From the **Navigator** area of a capsule diagram editor (either the collaboration diagram editor or the state editor), right-click on the Ports folder and select **Add New Port** from the popup menu.

If you use the port tool, a pick-list appears on the resulting port allowing you to select the protocol class to be used from a list of available protocol classes. The first entry in the pick-list is **Create a New Protocol** which creates a new protocol class called 'NewProtocol1'. If you choose to create a new protocol class it will be added to the same package as the container capsule class.

## Creating a Non-Wired Port Using a System Protocol

**To create a non-wired port to access one of the system services (a Frame, Timing, Log or Exception port):**

- 1 Right-click on the **Ports** folder on the Navigator area of the capsule's state diagram editor.
- 2 Select **Add New Port** from the popup menu.
- 3 Select one of the system services from the list that appears.

## Port Specification

---

The **Port Specification** provides control over information about ports on capsules.

The **Port Specification** contains two tabs: the **General Tab** and the **Files Tab**.

### General Tab

#### Name

The port is referenced by a name. The default name provided when the port is first created is based on the protocol name. In addition to appearing on the structure diagrams, the port name is used by the behavior of the capsule containing the port. To send and receive messages, the capsule's behavior references the port name in detailed code, and in transition trigger events.

## Stereotype

A stereotype represents the subclassification of an element. It represents a class within the UML metamodel itself, that is, a type of modeling element. Some stereotypes are already predefined, but you can also define your own to add new kinds of modeling types.

Stereotypes can be shown in the browser and on diagrams. The name of the stereotype may appear in angle brackets <<>>, depending on the settings found in either the Diagram or Browser tabs of the Options dialog located under the **Tools** menu. Refer to the Stereotype chapter for more information on stereotypes.

To show stereotypes on the diagrams, click **Options** from the shortcut menu and click **Stereotype Name** or **Stereotype Icon**. **Stereotype Name** displays the name in angle brackets (that is, <<stereotype>>). **Stereotype Icon** displays the graphical representation.

## Protocol

Specifies the protocol class to be used for the port. The protocol class (together with the Conjugation check-box) determines the set of messages that can be sent through this port (the **out** set), and the set of messages that can be received (the **in** set). The field has a pull-down menu to select from the available protocol classes in the model. The pull-down list always includes the service protocols for communicating with the target services library.

The **Open** button opens the Protocol Specification for the selected protocol class.

**Note:** If the **Protocol** box contains a value, the corresponding label becomes a hot link to the Specification dialog for that protocol.

## Cardinality

Specifies the number of instances of the port that will appear at run-time. The port is implemented as a member variable of the containing capsule. The variable may be an array of ports, connected to multiple ports on the other end of the connector. In this case, the port name points to an array of port instances. The Cardinality specifies the size of the array. Not all port instances in the array are necessarily connected. Individual port instances are referenced by indexing into the array. See the message send syntax in the *Programmer's Guide* for details. The Cardinality can be specified with an integer value or as the name of a constant defined within the model.

## Conjugated

A conjugated port is one in which the standard protocol class definition of in and out signals is reversed. That is, on a conjugated port the protocol class out signals become the port's in signals, and the protocol class in signals become the port's out signals. This enables two ports of the same protocol class to be connected together without having to define a separate reverse protocol. A connection can be made between two ports of the same protocol by conjugating one of the ports. This is the most common way of establishing communicating between two ports.

## End Port

Indicates that the port is an End Port, capable of sending and receiving messages. End Ports provide a connection between the behavior of the capsule containing the end port and the outside world. If this check-box is not checked, then the port is a *relay port*. *Relay ports cannot be protected, they must be public.*

To send messages, a capsule must have end ports. The end port's protocol defines the set of messages that can be sent.

To receive messages and process them within the capsule's behavior, the capsule must have end ports. The end port's protocol defines the set of messages that can be received.

Messages received on relay ports are not visible to the behavior of the capsule containing the port. Relay ports are intended to be connected to capsule roles contained within the capsule. Relay ports take messages from outside of the capsule and relay them through the capsule's encapsulation boundary to other capsules contained inside.

## Wired

Indicates that the port is a wired port. Wired ports are connected to other wired ports using connectors (via the Connector tool in the capsule structure diagram).

Non-wired ports are connected to other non-wired ports by name.

The connection of wired ports is done automatically based on the system structure.

Wired ports on fixed capsules are connected at initialization time. Wired ports on optional and plug-in capsules are connected dynamically when the capsule is instantiated or plugged-in.

The connection of non-wired ports may be done in two ways:

- 1 Automatically by name at the time the capsule is initialized (Automatic Registration).  
In this case, when the capsule is initialized, a non-wired protected port is connected to any non-wired public port of the same name. See Rules for Non-Wired Port Connection.
- 2 Dynamically by a name specified by the capsule's behavior (Application Registration).
- 3 Automatically by name at the time the capsule is initialized (Automatic locked Registration). Application calls to the following will fail and set the register to `RTController::badOperation`:

```
deregister, deregisterSAP, deregisterSPP, registerAs,  
registerSAP, registerSPP
```

In this case, when the capsule is initialized, a non-wired protected port is connected to any non-wired public port of the same name. See Rules for Non-Wired Port Connection.

In this case, the port is not connected at initialization time, it is connected when the capsule's behavior invokes a service function to register the port by a specified name. The same port may in fact be registered under different names at different points in the model execution.

This is determined by the registration method selected.

## Protected

Determines whether the port is visible outside of the capsule boundary. If the port is not protected, it is public. Public ports are part of the capsule interface and are visible to other capsules. By default, ports are protected.

## Notification

When selected, the port will receive **rtBound** and **rtUnbound** messages from the services library when ports get connected and unconnected.

**Note:** **rtBound** is sent at system priority and **rtUnbound** is sent at background priority.



## Publish

Determines whether the port is visible (SPP) or invisible (SAP). A Service Provisioning Point (SPP) describes an unwired port that is participating in a connection as the publisher. A Service Access Point (SAP) describes an unwired port that is participating in a connection as the subscriber.

## Registration

Specifies the type of registration for the port. This option is only enabled for non-wired ports. Non-wired ports are registered by name with a name service that performs the connection. Connections are made between protected non-wired ports (service clients) and a single public non-wired port (the service provider). There are three registration modes:

- **Automatic** - Automatic registration by name at the time the capsule is initialized. In this case, when the capsule is initialized, a non-wired protected port is connected to any non-wired public port of the same name.
- **Application** - Dynamic registration by a name specified by the capsule's behavior.
- **Automatic (locked)** - Automatic registration by name at the time the capsule is initialized. Application calls to the following will fail and set the register to `RTController::badOperation`:
  - `deregister`
  - `deregisterSAP`
  - `deregisterSPP`
  - `registerAs`
  - `registerSAP`
  - `registerSPP`

In this case, when the capsule is initialized, a non-wired protected port is connected to any non-wired public port of the same name. The port is not connected at initialization time. It is connected when the capsule's behavior invokes a service function to register the port by a specified name. The same port may be registered under different names at different points in the model execution. This is determined by the registration method selected.

## Files Tab

A list of referenced files. You can insert and delete references to files or URLs.

You can also link external files to the Specification for documentation purposes.

## Port Role Specification Dialog

---

### **Name**

Specifies the name for the port role.

**Note:** If the **Name** box contains a name, the corresponding label becomes a hot link to the Specification dialog for that port.

### **Protocol Class**

Specifies the protocol class for the port role.

**Note:** If the **Protocol Class** box contains a protocol name, the corresponding label becomes a hot link to the Specification dialog for that protocol.

### **Conjugated**

A conjugated port is one in which the standard protocol class definition of in and out signals is reversed. That is, on a conjugated port, the protocol class out signals become the port's in signals, and the protocol class in signals become the port's out signals. This enables two ports of the same protocol class to be connected together without having to define a separate reverse protocol. A connection can be made between two ports of the same protocol by conjugating one of the ports. This is the most common way of establishing communicating between two ports.

### **Cardinality**

The Cardinality field defines the maximum number of port instances that can exist in this role at any given time. If the role is Fixed, then the number of instances of the role instantiated at run-time will be exactly the number defined in the Cardinality field. If the role is Optional, then up to <Cardinality> instances may be created at run-time.

### **Documentation**

Specifies descriptive text about the port role.

## Adding a Capsule Role

---

Capsule roles may be added to a structure editor by dragging a class name from a browser onto the structure diagram.

You can also use the Capsule role tool from the structure toolbox. A pick-list is displayed on the capsule role allowing you to select the class of the capsule role from the list of capsule classes. The first entry in the pick-list is **Create a New Capsule** which creates a new capsule class called 'NewCapsule1'. If you choose to create a new capsule class it is added to the same package as the container capsule class.

The class specifies the "type" for the role. In the case of optional or plug-in roles, instances of other classes may actually be incarnated or imported into the role at execution time if they are of compatible types (that is, they have the same interfaces and are subclasses of the specified capsule role class).

## Capsule Role Specification

---

The Capsule Role Specification provides control over the properties of a capsule role in a capsule structure diagram. Capsule roles are references to capsule classes.

The Capsule Role Specification Dialog is a standard Specification Dialog, with additional fields controlling the properties of the capsule role.

### General Tab

#### Name

The name of the capsule role within the container capsule structure. The capsule role name may be used in the detailed code of the container capsule.

#### Class

The Class field defines the Capsule Class to be used in instantiating this role. If the capsule role is an Optional role or a Plug-In role, then subclasses of the specified Class may also be instantiated into this role, but only if the substitutable flag is checked.

## **Cardinality**

The Cardinality field defines the maximum number of capsule instances that can exist in this role at any given time. If the role is Fixed, then the number of instances of the role instantiated at run-time will be exactly the number defined in the Cardinality field. If the role is Optional, then up to <Cardinality> instances may be created at run-time. See Cardinality options.

## **Substitutable**

This check box indicates whether subclasses of the specified capsule role's class can be instantiated into this role. This may happen in one of two ways:

- 1 If the capsule role is Optional, the container capsule may instantiate a subclass of the specified capsule class into the capsule role.
- 2 A subclass of the container capsule may override the class of the inherited capsule role.

## **Fixed**

If the fixed check-box is checked, then a capsule of the specified class is automatically instantiated into the role in every instance of the container capsule at run-time. A number of instances equal to the specified cardinality will be created at initialization time.

## **Optional**

If the optional check-box is checked, then the capsule role is instantiated under the program control of the container class. The container class must explicitly instantiate the capsule role within the detailed code of the container capsule state machine. This is done using the incarnate function of the Frame service.

## **Plug-In**

If the Plug-In check-box is checked, then the capsule role is never directly instantiated, but rather an already existing instantiation from another capsule decomposition is imported into the role. That is, an existing capsule is dynamically “plugged in” to the specified role under the program control of the container class. The container class state machine must explicitly request the plug-in of a capsule at run-time within the detailed code. This is done using the import function of the Frame service.

## Connecting Ports on Capsule Roles Together

---

To enable communication between capsules, you must connect together the ports on their interfaces.

You can only connect compatible ports together. For a port to be compatible, the out signals on each side must be a subset of the in signals on the other side. Usually, this is satisfied by connecting the base role and conjugate role of the same protocol together.

## Connector Specification

---

The Connector Specification provides control over the properties of a connector in a capsule structure diagram. Connectors connect ports together to enable communication among capsules.

There are two tabs: General and Files.

### General Tab

#### Name

The name of the connector. Connector names are not usually displayed on the structure diagram and are not significant in the generated code.

#### Delay

Specifies a communication delay across a connector. This field is for documentation purposes only. There is no validation or calculation of actual communication delays at run-time.

#### Cardinality

Specifies the number of connectors indicated by a connector line. When a connector is used to connect ports with cardinality  $> 1$  or ports on capsule roles with cardinality  $> 1$ , the connector cardinality should match the cardinality of the port/capsule combination on either side of the connection.

## Creating a Collaboration Diagram

---

**To create a new collaboration diagram:**

- 1 Select a package, class, capsule, or use case in the Logical View or Use Case View where you want to define the collaboration
- 2 Right-click on the element in the model browser.
- 3 Select **New > Collaboration Diagram**.
- 4 Enter the name for the collaboration diagram

## Using the Collaboration Diagram Editor

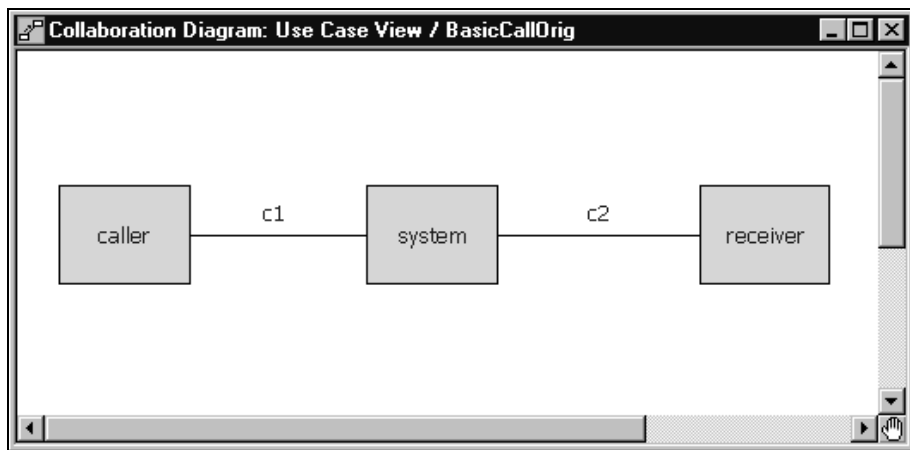
---

The collaboration diagram editor is used to create a diagram showing associations among object roles. An association between classifier roles is called an association role. A collaboration diagram represents a particular object configuration at run-time. The collaboration diagram consists of two parts: the diagram area and the Collaboration Diagram Toolbox. Multiple Collaboration diagrams can exist in the same model.

Elements of the collaboration diagram - such as classifier roles, capsule roles, and association roles - are added using the toolbox.

The window title bar shows the full name of the collaboration diagram.

**Figure 46 Collaboration diagram editor**



## Relationship Between Collaborations and Sequences

The collaboration diagram editor shows the general communication pattern among a set of objects for a particular scenario at run-time. You can associate sequence diagrams with a collaboration diagram. The relationship is that a sequence diagram shows a particular execution of a given scenario. There may be many sequences showing different alternative paths for the same scenario. They should all have the same basic collaboration pattern, though.

In the example above, the scenario is a telephone call. There are three roles being played by objects at run-time. A caller represents the object initiating the call. The receiver represents the object receiving the call. The system represents the object that makes the connection between them. Several sequence diagrams could be derived from this collaboration. For example, one sequence diagram might show a completed call where the receiver answers. Another sequence might show a call that is not answered.

## Opening a Sequence Diagram

To open a dialog listing all the Sequence diagrams associated with a particular Collaboration diagram, select **Open Sequence Diagrams** from the popup menu.

## Sequence Overlays

You can also overlay Message Flow Arrows from a Sequence diagram on top of the Collaboration dialogs by selecting **Sequence Overlays...** from the popup menu. Only "Request" actions - Call and Send - are shown. Create and Destroy messages are not. Messages are only displayed when there is an existing Association Role or Connector to bind them to. Messages To or From the Environment are not displayed.

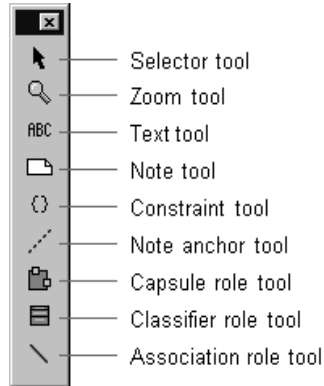
## Code Generation

There is no code generated from the collaboration diagram. It is for communication purposes only. The capsule structure diagram is a specialized form of collaboration diagram with specific constraints that enable code to be generated to implement the communication patterns shown in the capsule structure.

## Collaboration Diagram Toolbox

The collaboration diagram toolbox contains tools for adding elements to the collaboration diagram.

**Figure 47 Collaboration diagram toolbox:**



### Selector Tool

Selects objects for moving, resizing, and so forth.

### Zoom Tool

Zooms in on a portion of the diagram. Select the tool and then click on the part of the diagram you want to zoom in on.

### Text Tool

Adds text anywhere in the structure diagram.

### Note Tool

Annotates the diagram with textual notes. This is useful for marking up the diagram with explanations, review comments, and so forth. You can drag and drop a diagram or external document from the browser onto a note. Notice that the name of the diagram or external document is underlined. If you double-click on the note, the diagram or external document is opened. You can undo and redo this command.

### Constraint Tool

Adds UML constraints to the diagram. A constraint can be anchored to a view element by using the anchor tool. Currently, constraints do not have any semantic meaning to the tool. There are RRTEI APIs to add or remove, and enumerate constraints in a diagram.



### **Note Anchor Tool**

Anchors a note to a particular element on the diagram.

### **Capsule Role Tool**

Places a capsule\_role on the collaboration diagram. When you place a capsule role, a pick-list is displayed allowing you to select from available capsule classes, create a new capsule class, or leave the class unspecified. The class specifies a type that must be satisfied by any instances in that role. In practice, this usually means that subclasses of the specified capsule class can fill the role.

This tool also appears on the capsule Structure Diagram Toolbox. The tool performs the same function in both diagrams.

### **Classifier Role Tool**

Places a classifier role on the collaboration diagram. When you place a classifier role, a pick-list is displayed allowing you to select from available classifier classes, create a new class, or leave the class unspecified. The classifier specifies a type that must be satisfied by any instances in that role. In practice, this usually means that subclasses of the specified class can fill the role.

### **Association Role tool**

Draws a connection between two roles (capsule roles or classifier roles). An association between roles is a form of association with more explicit meaning than an association at the class level. It specifies that instances satisfying the types specified for these roles have some form of direct communication relating to the interaction specified for this collaboration.

## **Classifier Role Specification**

The **Classifier Role Specification** provides control over the properties of a classifier role in a collaboration diagram. The **Classifier Role Specification** is a standard Specification dialog, with additional fields controlling the properties of the classifier role.

There are two tabs: the **General Tab** and the **Files Tab**.

## General Tab

### Name

The name of the classifier role within the collaboration.

### Stereotype

A stereotype label for the association.

### Classifier

Specifies a class to fill this role.

**Note:** If the **Classifier** box contains a class, the corresponding label becomes a hot link to the Specification dialog for that class.

### Multiplicity

The multiplicity field defines the maximum number of instances that can exist in this role at any given time.

### Documentation

Use to describe this classifier role.

## Files Tab

A list of referenced files is provided here. The files list popup menu allows you to insert and delete references to files or URLs.

You can link external files to model elements for documentation purposes.

## Association Role Specification

The Association Role Specification provides control over the properties of an association role in a collaboration diagram.

The Association Role Specification is a standard Specification dialog, with additional fields controlling the properties of the classifier role.

There are two tabs: the General Tab and the Files Tab.

## General Tab

### Name

The name of the association role within the collaboration.

**Stereotype**

A stereotype label for the association.

**Association**

Specifies a class to fill this role.

**Multiplicity**

The multiplicity field defines the maximum number of instances that can exist in this role at any given time.

**Documentation**

Use to describe this association role.

**Files Tab**

A list of referenced files is provided here. The files list popup menu allows you to insert and delete references to files or URLs.

You can link external files to model elements for documentation purposes.



## Contents

This chapter is organized as follows:

- *Creating Capsule State Machines* on page 229
- *Using the State Diagram Editor* on page 230
- *Aggregating and Decomposing State Machines* on page 235
- *Transition Specification* on page 235
- *Choice Point Specification* on page 237
- *Initial State Specification* on page 237
- *Junction Point Specification* on page 238
- *Event Editor Dialog* on page 239
- *Adding a State* on page 242
- *Adding a Choice Point* on page 242
- *Drawing Transitions Between States* on page 242
- *Drawing the Initial Transition* on page 245
- *Defining State Transition Trigger Events* on page 246
- *Joining Transitions* on page 247
- *Creating Nested States* on page 248
- *Positioning from a Superclass for Transitions* on page 248
- *State Diagram - Showing Triggers and Code for Transitions* on page 250
- *Identifying Self Transitions on the Transitions Tab in the State Specification Dialog Box* on page 254

## Creating Capsule State Machines

---

Capsules are one of the primary modeling elements in Rational Rose RealTime. Complete executable code implementations are generated by the toolset for capsules.

Capsule behavior is defined through the **State Diagram Editor**.

You can include the following types of behavioral elements to a capsule behavior diagram:

- States
- Transitions
- Choice points

None of these elements are required. A capsule does not have to have any states or transitions. If the capsule has any interfaces (end ports) in its structure definition, then it must have a state machine to deal with events arriving on its interfaces.

Creating a complete capsule state diagram definition can consist of any of the following basic steps:

- Adding a State
- Adding a Choice Point
- Drawing Transitions Between States
- Defining State Transition Trigger Events
- Joining Transitions
- Creating Nested States

## Using the State Diagram Editor

---

The state diagram editor is used to define the finite **State machine** for a class. The utility of the state diagram depends on the type of element it is specifying:

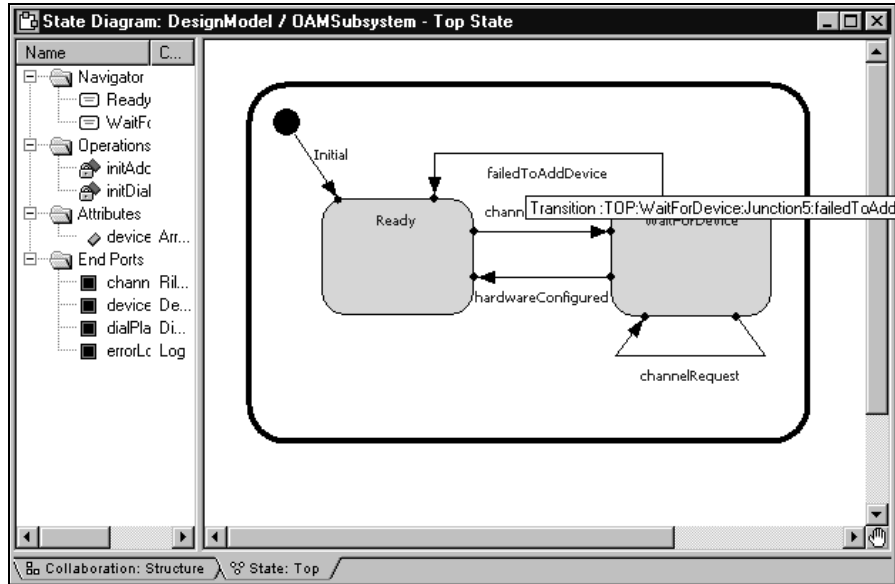
- For capsule classes, the state diagram will result in a complete code implementation generated for the class. The state diagram defines the majority of a capsule class implementation. The capsule class may also have operations defined on it, but the state diagram gives the capsule its asynchronous message processing capability.
- For protocols, the state diagram specifies the expected operation of any capsules that contain one of the protocol's roles. The protocol state diagram defines the allowable sequence of message inputs and outputs with respect to the protocol roles. There is no code generated for the protocol class behavior.
- For data classes, the state diagram captures the abstract behavior (often the abstract modes of operation) for the class. This does not result in any code being generated for the data class. The data class implementation is limited to the definitions of any attributes and operations specified through the Class Specification.

The state diagram consists of three parts: the diagram area, the navigator area, and the toolbox. Multiple State diagrams can exist in the same model.

Behavior elements, such as states and transitions, are added using the toolbox.

The window title bar shows the full name of the class.

**Figure 48 State Diagram Editor**



### State Diagram Elements

The state editor window has tabs on the bottom to allow quick navigation to any nested states, and to the capsule structure editor. You can use the popup menu to navigate between diagrams, as well.

The state diagram allows you to create or edit the following elements:

- States
- State Transitions
- Choice Points
- Initial point and initial transition
- Junction points
- Final States

The state machine can be nested, allowing you to create hierarchical state machines. Hierarchical state machines maintain a state history. When a transition terminates on a hierarchical state, the history mechanism may be triggered to determine which substate becomes the active state.

## Using the Navigator

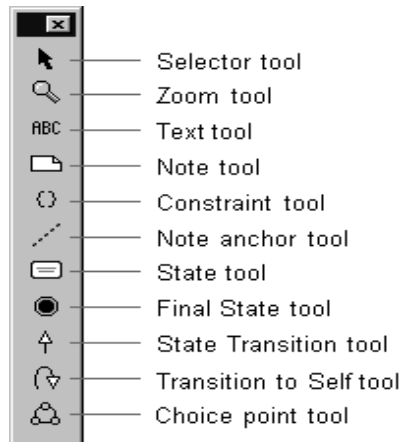
The Navigator area lists the states in hierarchical order, as well as operations, attributes, and ports (in state diagrams only).

Right-clicking on the items in the list provides a shortcut to many common operations, such as adding operations, attributes, and ports.

## State Diagram Toolbox

The state diagram toolbox contains tools for adding elements to the state diagram. The toolbox is associated with the State Diagram Editor (see *Using the State Diagram Editor* on page 230).

**Figure 49** State diagram toolbox



### Selector Tool

Select objects for moving, resizing, and so forth.

### Zoom Tool

Use to zoom in on a portion of the diagram. Click on the tool and then click on the part of the diagram you want to zoom in on.

### Text Tool

Adds text anywhere in the structure diagram.



## **Note Tool**

Annotates the diagram with textual notes. This is useful for marking up the diagram with explanations, review comments, and so forth. You can drag and drop a diagram or external document from the browser onto a note. Notice that the name of the diagram or external document is underlined. If you double-click on the note, the diagram or external document is opened. You can undo and redo this command.

## **Constraint Tool**

Adds UML constraints to the diagram. A constraint can be anchored to a view element by using the anchor tool. Currently, constraints do not have any semantic meaning to the tool. There are RRTEI APIs to add or remove, and enumerate constraints in a diagram.

## **Note Anchor Tool**

Use to anchor a note to a particular element on the diagram.

## **State Tool**

Adds a state to the diagram. Click on the diagram to place a new state at the selected location.

States have default names, such as 's1', when initially drawn. To change the name, click on the state and hit the BACKSPACE key to delete the default name, then type the new name.

## **Final State Tool**

Adds a terminal state to the diagram. Click on the diagram to place a new final state at the selected location.

Transitions cannot be drawn initiating from a final state.

Capsule state diagrams cannot have a final state, so this tool is not displayed for capsules.

## **State Transition Tool**

Draws Transitions from one state to another, from a state to a branch, from a branch to a state, from a transition junction point on the superstate to a substate or to a transition exit point on the superstate, or from the initial point to an initial state.

### **Transition to Self Tool**

Draws a transition from a state back to itself. This can include self transitions on the outer state border, as well as on any substate.

### **Choice Point Tool**

Adds a branch point allowing a transition to branch to two alternate destination states.

## **State Specification**

The state specification allows you to enter details about the state.

The state specification dialog contains the following tabs: General, Entry Actions, Exit Actions, Files.

**Note:** If this state is a top state, an initial point, or a final state on a data or protocol class, then it will not contain any **Entry Actions or Exit Actions tabs**.

### **General Tab**

#### **Name**

The name of the state. The state name appears on the state diagram, and will be part of the generated code for any capsule class. It will also be used in the verification of any sequence diagrams involving the capsule if the sequence diagram used as the specification contains state information.

#### **Class**

The class whose state machine this state is a part of.

### **Entry Actions / Exit Actions Tabs**

#### **Code**

A Code Editor used to enter the detail code that will be executed upon entry to or exit from the state.

This field may also be modified from the generated code and captured into the model using the Code Sync feature. For more information, see *Using Code Sync to Change Generated Code* on page 531.

## Aggregating and Decomposing State Machines

---

You can create new superstates by aggregating several states, transitions and choice points. To aggregate several states into a new superstate, multiply select the states and choose **Parts > Aggregate**. A new state is created containing the selected states.

A superstate can be decomposed into its immediate substates by selecting **Parts > Decompose**.

## Transition Specification

---

The transition specification is used to edit the properties of a state transition. There are up to four tabs: General, Triggers, Actions, and Files.

If the transition is an initial transition, or is not the originating segment of a joined transition, then the Triggers tab is not displayed.

### General Tab

#### Name

The name of the transition. If the transition is part of a capsule state diagram, the transition name will appear in the generated code for a capsule.

#### Internal

This check box indicates that a self-transition should not cause an exit from the state when triggered. The result is that when an internal transition is triggered, no exit or entry code is run.

### Triggers Tab

#### Triggers List

The triggers list is used for Defining State Transition Trigger Events. The triggers list contains the list of individual trigger events. Each event consists of a port name, a signal or set of signals, and an optional guard condition. The Transition Events tab contains the list of events that can trigger the transition. The list is an 'OR' list, meaning that the receipt of any one of the signals in the event list will cause the transition to fire. There is no 'AND' definition of event triggers (since only one message is processed at a time).

To add new trigger events, right-click in the list area and select **Insert** from the popup menu. This brings up the Event Editor Dialog allowing you to select the port(s) and signal(s) that will act as trigger events.

Filter check boxes provides options to display *Inherited* values, *Local* values and *Excluded* values.

## **Moving and Copying Triggers**

To move a trigger from one Specification sheet to another, drag and drop it. From the **Edit** menu of the main window, you can select **Undo** and **Redo**.

To copy a trigger from one Specification sheet to another, drag and drop it while holding down the Ctrl key. From the **Edit** menu of the main window, you can select **Undo** and **Redo**.

## **Actions Tab**

### **Code**

Contains a Code Editor for defining detailed action code.

This field may also be modified from the generated code and captured into the model using the Code Sync feature. For more information, see *Using Code Sync to Change Generated Code* on page 531.

For capsules, the transition action code will be output as part of the generated code, and the code will be executed when the transition is triggered at run-time.

Transition actions defined in state diagrams for protocols or regular (non-capsule) classes is not generated or executed. It is for information purposes only.

### **Files Tab**

The Files tab allows for linking external files to the transition.

## Choice Point Specification

---

The Choice Point Specification contains three tabs: General, Condition, and Files.

### General Tab

Contains standard specification dialog items.

### Condition Tab

Contains a Code Editor for entering the code that determines which branch of the transition will be taken. The code must return a true or false value (false is zero and true is non-zero).

This field may also be modified from the generated code and captured into the model using the Code Sync feature. For more information, see *Using Code Sync to Change Generated Code* on page 531.

### Files Tab

A list of referenced files is provided here. The files list popup menu allows you to insert and delete references to files or URLs.

You can link external files to model elements for documentation purposes.

## Initial State Specification

---

The Initial State Specification contains two tabs: General and Files.

### General Tab

#### Name

The default name for the initial state is Initial and should not be changed.

#### Class

A non-editable field indicating the class whose state machine the initial state is part of.

#### Documentation

A description of the initial state.

## Files Tab

A list of referenced files is provided here. The files list popup menu allows you to insert and delete references to files or URLs.

You can link external files to model elements for documentation purposes.

## Junction Point Specification

---

The dialog shows information about the junction point. See the *Modeling Language Guide* on junction points and history for more information on these selections.

There are only two tabs: General and Files.

### General Tab

#### Name

A name for the junction point. Most junction points are given automatically generated names.

#### Continuation

This selection specifies the semantics for how the state history will be used when there is no continuing transition. There are three options:

- **Default** - specifies that the default (initial) transition should be run.
- **History** - specifies that the state should return to shallow history.
- **Deep History** - specifies that the state should return to deep history, meaning that all substates also return to history. This is the behavior for all capsule state machines, so it is automatically selected.

**Note:** The default for capsule state machines is to always go to deep history, so deep history is automatically selected for capsule states, and the selections are grayed out.

#### Externally Visible

This check box indicates whether the junction point is visible on the outside of the state boundary.

## Files Tab

A list of referenced files is provided here. The files list popup menu allows you to insert and delete references to files or URLs.

You can link external files to model elements for documentation purposes.

## Event Editor Dialog

---

The event editor dialog is used to define triggering events for transitions in capsule state diagrams. The event editor is accessed from the Events tab of the transition specification dialog.

The event editor contains:

- A ports list
- A signals list
- A guard code area

The ports list contains a list of end ports in the capsule's collaboration diagram. Only end ports that have In signals are listed as they are the only ones capable of receiving messages.

The guard code may also be modified from the generated code and captured into the model using the Code Sync feature. For more information, see *Using Code Sync to Change Generated Code* on page 531.

### To specify the trigger event:

- 1 Select a port check box to display the list of signals that can be received on that port.

Multiple ports from the ports list can be selected, but when selecting multiple ports, the signals are only displayed if all ports share the same protocol.

The signals list displays signals that can be received by the currently selected end port, as well as a default wild card selection indicated by a '\*'.

- 2 Select one or more signals from the signal list.

Multiple signals from the signals list can be selected. Any one of the selected signals will act as a trigger for the transition.

## EventGuard Specification Dialog Box

An **EventGuard** is a grouping of an Event and a Guard that will trigger a transition. Use the **EventGuard Specification** dialog box to define triggering events for transitions in **State Diagrams** for capsules. The **EventGuard Specification** dialog box contains the following areas:

- Ports List
- Signals List
- Guard Code

**Note:** This dialog box is only available for events on capsule state machines. Data class state machines use a dialog box where you can select a trigger stereotyped operation (see *Creating State Machine Trigger Operations* on page 182).

**EventGuards** do not have names. As a result, the **EventGuard Specification** dialog box and the **Specification History** windows show "Untitled" in the title.

### Ports List

The **Ports** list contains end ports that currently exist in the **Collaboration Diagram** for a capsule. Only those end ports that have **In signals** appear in this list because they are the only ones capable of receiving messages.

**Note:** You cannot specify triggering events until you define the end port on which the event will arrive.

A port that is not an end port, which is then made into an end port, will not appear in the **Ports** list until the **EventGuard Specification** dialog box closes.

### Signals List

When you select a port from the Port list, the Signals list will contain all of the *in signals* from that port, as well as the universal signal (\*), **rtBound**, and **rtUnbound**. Use the universal signal to specify that any signal received on the selected port will trigger the transition.

If there is more than one port selected in the **Port** list, and all the selected ports use the same protocol, or use protocols that derive from a common protocol, then signals from that common protocol appear in the **Signal** list. Otherwise, only the universal signal (\*) appears in the **Signal** list. You can select the universal signal if it is the only signal selected. If other signals were selected previously, they will no longer be selected.



## Guard Code

The Guard Code area allows you to specify the code for a guard condition for this transition. This code is executed before the transition occurs and determines whether to take this transition. If the guard is True, the transition is taken to the next state. If the guard is False, the transition is not taken and event processing will continue (the runtime service library will attempt to find another transition that triggers from the same port or single pair).

**Note:** You can also modify **Guard Code** from the generated code and then capture the changes into the model using the Code Sync feature. For more information on code sync, see *Using Code Sync* in the online Help.

### To create a trigger event:

- 1 Ensure that you have defined the end port on which the event will arrive.
- 2 On the **State Diagram**, select the transition.
- 3 Click the **Triggers** tab.
- 4 In the **Port, Signal, or Guard** box, right-click and select **Insert**.

**Note:** If the **EventGuard** currently exists, select the port name from the **Port** column in the **Triggers** tab, and then right-click and click **Open Specification**.

- 5 Select a port.

**Note:** If a port does not appear in the **Port** list, right-click and select **Insert Port**.

The signals for the selected port (the signals that can be received on that port) display in the **Signals** list.

**Note:** Multiple ports from the ports list can be selected, but when selecting multiple ports, the signals are only displayed if all ports share the same protocol.

The signals list displays signals that can be received by the currently selected end port, as well as a default universal signal selection (\*).

- 6 Select one or more signals in the **Signal** list.

Multiple signals from the signals list can be selected. Any one of the selected signals will act as a trigger for the transition.

- 7 Specify code for a guard condition in the **Guard Code** area, if required.
- 8 Click **OK**.

## Adding a State

---

States can be added by clicking on the state tool in the state diagram toolbox, and then clicking on the state diagram where you want to add a state.

Alternatively, you can add states through the navigator area of the state diagram editor.

If you have state entry or exit actions to define, use the state specification dialog or Code window.

## Adding a Choice Point

---

To draw a choice point, click on the choice point tool from the state diagram toolbox and then click on the diagram where you want the choice point added.

The choice point can be rotated by grabbing one of its handles and turning. The true and false branches can be flipped using the popup menu.

After you add a choice point, you should define the condition for the choice point.

## Drawing Transitions Between States

---

The transition tool is used to draw transitions.

Transitions are drawn originating from states, transition join points or choice points and terminating on those same elements.

To draw a transition, click on the transition tool, click on the originating element for the transition and drag the transition on to the terminating element.

When a new transition is drawn, the transition does not usually have a triggering event. Transitions with no trigger event are shown with a broken line:



Transitions containing code are shown with the arrowhead filled in black.

## Drawing Transitions

There are two options that affect how the transition the user draws is placed on the diagram, and one option that affects how the transition line can be drawn and re-drawn. All global options are on the **Diagrams** tab when you click **Tools > Options**.

**Auto-adjust transitions** is in the **Miscellaneous** area. There is also a shortcut menu option for junction points, called **Auto Adjust**, that will be enabled for fixed junction points of regular transitions.

The **Line Style** and **Routing** options are accessed in the **Line Attributes** dialog available by clicking **Default Line Attributes** in the **Display** area. There is also a shortcut menu option called **Line Attributes** for transition lines.

**Line Style** affects the appearance of the transition line:

- **Oblique** - Lines may have line segments at any angle. If smoothing is applied the lines will be curved instead of straight.
- **Rectilinear** - Lines may only have vertical and horizontal segments. Smoothing is disabled for rectilinear lines.
- **Auto-adjust transitions** - Affects the placement of the end points of the transition. When selected, the points may adjust to avoid existing points, and the routing algorithm is free to move the endpoints from where they were initially placed by the user. Each endpoint continues to "float" until the user moves it, thereby fixing its' relative position on the state (the position is relative since the state may be resized).

**Note:** The points on a self-transition do not auto-adjust. However, if the transition is converted to a regular transition by moving one of its' endpoints to another state, then the auto-adjust status of the remaining endpoint determines whether it will be moved by the **Routing** algorithm. The shortcut menu entry performs the auto adjustment and sets the status of the point back to floating.

## Regular Transitions (Not Self Transitions).

**Routing** is used in conjunction with **Auto-adjust transitions** to determine how the transition line is redrawn from the original placement by the user. The following combinations apply:

- **Auto-adjust transitions** off - The endpoints remain where the user placed them
  - Normal routing - The line will remain as drawn by the user
  - Closest distance - The line will be redrawn as a single straight line in the "Oblique" line style and the shortest set of horizontal and vertical segments for the "Rectilinear" line style.
  - Avoid obstructions - Both line styles will be redrawn to avoid obstructions (if possible) - the line will be redrawn as a the shortest set of line segments in the "Oblique" line style and the shortest set of horizontal and vertical segments for the "Rectilinear" line style.
- **Auto-adjust transitions** on, - The endpoints may be moved from where the user placed them
  - **Normal** routing - The line drawn may be adjusted to connect the end points (which may have moved because they were auto-adjusted).
  - **Closest distance** - The lines drawn are similar to those for **Closest distance** above except that the endpoints will be moved to give the closest distance, instead of the closest distance between fixed endpoints.
  - **Avoid obstructions** - The lines drawn are similar to those for **Avoid obstructions** above except that the endpoints will be moved to give the closest distance, instead of the closest distance between fixed endpoints.

Self transitions endpoints do not auto adjust. However, the self transition line has a standard form which usually consists of three line segments: two short ones coming from the state border, and another that is as long as necessary to connect the other two segments. If smoothing is applied, the three segments may appear to be one curved line. The three line segments may be collapsed to one segment if:

- The endpoints are on parallel borders of the state.
- **Closest distance** or **Avoid obstructions** routing is in effect.

Self transition lines are redrawn to this standard form when the self-transition is originally drawn, when one of the endpoints is moved or when the state is resized. Bend points may be inserted in the self transition line, but any adjustments will cause the line to be redrawn to standard form.

For additional information on drawing transitions, see *Drawing the Initial Transition* on page 245.

## Specifying the Transition

Once you have drawn a transition, you can specify the transition details. The details of a transition include the trigger event(s), and the action code. These are specified through the Transition specification dialog.

The trigger event and action code for capsule state machines result in generated code as part of the capsule implementation. No code is generated for the trigger event and action for other class state machines.

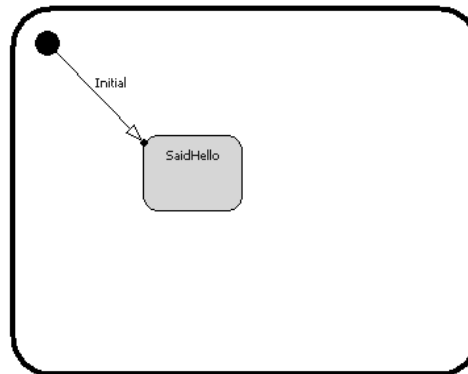
## Drawing the Initial Transition

---

**To draw an initial transition in a state diagram editor:**

- 1 Click on the transition tool in the state diagram toolbox.
- 2 Click on the initial point in the diagram and drag the transition on top of the target state. The initial point is the black circle that appears in the top-left corner of the diagram.

**Figure 50** Initial transition



The initial transition has a default name of 'Initial'. You can change the name by selecting the label and typing in it.

**Note:** If you select a label, the tether to that label displays for a short time only. To view the tether for longer periods, press and hold the left mouse button. The tether to the label will display for as long as you hold the left mouse button. If you start to drag the label, the tether is replaced with a tether and tracking box.

# Defining State Transition Trigger Events

---

## State Diagrams

### To define a new state transition trigger event:

- 1 Open the **State Transition Specification** from the capsule State Diagram editor (double-click on the state transition).
- 2 Select the **Events** tab.
- 3 Right-click in the event list area.
- 4 Select **Insert** from the popup menu.  
The Event Editor Dialog appears.

### To define a new event in a capsule:

- 1 Click on the check box for the port and signal items to be included in the event.
- 2 The chosen items have a check mark next to them. Deselecting the check box removes items from the event definition.

**Note:** A state transition trigger event can have more than one signal selected on a port, and can have more than one port selected, though the signals list only shows the signals that are common in the protocols of the two ports in that case. To trigger a transition on signals on different ports, use multiple trigger events. A wild card trigger is available ( \* ) in the signals list which triggers a transition if any of the valid input signals of the currently selected ports is encountered.

### To define a new event in a protocol:

- 1 Click on the check box for the signal items to be included in the event.
- 2 The chosen items have a check mark next to them. Deselecting the check box removes items from the event definition.

### Defining a new event in a data class

To define a new event in a data class, specify the name of the event.

## Joining Transitions

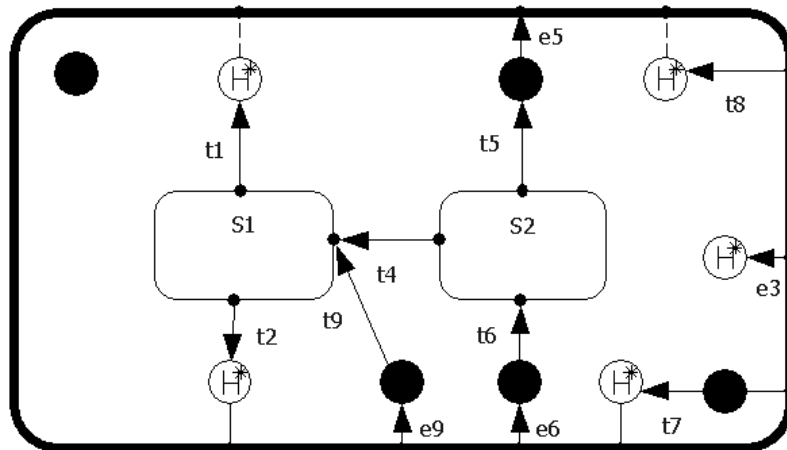
---

Transitions terminating on a superstate can be joined to transitions inside the state to terminate directly on a substate. Similarly, transitions inside a hierarchical state can be joined to transitions leaving the superstate. The points where a transition begins or ends are represented inside the state with join points. Join points may be dark circles or light circles.

To connect a new transition to an existing transition, select the transition tool and draw the transition starting from or terminating on a join point.

To join two existing transitions, select one of the transitions and move it so that the end point lands on the beginning point of the other transition, or so that the beginning point lands on the end point of the other transition.

**Figure 51** Joined Transitions



**Note:** If you select a label, the tether to that label displays for a short time only. To view the tether for longer periods, press and hold the left mouse button. The tether to the label will display for as long as you hold the left mouse button. If you start to drag the label, the tether is replaced with a tether and tracking box.

## Creating Nested States

---

Nested states are created as follows:

- During state creation, select the state icon from the diagram toolbox and place over a targeted superstate.
- Use **Parts menu**, **Aggregate** and copy/paste to move state machine pieces into another state.

The border of the target superstate becomes bold as the nested state moves over it. Once the nested state is dropped on the superstate, the boundaries of the superstate may grow to accommodate the nested state. If the cursor is positioned over more than one state at the same time, the state at the deepest level of nesting is considered the target superstate. Multiple states can be selected and nested as a group.

Nesting is determined completely by cursor position. Once the cursor is moved outside the target state, no nesting occurs. The bold display of the target state's border serves as an indicator for nesting. States can overlap without nesting

## Positioning from a Superclass for Transitions

---

You can define generalization relationships between classes (including capsule and protocol classes) in Rational Rose RealTime. When a generalization relationship is defined, the specializing class inherits the properties (such as all attributes, operations, state machines, and signals) of the generalizing class. You can modify the **State Diagram** for a capsule by positioning the selected transition according to its position in the superclass. This means that when **Position from Superclass** is selected from the context menu for a transition, the transition changes to the same location as on the parent class.

**Note:** This option attempts to position (from the superclass) transitions, states, and choice points on a **State Diagram**.



For example, a model contains two capsules, **MyParent** and **MyChild** (see Figure 52). The capsule **MyChild** is dependent on the parent capsule **MyParent**.

**Figure 52 Class Diagram - Example Model**

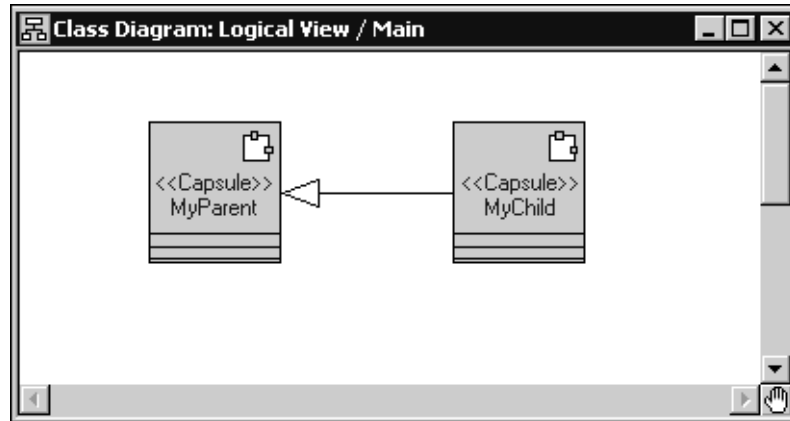


Figure 53 shows the **State Diagram** for the parent capsule **MyParent**.

**Figure 53 State Diagram for MyParent**

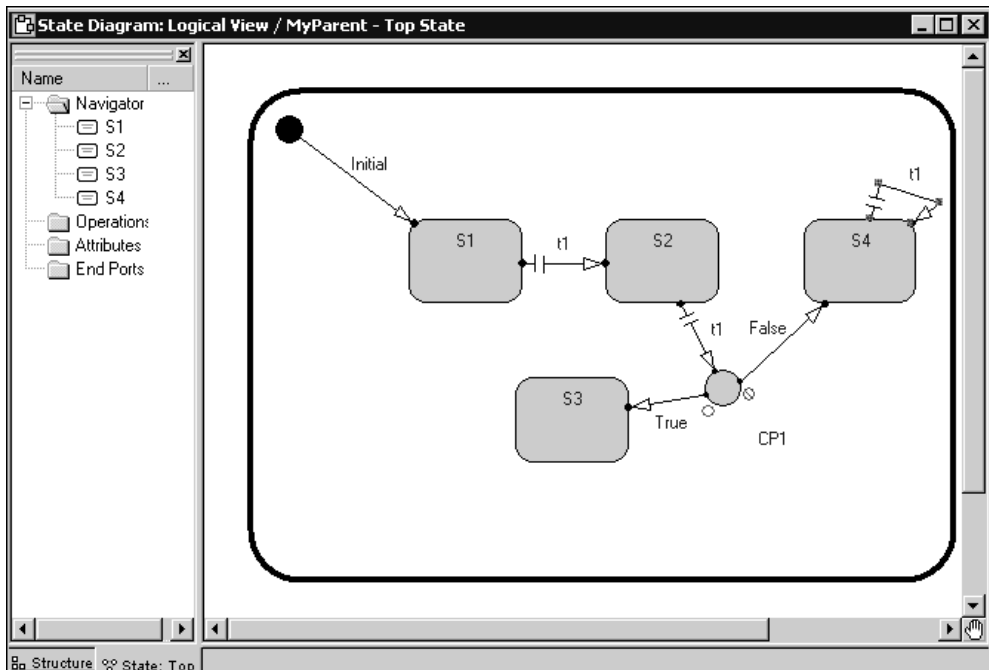
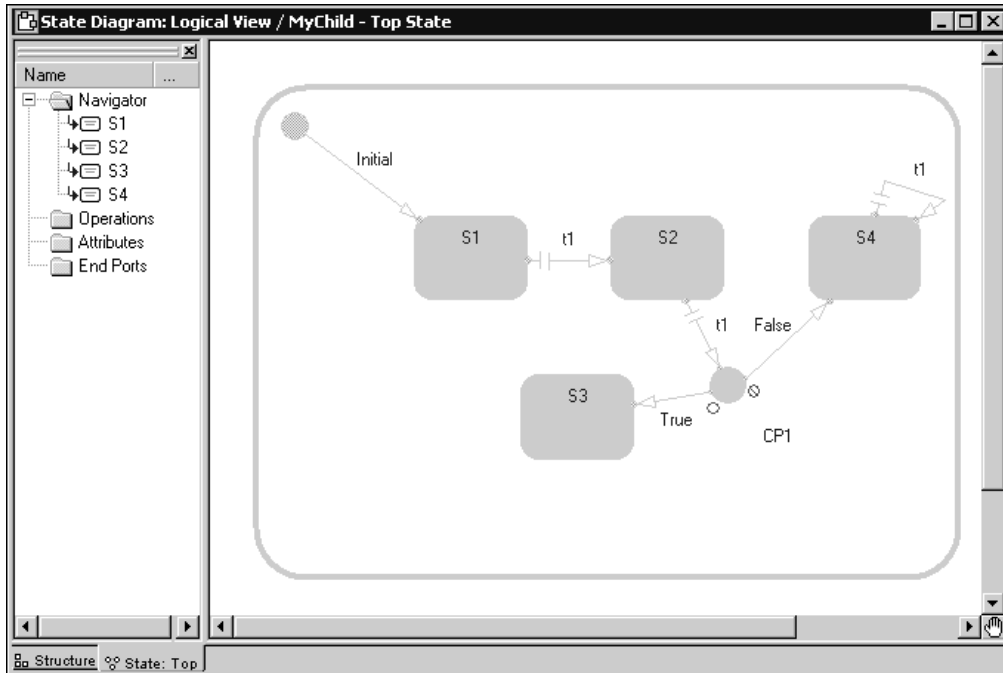


Figure 54 shows the **State Diagram** (inherited from **MyParent**) for **MyChild**.

**Figure 54 State Diagram for MyChild**



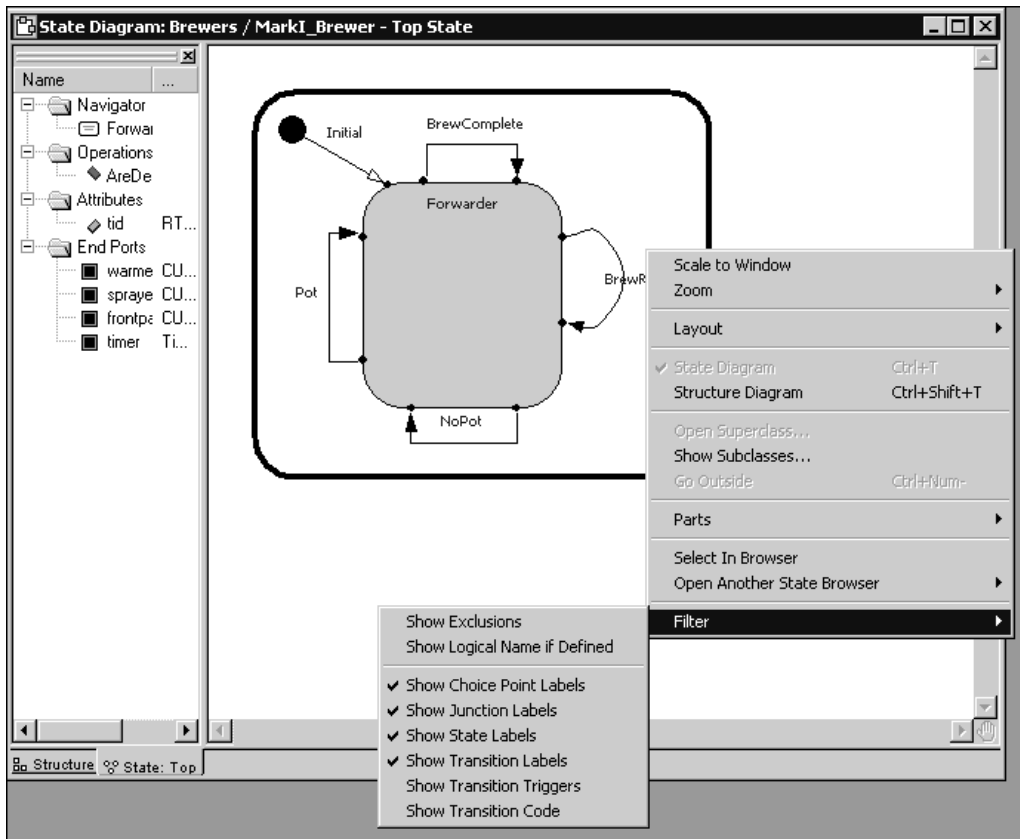
To modify the position of an element (such as a state, transition, choice point, or junction point) on the **State Diagram** for the child, click **Tools > Layout > Reposition selected from superclass** to clear the option so that it is not selected.

## State Diagram - Showing Triggers and Code for Transitions

State Diagrams can show transition triggers and transition code to provide you with immediate visual access to the UML event string for the transition. The **Filter** submenu in the **State Diagram** context menu includes two new options: **Show Transition Triggers** and **Show Transition Code**.

**Note:** If **Show Transition Labels** is not selected, the **State Diagram** will not display any triggers or transition code (even when **Show Transition Triggers** or **Show Transition Code** are selected).

Figure 55 State Diagram - Context Menu



The **Filter** menu on state diagrams now contains two additional options. Select **Show Transition Triggers** to display the UML event string for the transition (similar to what you see in the ToolTip for the transition).

**Figure 56 State Diagram without Transition Triggers or Code**

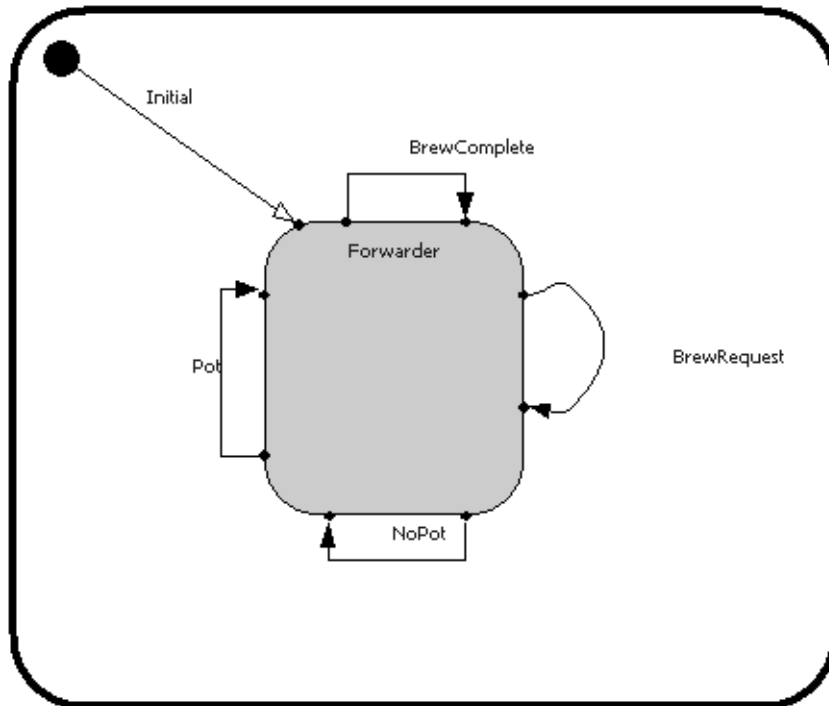
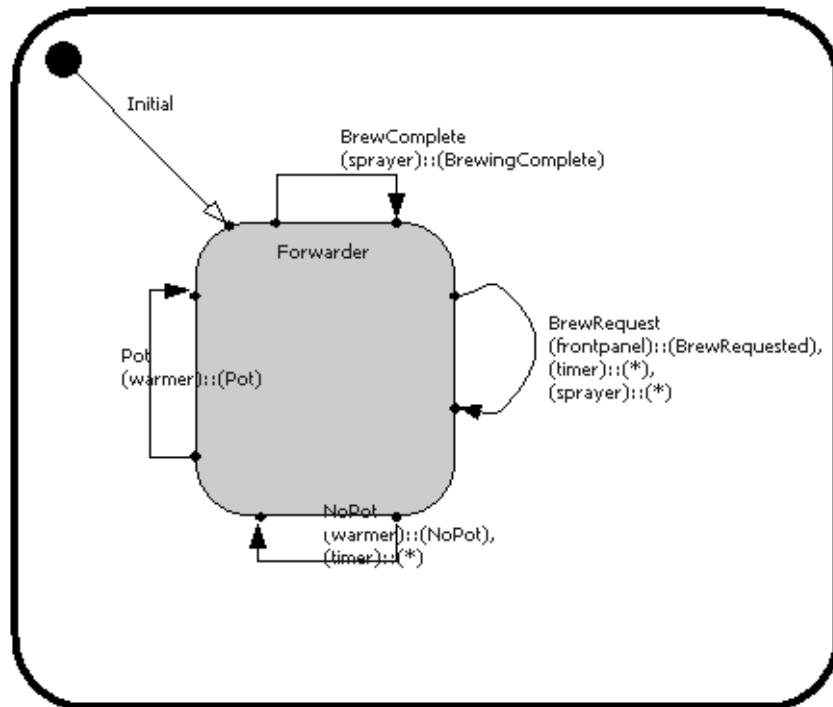
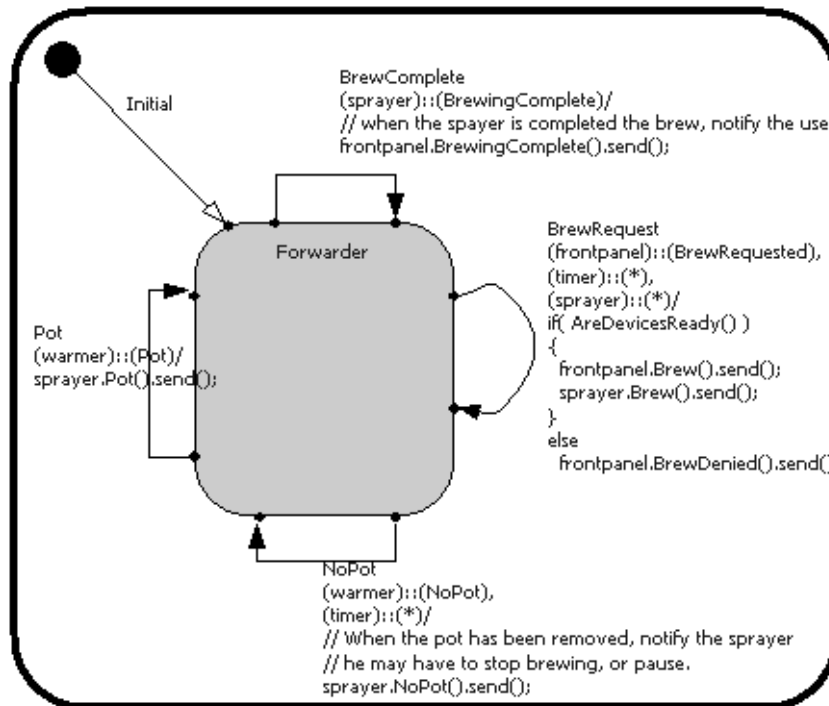


Figure 57 State Diagram Showing Transition Triggers



From the **Filter** submenu, select **Show Transition Code** to display the **Action** code for all transitions in the current state diagram.

Figure 58 State Diagram Showing Transition Triggers and Code



These filter options are specific to the current **State Diagram**; setting these options in one **State Diagram** does not set these options in other **State Diagrams**.

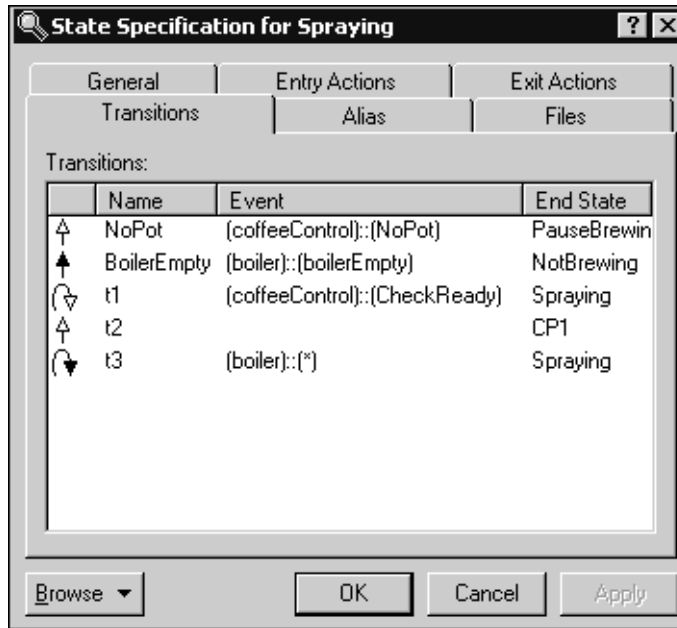
**Note:** You can change the default value for displaying transition triggers and code for new **State Diagrams** by clicking **Tools > Options** and setting **Trigger labels** and **Code labels** in the **State Diagram** area in **Filtering** tab.

## Identifying Self Transitions on the Transitions Tab in the State Specification Dialog Box

You can now view transitions and self transitions (transitions originating and terminating on the same state) directly from the **Transitions** tab in the **State Specification** dialog box.

**Note:** Transitions from the **Initial Point** do not appear in the **Transitions** list.

Figure 59 State Specification Dialog Box




## Descriptions

### Transition Type

Specifies the type of transition entering the selected state (the first column in the **Transition** tab). The transition types are:

- **↑ Internal Transition** - A transition that does not cause a change in state. This type of transition is not visible outside the state boundary.
- **↑ External Transition** - A transition that causes a change in state. This type of transition is visible outside the state boundary.
- **↻ Internal Self Transition** - This type of self transition does not cause an exit from the state when triggered and no exit or entry code is run. This means that the transition executes without exiting or re-entering the state in which it is defined. Also, the exit and entry actions of all states which were exited and re-entered are not executed. These kinds of transitions are similar to having global operations defined on a state machine; when they are taken do not change the state of the system. This type of transition is not visible outside the state boundary.

-  **External Self Transition** - For this type of self transition, the exit and entry code is executed for the state on which it originates and terminates. This type of transition is visible outside the state boundary.

### **Name**

Assigned to each transition entering the selected state. This name appears in the generated code for a capsule.

### **Event**

Contains a list of individual trigger events. An event is an occurrence of a stimulus that causes a state transition. Each event consists of a port name, a signal or set of signals, and an optional guard condition. The data for an event is a UML string indicating the ports and signals that trigger this transition. The text that appears in this column is similar to that seen in the pop-up ToolTip for the transition, excluding the **Guard** code and the **Action** code.

To add new events, right-click in the list area and **Open Specification**, then right-click in the **Triggers** tab in the **Transition Specification** dialog box and click **Insert**. You can now select the ports) and signals that will act as trigger events.

### **End State**

Indicates the name of the final execution state that the transition ends on, based on the completion of the current state. For internal and external self transitions, the end state will be the current state.



## Contents

This chapter is organized as follows:

- *Modeling Using Activity Diagrams* on page 258
- *Creating an Activity Diagram* on page 260
- *Activities* on page 264
- *Activity Specification Dialog* on page 266
- *Actions* on page 269
- *Action Specification Dialog* on page 270
- *Decisions* on page 271
- *Decision Specification Dialog* on page 272
- *End State* on page 274
- *Start State* on page 274
- *States* on page 275
- *State Specification Dialog* on page 277
- *Synchronizations* on page 281
- *Synchronization Specification Dialog* on page 282
- *Transitions* on page 283
- *Transition Specification Dialog* on page 284
- *Swimlanes* on page 286
- *Swimlane Specification Dialog* on page 289
- *Objects and Object Flows* on page 290
- *Object Specification Dialog* on page 294
- *Object Flow Specification Dialog* on page 296
- *Cutting Objects on Activity Diagrams* on page 298
- *Copying Objects on Activity Diagrams* on page 298
- *Pasting Objects on Activity Diagrams* on page 298

## Modeling Using Activity Diagrams

---

Activity Diagrams allow you to model dynamic behavior in a model. Typically, you use an **Activity Diagram** to model the discrete stages of an object's lifetime, and the sequence of activities in a process.

Activity Diagrams provide a way to model the dynamic aspects (workflow) of a business process or system, and to model the dynamic behavior of individual classes, or any other type of object. For example, you can use Activity Diagrams to model code-specific information, such as a class operation. Activity Diagrams are available for Use Cases, Logical Packages, actors, classes, capsules, packages, interfaces, and operations for classes and Use Cases.

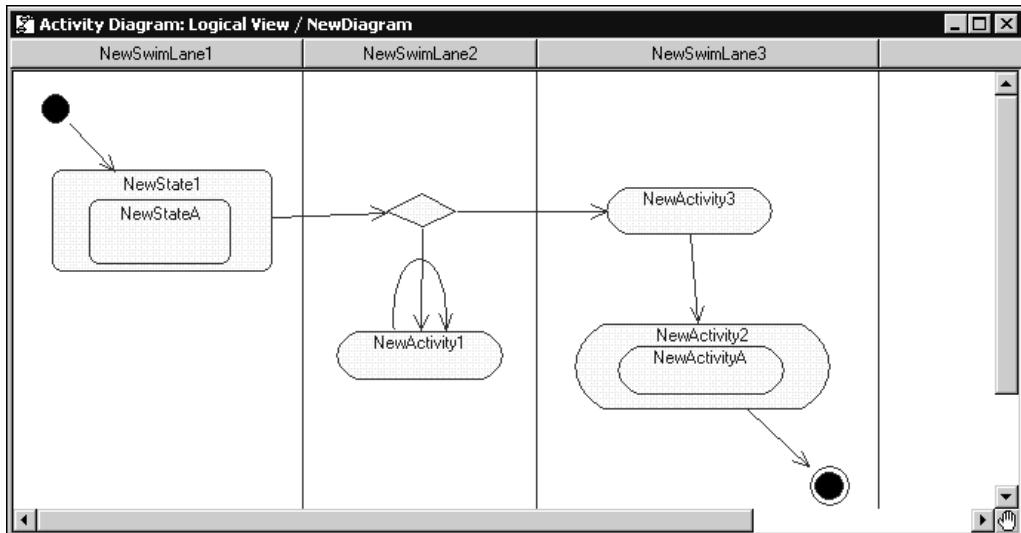
**Activity Diagrams** are very similar to a flowchart because you can model a workflow from activity to activity. An **Activity Diagram** is a special type of state machine in which some of the states are activities, and some of the transitions are implicitly triggered by completion of the actions in the source activities.

Additionally, **Activity Diagrams** can show the sequences of states that an object goes through, the events that cause a transition from one state to another, and the actions that result from a state change. Each state represents a named condition during the life of an object during which it satisfies some condition, or waits for some event.

### Activity Diagrams

**Activity Diagrams** can model many different types of workflows. For example, a company could use **Activity Diagrams** to model the flow for an approval of orders or to model the paper trail of invoices. An accounting firm could use **Activity Diagrams** to model any number of financial transactions. A software company could use **Activity Diagrams** to model a software development process.

**Figure 60 Example of a Simple Activity Diagram**



Typically, an Activity Diagram contains one **Start State** and multiple **End States**. Transitions connect the various states on the diagram. The workflow on an Activity Diagram stops when a transition reaches an end state.

Each state and activity represents the performance of a group of events or actions in a workflow. After the state or activity completes, the flow of control moves to the next state or activity through a transition. If an outgoing transition is not clearly triggered by an event, then it is triggered by the completion of the contained actions inside the activity.

A unique Activity Diagram feature is a swimlane that defines who or what is responsible for carrying out a given state or activity. It is also possible to place objects on Activity Diagrams.

**Note:** You can attach Activity Diagrams to most model elements in the Use Case or Logical Views.

## Creating an Activity Diagram

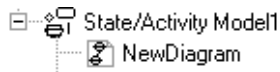
---

You can create an **Activity Diagram** for various model elements. However, if you create more than one **Activity Diagram** for a model element, those diagrams only represent different views for that element. For example, multiple **Activity Diagrams** for a single element will contains only a single start state.

**Note:** If an object has multiple **Activity Diagrams** (meaning that all **Activity Diagrams** at the same level can reference the same objects, but each diagram represents a different view), you must specify a unique name.

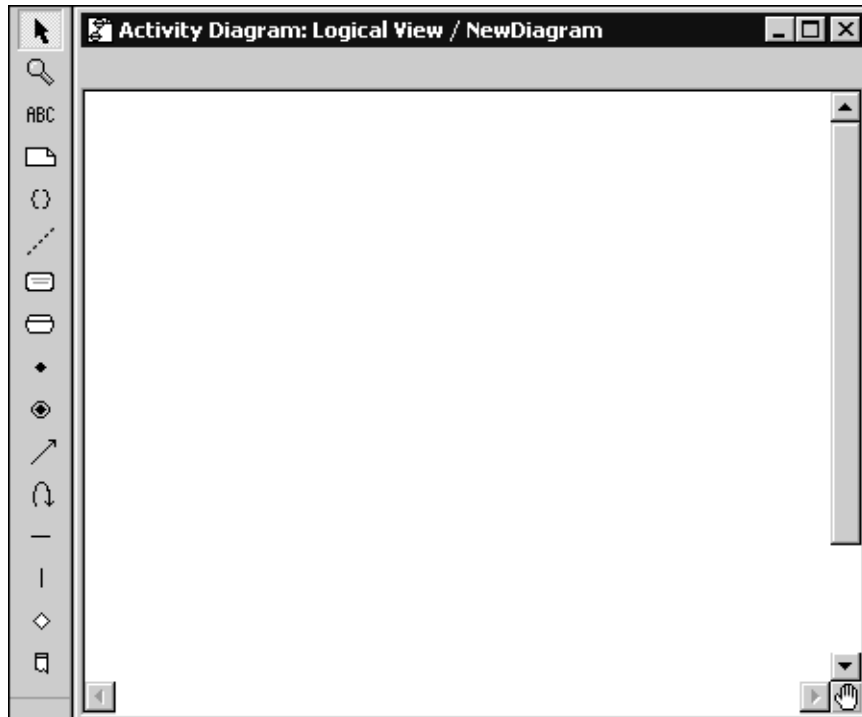
### To create an Activity Diagram:

- 1 On the **Model View** tab in the browser, click on a model element, excluding attributes and associations.
- 2 Right-click and click **New > Activity Diagram**.
- 3 In the **Model View** tab, expand the new entry for the **State/Activity Model** diagram.



- 4 Double-click on the **NewDiagram** name.

An empty Activity Diagram displays.



## Activity Diagram Specification Dialog

---

Use the **Activity Diagrams Specification** dialog to maintain the properties associated with an **Activity Diagram**.

To open the **Activity Diagram Specification** dialog, right-click on the **Activity Diagram** name on the **Model View** tab in the browser, and click **Open Specification**.

The **Activity Diagram Specification** dialog has the following tab:

- **General**, see *Activity Diagram Specification Dialog - General Tab* on page 262

## Activity Diagram Specification Dialog - General Tab

### Name

Specifies the name of the currently selected **Activity Diagram**.

### Documentation

Describes the purpose or intent of the **Activity Diagram**. The description can include information such as the essential behavior of the diagram. The information you type in this field is not displayed in the **Activity Diagram**.

## StateMachine Specification for State/Activity

---

Use the **StateMachine Specification** dialog to maintain the properties associated with a **State/Activity Diagram**.

To open the **StateMachine Specification** dialog, right-click on **State/Activity Diagram** on the **Model View** tab in the browser, and click **Open Specification**.

The **StateMachine Specification** dialog has the following tabs:

- **General**, see *StateMachine Specification for State/Activity - General Tab* on page 262
- **Files**, see *StateMachine Specification for State/Activity - Files Tab* on page 263

## StateMachine Specification for State/Activity - General Tab

### Name

Specifies the name of the currently selected **State/Activity** diagram.

### Stereotype

Specifies a keyword that further defines the classification of the diagram. A stereotype represents the subclassification of a model element. Some stereotypes are already predefined, but you can also define your own to specify new modeling types.

### Owner

Specifies the model elements that own the selected **State/Activity Diagram**.

### Context

Specifies a view for a related set of modeling types.

## **Documentation**

Describes the diagram. The description can include such information as the constraints, purpose, and essential behavior of the element.

## **StateMachine Specification for State/Activity - Files Tab**

### **Filename**

Displays a list of referenced files. You can insert and delete references to files or URLs by linking external files to model elements for documentation purposes.

### **Path**

Specifies the location of the file or URL.

## **Activity Diagram Tools**

---

You can use the following tools on the Activity Diagram toolbox when modeling your Activity Diagrams:

- Activities
- Anchor Note or Constraint to Item
- Constraint
- Decisions
- End State
- Note
- Objects
- Object Flow
- Lock Selection
- Start State
- States
- Swimlanes
- Synchronizations
- Text Box
- Transitions
- Zoom Tool

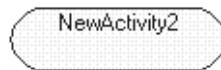
## Activities

---

An activity represents the performance of a task or duty in a workflow. It may also represent the execution of a statement in a procedure. An activity is similar to a state but expresses the intent that there is no significant waiting (for events) in an activity. Transitions connect activities with other model elements.

Figure 61 shows an activity for an Activity Diagram. The name of an activity must be unique and describe the activity's purpose.

**Figure 61 Graphical Representation of an Activity**



### Activity History

The history icon, (H\*), provides a mechanism to return to the most recently visited state or activity when transitioning directly to a state or activity with substates. History applies to the given level in which it appears. History may also be applied to the lowest depth of nested states. You may place an asterisk in the name field of the history state to designate the lowest depth.

To delete history icons, use **Cut** from the toolbar or click **Edit > Delete**.

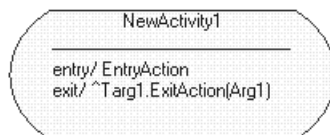
### Specifying Actions for Activities

Actions on activities can occur:

- **on entry** - The task is performed when the object enters the activity.
- **on exit** - The task is performed when the object exits the activity.
- **do** - The task is performed while in the activity and must continue until exiting the activity.
- **on event** - The task triggers an event only when a specific event is received.

Figure 62 shows an activity with actions. You can only add actions for an activity using the **Action Specification Dialog**.

**Figure 62 Graphical Representation of an Activity with Actions**

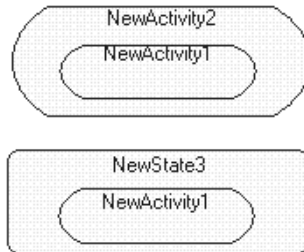




## Nested Activities

You can nest activities to any depth level. Everything that lies within the bounds of the nested activity (sub-activity) is referred to as its contents.

**Figure 63 Graphical Representations of a Nested Activity**



## Manipulating Nested Activities

Nested activities can be moved, resized, and transitioned to and from as if they were top-level activities. However, if you move a nested activity outside a boundary of the super-activity, the size of the container adjusts to accommodate the new position of the nested activity.

You can "un-nest" an activity by selecting the nested activity, then dragging-and-dropping the activity to an empty location on the Activity Diagram.

Nesting is determined by your mouse pointer location and activities can overlap without nesting.

## Creating Nested Activities

You can create nested activities by:

- Selecting the **Activity** icon from the Toolbox and clicking over top of an existing state or activity.
- Use drag-and-drop to place an activity over another state or activity.
- Right-click on an existing activity on the **Model View** tab in the browser, right-click and click **New > Activity**.

The border of the target activity becomes bold as the nested activity moves over it.

**Note:** After you drop an activity, the boundaries of the super-state or super-activity enlarge to accommodate the nested element. If the mouse pointer is positioned over more than one state or activity at the same time, the element at the deepest level of nesting is considered the target. You can select multiple activities and nest them as a group.

## Activity Specification Dialog

---

On the **Activity Specification** dialog, you can view and modify the properties and relationships of an activity on an **Activity Diagram**.

To view the **Activity Specification** dialog, double-click on an **Activity** icon on the **Activity Diagram**, or double-click on an activity on the **Model View** tab in the browser.

The **Activity Specification** dialog has the following tabs:

- **General**, see Activity Specification Dialog - General Tab
- **Actions**, see Activity Specification Dialog - Actions Tab
- **Transitions**, see Activity Specification Dialog - Transitions Tab
- **Swimlanes**, see Activity Specification Dialog - Swimlanes Tab
- **Files**, see Activity Specification Dialog - Files Tab

### Activity Specification Dialog - General Tab

#### Name

Specifies the name for the currently selected activity.

#### Stereotype

Specifies a keyword that further defines the classification of the model element. A stereotype represents the subclassification of a model element. Some stereotypes are already predefined but you can also define your own to specify new modeling types.

To view stereotypes on the Activity Diagrams, click **Tools > Options**, select the **Diagram** tab, and click **Label**, **Decoration and label**, **Decoration only**, or **Icon** in the **Stereotype** box. **Label** displays the stereotype name in angle brackets (for example, <<stereotype>>). **Decoration** displays a graphic marker such as highlighting an icon or tool. **Icon** displays the graphical representation, if any.

**Owner**

Specifies the model elements that own the selected activity.

**Context**

Specifies a view for a related set of modeling types.

**Documentation**

Describes model elements or relationships. The description can include information such as the constraints, purpose, and essential behavior of the element. The information you type in this field is not displayed in the Activity Diagram.

**State/activity history**

Specifies whether to return the most recently visited state or activity when transitioning directly to a state or activity with sub-states or sub-activities. Set this option to apply history at the state or activity level.

History provides a mechanism to return to the most recently visited state (or activity) when transitioning directly to a state (or activity) with sub-states (sub-activities).

History applies to the level in which it appears. It may also be applied to the lowest depth of nested states (or activity).

**Sub state/activity history**

Specifies history for all depths for nested states or activities within the state or activity level. Set this option to apply history to all the depths of nested states or activities within the state or activity level.

History provides a mechanism to return to the most recently visited state (or activity) when transitioning directly to a state (or activity) with sub-states (sub-activities).

History applies to the level in which it appears. It may also be applied to the lowest depth of nested states (or activity).

## Activity Specification Dialog - Actions Tab

### Type

Displays the action specified in the **Action Specification** dialog. The following actions on activities can occur:

- **Entry** - The task is performed when the object enters the activity.
- **Exit** - The task is performed when the object exits the activity.
- **Do** - The task is performed while in the activity and must continue until exiting the activity.
- **Event** - The task triggers an event only when a specific event is received.

The type of action determines the options that are available in the dialog box.

Double-click on an action to open the **Action Specification** dialog for the selected action. If there are no actions listed, right-click on the **Actions** tab and click **Insert**.

### Action Expression

Displays the timing option that specifies when to carry out an action and the types of actions that are carried out.

## Activity Specification Dialog - Transitions Tab

### Event

Specifies the names of all the events for transitions associated with the activity.

An event causes a state transition. You do not have to uniquely label events because one event can cause a transition to many different states, activities, decisions, or synchronizations.

### End

Specifies the target state or activity for transitions.

## Activity Specification Dialog - Swimlanes Tab

### Name

Specifies the swimlane name where the enclosed activity resides.

## Activity Specification Dialog - Files Tab

### Filename

Displays a list of referenced files. You can insert and delete references to files or URLs by linking external files to model elements for documentation purposes.

### Path

Specifies the location of the file or URL.

## Actions

---

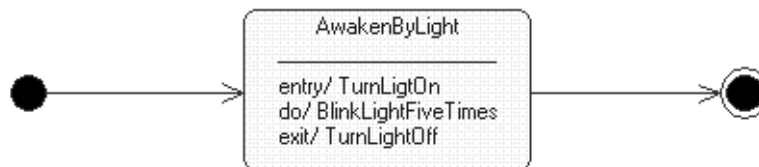
Each state and activity on an Activity Diagram can contain any number of internal actions. An action is a task that takes place while inside a state or activity.

Within a state or activity, there are four possible types for an action:

- on Entry
- on Exit
- do
- on Event

For example, Figure 64 shows that when entering the **AwakenByLight** state, the **entry** action (**On Entry**) turns on the light. Next, the **do** action changes the behavior of the state so that it blinks five times. Upon exiting the exits the **AwakenByLight** state, the **exit** action (**On Exit**) turns off the light.

**Figure 64 Example of Actions on a State**



## Action Specification Dialog

---

An **Action Specification** dialog enables you to display and modify the action properties in an Activity Diagram.

To open the **Action Specification** dialog, double-click on an action on the **Actions** tab.

The **Action Specification** dialog has the following tabs:

- **Detail**, see Action Specification Dialog - Detail Tab
- **Files**, see Action Specification Dialog - Files Tab

### To display the Action Specification dialog:

- 1 Open the **Specification** dialog for a state or activity.
- 2 Click the **Actions** tab.
- 3 If there are no actions, right-click and select **Insert**.
- 4 Double-click on an action to open its **Specification** dialog.

## Action Specification Dialog - Detail Tab

### When

Specifies a timing option to carry out for the selected action.

### On Event

The **On Event** parameters are available only when you set the **On Event** timing parameter in the **When** box.

- **Event** - In an **Activity Diagram**, an event is an occurrence that can trigger a state transition. Type the name of the event that will trigger the action.
- **Arguments** - Specifies any optional arguments associated with the event.
- **Condition** - Specifies a conditional **Boolean** expression.

You can use an **On Event** action rather than a self-transition because self-transitions trigger all the actions associated with a state, whereas state and activity actions handle internal state and activity transitions. This means that you can process an internal event without triggering the **entry** and **exit** actions.

## Type

Specifies the type for the action.

- **Action** - A simple action may be the invocation of a method, or the starting or stopping of an activity.
- **Send Event** - Send events are actions that trigger another event.

The type of action determines the options that are available in the dialog box.

## Name

Specifies a name of the **Action** or **Send Event**. This name appears on the state or activity on the Activity Diagram.

## Send arguments

Specifies any arguments for a send event. One or more arguments can accompany a send event.

## Send target

Specifies any targets for the send event. A target is any object that will receive the transition event.

## Action Specification Dialog - Files Tab

### Filename

Displays a list of referenced files. You can insert and delete references to files or URLs by linking external files to model elements for documentation purposes.

### Path

Specifies the location of the file or URL.

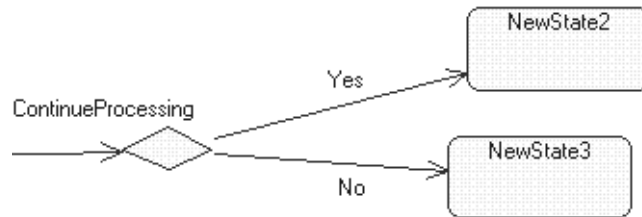
## Decisions

---

A decision represents a specific location on an Activity Diagram where the workflow may branch based upon guard conditions. There may be more than two outgoing transitions with different guard conditions, but for the most part, a decision has only two outgoing transitions determined by a Boolean expression.

Figure 65 shows an example of using a decision on an **Activity Diagram**.

**Figure 65 Graphical Representation of a Decision**



**Note:** Decisions only appear on **Activity Diagrams**; they do not appear on the **Model View** tab in the browser.

## Decision Specification Dialog

---

On the **Decision Specification** dialog, you can view and modify the properties and relationships of a decision on an **Activity Diagram**.

To view the **Decision Specification** dialog, double-click on a **Decision** icon on a State/Activity Diagram, or double-click on a **Decision** on the **Model View** tab in the browser.

**Note:** Decisions do not appear on the **Model View** tab in the browser.

The **Decision Specification** dialog has the following tabs:

- **General**, see Decision Specification Dialog - General Tab
- **Transitions**, see Decision Specification Dialog - Transitions Tab
- **Swimlanes**, see Decision Specification Dialog - Swimlanes Tab
- **Files**, see Decision Specification Dialog - Files Tab

### Decision Specification Dialog - General Tab

#### **Name**

Specifies the name for the selected decision.

#### **Owner**

Specifies the owner of the decision; the object that owns this element in the model.



## **Stereotype**

Specifies a keyword that further defines the classification of the model element. A stereotype represents the subclassification of a model element. Some stereotypes are already predefined, but you can also define your own to specify new modeling types.

To view stereotypes on the Activity Diagrams, click **Tools > Options**, select the **Diagram** tab, and click **Label**, **Decoration and label**, **Decoration only**, or **Icon** in the **Stereotype** box. **Label** displays the stereotype name in angle brackets (for example, <<stereotype>>). **Decoration** displays a graphic marker such as highlighting an icon or tool. **Icon** displays the graphical representation, if any.

## **Documentation**

Describes model elements or relationships. The description can include information such as the constraints, purpose, and essential behavior of the element.

## **Decision Specification Dialog - Transitions Tab**

### **Event**

Specifies the names of all the events for transitions associated with the decision.

An event causes a state transition. You do not have to uniquely label events because one event can cause a transition to many different states, activities, decisions, and synchronizations.

### **End**

Specifies the target state or activity for transitions.

## **Decision Specification Dialog - Swimlanes Tab**

### **Name**

Specifies the swimlane name where the enclosed decision resides.

## **Decision Specification Dialog - Files Tab**

### **Filename**

Displays a list of referenced files. You can insert and delete references to files or URLs by linking external files to model elements for documentation purposes.

### **Path**

Specifies the location of the file or URL.

## End State

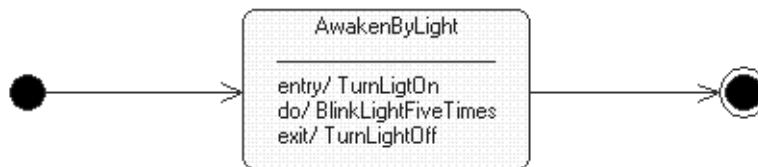
---

An end state represents a final or terminal state on an Activity Diagram. Add an end state when you want to explicitly show the end of a workflow on an Activity Diagram.

**Figure 66 End State**



**Figure 67 Example of an End State**



Transitions can only occur into an end state; however, there can be any number of end states or activities that transition to a single end state. You can also have multiple end states on an Activity Diagram.

For information on the **State Specification** dialog, see *State Specification Dialog* on page 277.

## Start State

---

A start state (also called an initial state) explicitly shows the beginning of a workflow or execution of states on an **Activity Diagram**. You can have only one start state for each **Activity Diagram** because each workflow/execution starts at the same location. If you use multiple **Activity Diagrams** to model a single element for different views, you can use the same start state as long as it is the same start state from the **Model View** tab in the browser. When you model nested states or nested activities, you can create one new start state for each context.

**Figure 68 Start State**



Typically, you can add only one outgoing transition from the start state. However, you can add multiple transitions on a start state if at least one of them is labeled with a condition. No incoming transitions are allowed for start states.

For information on the **State Specification** dialog, see *State Specification Dialog* on page 277.

## States

---

A state represents a condition or situation during the life of an object during which it satisfies some condition or waits for some event. Each state represents the cumulative history of its behavior.

Figure 69 shows a state for an **Activity Diagram**. The name of a state should be unique to its enclosing class, or if nested, within the state.

**Figure 69 Graphical Representation of a state**



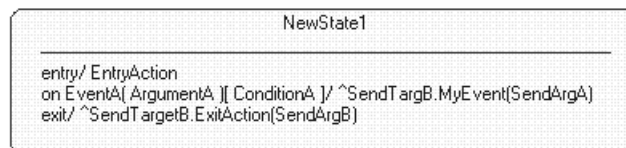
## Specifying Actions for States

Actions on states can occur:

- **on entry** - The task is performed when the object enters the state.
- **on exit** - The task is performed when the object exits the state.
- **do** - The task is performed while in the activity and must continue until exiting the state.
- **on event** - The task triggers an event only when a specific event is received.

Figure 70 shows a state with actions. You can only add actions for a state using the **Action Specification Dialog**.

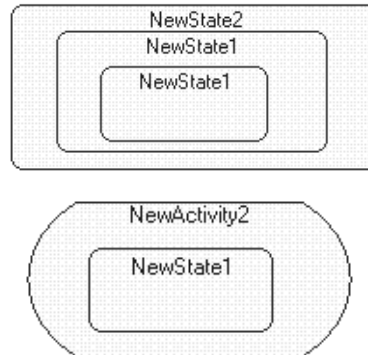
**Figure 70 Graphical Representation of a State with Actions**



## Nested States

You can nest states to any depth level. Everything that lies within the bounds of the nested state (sub-state) is referred to as its contents.

**Figure 71 Graphical Representations of Nested States**



## Manipulating Nested States

Nested states can be moved, resized, and transitioned to and from as if they were top-level activities. However, if you move a nested state outside a boundary of the super-state, the size of the container adjusts to accommodate the new position of the nested state.

You can "un-nest" a state by selecting the nested state, then dragging-and-dropping the state to an empty location on the **Activity Diagram**.

Nesting is determined by your mouse pointer location and states can overlap without nesting.

## Creating Nested States


You can create nested states by:

- Selecting the **State** icon from the Toolbox and clicking over the top of an existing state or activity.
- Use drag-and-drop to place a state over another state or activity.
- Right-click on an existing activity on the **Model View** tab in the browser, right-click, and click **New > State**.

The border of the target state becomes bold as the nested state moves over it.

**Note:** After you drop a state, the boundaries of the super-state enlarge to accommodate the nested element. If the mouse pointer is positioned over more than one state or activity at the same time, the element at the deepest level of nesting is considered the target. You can select multiple states and nest them as a group.

## State History

The history icon, , provides a mechanism to return to the most recently visited state or activity when transitioning directly to a state or activity with substates. History applies to the given level in which it appears. History may also be applied to the lowest depth of nested states. You may place an asterisk in the name field of the history state to designate the lowest depth.

To delete history icons, use **Cut** from the Toolbar or click **Edit > Delete**.

## State Specification Dialog

---

On the **State Specification** dialog, you can view and modify the properties and relationships of a state, start state, or end state on an **Activity Diagram**.

To view the **State Specification** dialog, double-click on a **State** icon on an **Activity Diagram**, or double-click on a state on the **Model View** tab in the browser.

The **State Specification** dialog has the following tabs:

- **General**, see State Specification Dialog - General Tab
- **Actions**, see State Specification Dialog - Actions Tab
- **Transitions**, see State Specification Dialog - Transitions Tab
- **Swimlanes**, see State Specification Dialog - Swimlanes Tab
- **Files**, see State Specification Dialog - Files Tab

### State Specification Dialog - General Tab

#### Name

Specifies the name for the currently selected state.

#### Stereotype

Specifies a keyword that further defines the classification of the state. A stereotype represents the subclassification of a model element. Some stereotypes are already predefined, but you can also define your own to specify new modeling types.

To view stereotypes on the Activity Diagrams, click **Tools > Options**, select the **Diagram** tab, and click **Label**, **Decoration and label**, **Decoration only**, or **Icon** in the **Stereotype** box. **Label** displays the stereotype name in angle brackets (for example, <<stereotype>>). **Decoration** displays a graphic marker such as highlighting an icon or tool. **Icon** displays the graphical representation, if any.

**Note:** Start States and End States do not have stereotypes.

### **Owner**

Specifies the model elements that own the selected state.

### **Context**

Specifies a view for a related set of modeling types.

### **Documentation**

Describes model elements or relationships. The description can include such information as the constraints, purpose, and essential behavior of the element. The information you type in this field is not displayed in the Activity Diagram.

### **State/activity history**

Specifies whether to return the most recently visited state or activity when transitioning directly to a state or activity with sub-states or sub-activities. Set this option to apply history at the state or activity level.

**Note:** Start States and End States do not have state or activity history.

History provides a mechanism to return to the most recently visited state (or activity) when transitioning directly to a state (or activity) with sub-states (sub-activities). History applies to the level in which it appears. It may also be applied to the lowest depth of nested states (or activity).

### **Sub state/activity history**

Specifies history for all depths for nested states or activities within the state or activity level. Set this option to apply history to all the depths of nested states or activities within the state or activity level.

**Note:** Start States and End States do not have sub-state history or sub-activity history.

History provides a mechanism to return to the most recently visited state (or activity) when transitioning directly to a state (or activity) with sub-states (sub-activities). History applies to the level in which it appears. It may also be applied to the lowest depth of nested states (or activity).

## State Specification Dialog - Actions Tab

### Type

Displays the action specified in the **Action Specification** dialog. Actions on states can occur:

- **Entry** - The task is performed when the object enters the state.
- **Exit** - The task is performed when the object exits the state.
- **Do** - The task is performed while in the activity and must continue until exiting the state.
- **Event** - The task triggers an event only when a specific event is received.

The type of action determines the options that are available in the dialog box.

Double-click on an action to open the **Action Specification** dialog for the selected action. If there are no actions listed, right-click on the **Actions** tab and click **Insert**.

### Action Expression

Displays the name of the corresponding timing option that specifies when to carry out an action for the selected state.

## State Specification Dialog - Transitions Tab

### Event

Specifies the names of all the events for transitions associated with the state.

An event causes a state transition. You do not have to uniquely label events because one event can cause a transition to many different states or activities.

It is possible for a state transition to have no associated event.

### End

Specifies the target state or activity for transitions.

## State Specification Dialog - Swimlanes Tab

### Name

Specifies the swimlane name where the enclosed state resides.

## State Specification Dialog - Files Tab

### Filename

Displays a list of referenced files. You can insert and delete references to files or URLs by linking external files to model elements for documentation purposes.

### Path

Specifies the location of the file or URL.

## Trigger Specification Dialog

---

Use the **Trigger Specification** dialog to view or modify the properties of a state transition.

To open the **Trigger Specification** dialog, double-click on a transition on the **Transitions** tab.

The **Trigger Specification** dialog has the following tabs:

- **Detail**, see *Trigger Specification Dialog - Detail Tab* on page 280
- **Files**, see *Trigger Specification Dialog - Files Tab* on page 281

## Trigger Specification Dialog - Detail Tab

### When

Specifies a timing option to carry out for the selected action.

### On Event

The **On Event** parameters are only enabled when you set the **On Event** timing parameter in the **When** box.

- **Event** - In an **Activity Diagram**, an event is an occurrence that can trigger a state transition. Type the name of the event that will trigger the action.
- **Arguments** - Specifies any optional arguments associated with the event.
- **Condition** - Specifies a conditional **Boolean** expression.

You can use an **On Event** action rather than a self-transition because self-transitions trigger all the actions associated with a state, whereas state and activity actions handle internal state and activity transitions. This means that you can process an internal event without triggering the **entry** and **exit** actions.



## Type

Specifies the type for the action.

- **Action** - A simple action may be the invocation of a method, or the starting or stopping of an activity.
- **Send Event** - Send events are actions that trigger another event.

The type of action determines what options are available in the dialog box.

## Name

Specifies a name of the **Action** or **Send Event**. This name appears on the state or activity on the Activity Diagram.

## Send arguments

Specifies any arguments for a send event. One or more arguments can accompany a send event.

## Send target

Specifies any targets for the send event. A target is any object that will receive the transition event.

## Trigger Specification Dialog - Files Tab

### Filename

Displays a list of referenced files. You can insert and delete references to files or URLs by linking external files to model elements for documentation purposes.

### Path

Specifies the location of the file or URL.

## Synchronizations

---

Synchronizations let you to see a simultaneous workflow in an Activity Diagram. Synchronizations visually define forks and joins representing a parallel workflow. For example, a synchronization can have a single incoming transition with multiple outgoing transitions, or have multiple incoming transitions with a single outgoing transition.

**Note:** Synchronizations appear as a horizontal or vertical bar on an Activity Diagram and may cross swimlanes. Synchronizations do not appear in the browser.

# Synchronization Specification Dialog

---

The **Synchronization Specification** dialog enables you to display and modify the properties and relationships of a synchronization on an **Activity Diagram**.

To view the **Synchronization Specification** dialog, select a **Synchronization** on an **Activity Diagram** and double-click.

The **Synchronization Specification** dialog has the following tabs:

- **General**, see Synchronization Specification Dialog - General Tab
- **Transitions**, see Synchronization Specification Dialog - Transitions Tab
- **Files**, see Synchronization Specification Dialog - Files Tab

## Synchronization Specification Dialog - General Tab

### Name

Specifies the name for the selected synchronization (vertical or horizontal).

### Owner

Specifies the owner of the synchronization; the object that owns this synchronization in the model.

### Stereotype

Specifies a keyword that further defines the classification of the model element. A stereotype represents the subclassification of a model element. Some stereotypes are already predefined, but you can also define your own to specify new modeling types.

To view stereotypes on the Activity Diagrams, click **Tools > Options**, select the **Diagram** tab, and click **Label**, **Decoration and label**, **Decoration only**, or **Icon** in the **Stereotype** box. **Label** displays the stereotype name in angle brackets (for example, <<stereotype>>). **Decoration** displays a graphic marker such as highlighting an icon or tool. **Icon** displays the graphical representation, if any.

### Documentation

Describes model elements or relationships. The description can include information such as the constraints, purpose, and essential behavior of the element.

## Synchronization Specification Dialog - Transitions Tab

### Event

Specifies the names of all the events for transitions associated with the selected synchronization.

An event causes a state transition. You do not have to uniquely label events because one event can cause a transition to many different states, activities, synchronizations, or decisions.

### End

Specifies the target state or activity for transitions.

## Synchronization Specification Dialog - Files Tab

### Filename

Displays a list of referenced files. You can insert and delete references to files or URLs by linking external files to model elements for documentation purposes.

### Path

Specifies the location of the file or URL.

## Transitions

---

A state transition indicates that an object in the source state will perform certain specified actions and enter the destination state when a specified event occurs, or when certain conditions are satisfied. A state transition is a relationship between two states, two activities, between an activity and a state, a self-transition, a synchronization, or a decision.

You can show one or more state transitions from a state as long as each transition is unique. Transitions originating from a state cannot have the same event, unless there are conditions on the event.

A state transition is a line with an arrowhead pointing toward the destination state or activity.

You should label each state transition with the name of at least one event that causes the state transition. You do not have to use unique labels for state transitions because the same event can cause a transition to many different states or activities.

Only one event is allowed per transition, and one action per event.

You can add events, conditions, and actions by using the **Transition Specification** dialog.

**Note:** Transitions do not appear on the **Model View** tab in the browser.

## Transition Specification Dialog

---

A **Transition Specification** dialog lets you to display and modify the properties and relationships of a transition on an Activity Diagram. The state transition specification lists the events and actions that comprise the transition.

To open the **Transition Specification** dialog, double-click on a transition line on an **Activity Diagram**.

The **Transition Specification** dialog has the following tabs:

- **General**, see Transition Specification Dialog - General Tab
- **Detail**, see Transition Specification Dialog - Detail Tab
- **Files**, see Transition Specification Dialog - Files Tab

### Transition Specification Dialog - General Tab

#### Event

Specifies the event that causes the state transition. You do not have to uniquely label events because one event can cause a transition to many different states or activities.

An event label is one of the following:

- Symbolic name
- Class name
- Name of an operation

It is possible for a state transition to have no associated event.

#### Arguments

Specifies any optional arguments associated with the transition. One or more arguments may accompany an event.

#### Stereotype

Specifies a keyword that further defines the classification of the model element. A stereotype represents the subclassification of a model element. Some stereotypes are already predefined, but you can also define your own to specify new modeling types.

To view stereotypes on the Activity Diagrams, click **Tools > Options**, select the **Diagram** tab, and click **Label**, **Decoration and label**, **Decoration only**, or **Icon** in the **Stereotype** box. **Label** displays the stereotype name in angle brackets (for example, <<stereotype>>). **Decoration** displays a graphic marker such as highlighting an icon or tool. **Icon** displays the graphical representation, if any.

### **Documentation**

Describes model elements or relationships. The description can include information such as the constraints, purpose, and essential behavior of the element.

## **Transition Specification Dialog - Detail Tab**

### **Guard Condition**

Conditional state transitions are triggered only when the conditional expression evaluates to true.

### **Action**

Shows the action that invokes a method, or starts and stops an activity on an Activity Diagram. An action shows what occurs upon entering or exiting the state. Actions can be messages to other objects, particularly when an Activity Diagram refers to an active class (one that drives other objects).

### **Send event**

Shows the send event for the selected transition. Event triggers can occur whenever an action has occurred. An event can contain a symbolic name, class name, or name of an operation. Event triggers are parsed into three components: **Send Event**, **Send Arguments**, and **Send Target**.

### **Send arguments**

Specifies any arguments for a send event. One or more arguments can accompany a send event.

### **Send target**

Specifies any targets for the send event. A target is any object that will receive the transition event.

### **Transition between substates**

Specifies transitions that occur between substates. Set this option when adding a transition to, or from, a sub-state or sub-activity that you want hidden from view. The drop-down list contains the name of all the states or activities that reside within the bounds of the top level superstate, including the superstate. The **From** box displays the state name that the transition is initiated from. The **To** field displays the state name that the transition is pointing to.

## **Transition Specification Dialog - Files Tab**

### **Filename**

Displays a list of referenced files. You can insert and delete references to files or URLs by linking external files to model elements for documentation purposes.

### **Path**

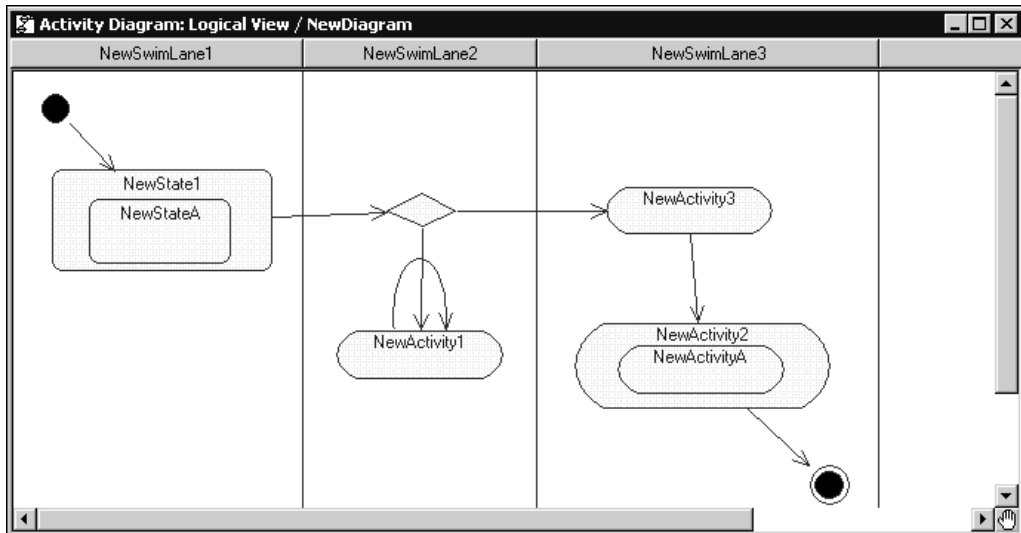
Specifies the location of the file or URL.

## **Swimlanes**

---

Swimlanes are partitions on an **Activity Diagram**. Swimlanes are helpful when modeling because they can represent organizational units or roles within a model. Swimlanes are very similar to an object because they provide a way to tell who is performing a certain role.

Figure 72 Activity Diagram with Swimlanes



You can add states and activities within swimlanes to determine which unit is responsible for carrying out the specific activity.

As you drag a swimlane onto an Activity Diagram, it becomes a swimlane view. Swimlanes appear as small icons in the browser, while swimlane views appear between vertical lines with a title that you can relocate and rename.

## Creating Swimlanes

You will want to create a swimlane to add partitions to your Activity Diagrams.

### To create a swimlane:

- 1 Create an **Activity Diagram**.
- 2 Click the **Swimlane** icon from the **Activity Diagram** Toolbox.
- 3 Click the pointer anywhere on the **Activity Diagram** to place the swimlane.

## Deleting a Swimlane

You can delete a swimlane from an Activity Diagram, or from the model. Deleting a swimlane from an Activity Diagram only deletes it from the diagram; the swimlane remains on the **Model View** tab in the browser. Performing a hard delete removes the swimlane from the entire model.

**Note:** The model elements residing within a swimlane are not deleted from the model or removed from the diagram when you delete a swimlane.

### **To delete a swimlane entirely from a model:**

- 1 On an Activity Diagram, select the header of the swimlane.
- 2 Press CTRL + D.

### **To delete a swimlane only from the Activity Diagram (not the entire model):**

- 1 Select a swimlane from an **Activity Diagram** by clicking on the swimlane header.
- 2 Press **Delete** or click **Edit > Delete**.

## **Moving a Swimlane**

You can easily change the order of the swimlanes on an Activity Diagram. When moving a swimlane, all the model elements within that swimlane, such as an Activity, State, or Decision, move to the new location.

**Note:** When moving a swimlane, all diagram elements, excluding synchronizations, are moved with the swimlane.

### **To relocate a swimlane on an Activity Diagram:**

- 1 Select the title of a swimlane.
- 2 Drag the header horizontally to the desired location.

## **Displaying Multiple Views of a Swimlane**

Since a swimlane can own other activities and states in different locations within the Activity Diagram, you may want to display multiple views of a swimlane. You can display multiple views of a swimlane by dragging the same swimlane from the **Model View** tab in the browser onto an Activity Diagram.

### **To display another view of a swimlane:**

- 1 On the **Model View** tab in the browser, select a swimlane.
- 2 Drag the swimlane from the browser and place it on an Activity Diagram.

A swimlane appears in the browser and the swimlane view appears on an Activity Diagram.



## Changing the Assignment of Responsibility of a Swimlane

When the assignment of responsibility for a swimlane changes, you can replace an existing swimlane on an Activity Diagram, with another swimlane.

**To change the assignment of responsibility for a swimlane, use the following steps:**

- 1 In the **Model View** tab in the browser, select a swimlane.
- 2 Drag the swimlane over an existing swimlane on an Activity Diagram.
- 3 Click on the diagram to place the swimlane.

The title of the swimlane changes to the newly assigned swimlane.

## Swimlane Specification Dialog

---

A **Swimlane Specification** dialog enables you to display and modify the properties and relationships of a swimlane.

To view the **Swimlane Specification** dialog:

- Select the swimlane title on an Activity Diagram and double-click
- Right-click on a swimlane on the **Model View** tab in the browser

The **Swimlane Specification** dialog has the following tabs:

- General, see Swimlane Specification Dialog - General Tab
- Files, see Swimlane Specification Dialog - Files Tab

### Swimlane Specification Dialog - General Tab

#### **Name**

Specifies the name of the currently selected swimlane.

#### **Class**

Specifies the name of the class the current swimlane is assigned. By default, the class is unspecified.

#### **Owner**

Specifies the model elements that owns the selected **Swimlane**.

### **Context**

Specifies a view for a related set of modeling types.

### **Documentation**

Describes model elements or relationships. The description can include information such as roles, keys, constraints, purpose, and essential behavior of the element.

## **Swimlane Specification Dialog - Files Tab**

### **Filename**

Displays a list of referenced files. You can insert and delete references to files or URLs by linking external files to model elements for documentation purposes.

### **Path**

Specifies the location of the file or URL.

## **Objects and Object Flows**

---

Typically, on **Activity Diagrams**, objects are model elements that represent something you can feel and touch. It might be helpful to think of objects as the nouns of the Activity Diagram and activities as the verbs of the activity diagram. Further, objects on Activity Diagrams allow you to represent the input and output relationships between activities. An object flow on an Activity Diagram represents the relationship between an activity and the object that creates it (as an output) or uses it (as an input).

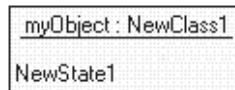
### **Objects**

An object has state, behavior, and identity. The structure and behavior of similar objects are defined in their common class. Each object in an **Activity Diagram** indicates some instance of a class. An object that is not named is referred to as a class instance.

If you use the same name for several object icons appearing in the same **Activity Diagram**, they are assumed to represent the same object; otherwise, each object icon represents a distinct object. Object icons appearing in different diagrams denote different objects, even when their names are identical.

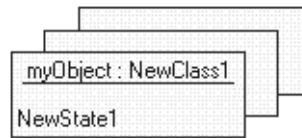
The **Object** icon is similar to a **Class** icon, except that the name is underlined:

**Figure 73 Object**



If you have multiple objects that are instances of the same class, you can modify the object icon by setting the **Multiple Instances** option on the **General** tab in the **Object Specification** dialog.

**Figure 74 Multiple Instances of an Object**



### **Concurrency**

An object's concurrency is defined by the concurrency of its class. You can display concurrency by right-clicking on an object and clicking **Show Concurrency**. The adornment appears at the bottom of the **Object** icon.

### **Persistence**

You can explicitly set the persistence of an object in the **Object Specification** dialog. You can display this value as an adornment by right-clicking on an Object and clicking **Show Persistence**. If you display both concurrency and persistence, the object's persistence appears after the concurrency.

## **Object State**

Most objects can appear in an infinite number of states. When you associate a new state with an object, a new state appears in the browser along with the object. You may specify more details of the object's state in the **State Specification** dialog.

## **Object Flow**

An object flow on an Activity Diagram represents the relationship between an activity and the object that creates it (as an output) or uses it (as an input).

In Rational Rose RealTime, object flows appear as dashed arrows rather than solid arrows to distinguish them from a typical transition. Object flows look identical to dependencies that appear on other diagram types.

## Object Flows and Transitions

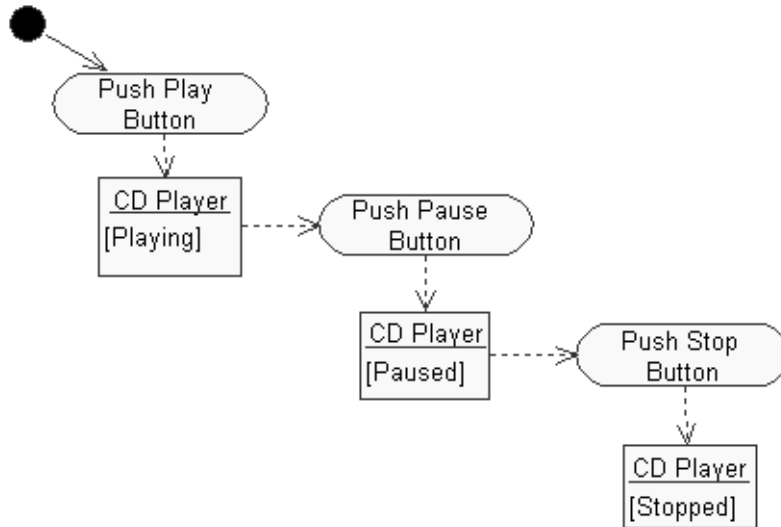
You do not need a transition if your diagram has two activities connected through an object and two corresponding object flows.

## Modeling Object State changes

The object flow sample demonstrates how activities affect object state on activity diagrams. The object flow sample illustrates three important aspects of activity diagram objects:

- Objects may appear more than once and in several states
- Activities may change object state
- Objects connect with activities through object flows

**Figure 75 Object Flow Example**



In the object flow sample, notice that the **CD Player** object appears on the diagram more than once. However, each object appears in a different state: **Playing**, **Paused**, and **Stopped**. Each activity changes the state of the **CD Player** when you push the various buttons or perform the appropriate activity. For example, when the activity **Push Pause Button** occurs, the state of the **CD Player** changes from **[Playing]** to **[Paused]**. In most cases, the same object may be (and usually is) the output of one activity, and the input of one or more subsequent activities.

## Creating an Object

You want to create objects on Activity Diagrams to allow you to represent the input and output relationships between activities.

### To create an object:

- 1 Open an **Activity Diagram**.
- 2 Click **Tools > Create > Object**.
- 3 Click on the **Activity Diagram** to place an object.

## Creating an Object Flow

Object flows appear as dashed arrows and are different from a typical transition (a solid line). You do not require a transition if your diagram currently has two activities connected through an object with two corresponding object flows.

### To create an object flow:

- 1 Open an **Activity Diagram**.
- 2 Click **Tools > Create > Object Flow**.
- 3 On the **Activity Diagram**, click on an object or activity to draw an object flow.

## Adding the Object, Object Flow, and Lock Selection Tools to the Toolbar

By default, the Toolbar for Activity diagrams does not include the **Object**, **Object Flow**, and **Lock Selections** Tools. You can easily add these tools by customizing the Toolbar.

### To add tools to the Toolbar for Activity Diagrams:

- 1 Right-click on the Toolbar for an **Activity Diagram**.
- 2 Click **Customize**.
- 3 In the **Customize Toolbar** dialog, select all desired tools from the list and click **Add**.
- 4 Click **OK**.

## Object Specification Dialog

---

An **Object Specification** dialog box lets you to display and modify the properties and relationships of an object on an **Activity Diagram**.

To view the **Object Specification** dialog, select an **Object** on an **Activity Diagram** and double-click.

The **Object Specification** dialog box has the following tabs:

- **General**, see *Object Specification Dialog - General Tab* on page 294
- **Incoming Object Flows**, see *Object Specification Dialog - Incoming Object Flows Tab* on page 296
- **Outgoing Object Flows**, see *Object Specification Dialog - Outgoing Object Flows Tab* on page 296
- **Files**, see *Object Specification Dialog - Files Tab* on page 296

### Object Specification Dialog - General Tab

#### Name

Specifies the name of the parent class for the class instance. The name must identify a class defined in the model.

#### Class

Specifies the name of the class that this object belongs. The default class for a newly created object is **(Unspecified)**.

If you delete a class from the model after you associated it with one or more objects, the class name is enclosed in parentheses. If you re-create the class or create a new class with the same name, the object becomes an instance of the new class.

#### State

Specifies the name for the object's state. The default state for a newly created object is **(Unspecified)**.

#### Stereotype

Specifies the name for an object stereotype.

A stereotype represents the subclassification of a model element. Some stereotypes are already predefined, but you can also define your own to specify new modeling types.

To view stereotypes on the Activity Diagrams, click **Tools > Options**, select the **Diagram** tab, and click **Label**, **Decoration and label**, **Decoration only**, or **Icon** in the **Stereotype** box. **Label** displays the stereotype name in angle brackets (for example, <<stereotype>>). **Decoration** displays a graphic marker such as highlighting an icon or tool. **Icon** displays the graphical representation, if any.

## Documentation

Specifies any information regarding the selected object, such as its purpose and any possible constraints.

## Persistence

Specifies the lifetime for the object's instances.

Persistence defines the lifetime of the instances of a class. A persistent element is expected to have a life span beyond that of the program, or one that is shared with other threads of control or other processes. Use this field to identify the persistence for elements of this class:

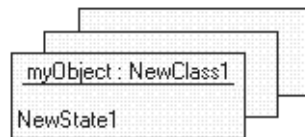
- **Persistent** - The state of the element exceeds the lifetime of the enclosing element.
- **Static** - The state of the element is fixed.
- **Transient** - The state and lifetime of the element are identical.

The persistence of an element must be compatible with the persistence you specified for its corresponding class. If a class persistence is set to **Persistent**, the object persistence is either persistent, static, or transient. If a class persistence is set to **Transient**, the object persistence is either static or transient.

## Multiple instances

Specifies that this object represents multiple instances of the same class. After you select this option, the icon for an object on the **Activity Diagram** changes from one object to three staggered objects.

**Figure 76 Multiple Instance of an Object**



This object group is considered one entity, but this icon indicates that several objects are involved.

**Note:** If an object is displayed as a stereotype, multiple instances are not graphically displayed.

## Object Specification Dialog - Incoming Object Flows Tab

### Name

Displays a list of **Object Flows** coming in to the selected **Object**.

## Object Specification Dialog - Outgoing Object Flows Tab

### Name

Displays a list of **Object Flows** going out of the selected **Object**.

## Object Specification Dialog - Files Tab

### Filename

Displays a list of referenced files. You can insert and delete references to files or URLs by linking external files to model elements for documentation purposes.

### Path

Specifies the location of the file or URL.

## Object Flow Specification Dialog

---

An **Object Flow Specification** dialog box lets you to display and modify the properties and relationships of an Object Flow on an **Activity Diagram**.

To view the **Object Flow Specification**, select an Object Flow on an **Activity Diagram** and double-click.

The **Object Flow Specification** has the following tabs:

- **General**, see *Object Specification Dialog - General Tab* on page 294
- **Files**, see *Object Flow Specification Dialog - Files Tab* on page 297



## Object Flow Specification Dialog - General Tab

### Name

Specifies the name of the currently selected **Object Flow**.

### Class

Specifies the name of the class that this **Object Flow** belongs.

### Stereotype

Specifies a keyword that further defines the classification of the **Object Flow**.

A stereotype represents the subclassification of a model element. Some stereotypes are already predefined but you can also define your own to specify new modeling types.

To view stereotypes on the Activity Diagrams, click **Tools > Options**, select the **Diagram** tab, and click **Label**, **Decoration and label**, **Decoration only**, or **Icon** in the **Stereotype** box. **Label** displays the stereotype name in angle brackets (for example, <<stereotype>>). **Decoration** displays a graphic marker such as highlighting an icon or tool. **Icon** displays the graphical representation, if any.

### Documentation

Describes information, such as purpose and essential behavior, for the **Object Flow**.

## Object Flow Specification Dialog - Files Tab

### Filename

Displays a list of referenced files. You can insert and delete references to files or URLs by linking external files to model elements for documentation purposes.

### Path

Specifies the location of the file or URL.

## Cutting Objects on Activity Diagrams

---

Use the **Cut** command from the **Edit** menu to remove the selected items from the current diagram and places them on the clipboard. From the clipboard, you can paste the item(s) on any related diagram. You can copy and then paste items from an **Activity Diagram** to another **Activity Diagram** in the same state machine, or in a different state machine.

When you cut an item, all relationships, such as transitions, for that item are also cut.

This command works only on the graphic representation of a diagram. It does not change the current model.

## Copying Objects on Activity Diagrams

---

Use the **Copy** command from the **Edit** menu to copy the selected items to the clipboard. You can copy and then paste items from an **Activity Diagram** to another **Activity Diagram** in the same state machine, or in a different state machine.

The **Copy** command works only on the graphic representation of a diagram. They do not change the current model.

## Pasting Objects on Activity Diagrams

---

Use the **Paste** command from the **Edit** menu to paste the items from the clipboard into the current diagram. From the Clipboard, you can paste items to another **Activity Diagram** in the same state machine, or in a different state machine.

The **Paste** command places previously cut or copied items into the current diagram. The pasted items appear in the center of the current view of the diagram.

## Contents

This chapter is organized as follows:

- *Creating a Sequence Diagram* on page 299
- *Cloning a Sequence Diagram* on page 302
- *Using Copy and Paste within Sequence Diagrams* on page 302
- *Using the Sequence Diagram Editor* on page 305
- *Sequence Diagram Toolbox* on page 307
- *Send Message Specification - Adding Ports to Capsule Classes* on page 315
- *Sequence Validation Dialog* on page 318
- *Focus of Control* on page 319

## Creating a Sequence Diagram

---

Sequence diagrams can be created in both the Use Case View and the Logical View. A Sequence diagram shows a particular interaction scenario among roles or instances in the model. A Sequence diagram is created from a collaboration diagram, which shows a general interaction pattern among roles or instances. (This includes capsule structure diagrams.) That is, the collaboration diagram shows the general pattern of associations among roles or instances, which is often created first and usually evolved in parallel with the associated Sequence diagrams. The Sequence diagram shows a specific sequence of interactions among roles or instances for a particular scenario.

Sequences can also be associated with protocols. Protocols, which are currently always binary, do not show their collaboration because it is fixed.

### Creating a New Diagram

There are four ways to create a new Sequence diagram: from the

- Model View Tab in the browser
- structure diagram browser
- structure or collaboration diagram
- trace window

## From the Browser

### To create a new Sequence diagram from the browser:

- 1 Select or create a collaboration diagram, capsule structure, protocol, package, use case, or class.
- 2 Right-click on the element in the model browser.
- 3 Select **New > Sequence Diagram** from the popup menu.
- 4 Enter the name of the Sequence diagram.

## From the Structure Diagram Browser

### To create a Sequence diagram from the structure diagram browser:

- 1 Right-click on the Sequence Diagrams folder in the Structure diagram browser.
- 2 Select Add New Sequence Diagram.

## From the Collaboration or Structure Diagram

### To create a new Sequence diagram from the collaboration or structure diagram:

- 1 Select or create a collaboration diagram, capsule structure or protocol.
- 2 Multi-select the model elements from the diagram to pre-populate the Sequence diagram.
- 3 Click in the diagram background and select **New > Sequence Diagram** from the popup menu.
- 4 Enter the name of the Sequence diagram.

## Editing a Diagram

### To edit a Sequence diagram:

- 1 Double-click on the diagram in the model browser.

The Sequence Diagram editor appears.

You can also select the **Open** menu item in the context menu for the Sequence diagram in the model browser.

- 2 Place capsules or class roles or instances in the diagram by dragging the class or capsule from the model browser, or by using the tools in the Sequence diagram toolbox.

- 3 Sequence diagrams are by default empty when created, except when created from protocols. Instances can be added to the Sequence diagram by dragging roles from the roles navigator within the structure browser. An instance representing the containing capsule class can also be added by dragging that class from the browser into the Sequence diagram.

A Sequence diagram can also be pre-populated with instances by selecting the desired set of roles in the structure or collaboration diagram and then selecting the **Create Sequence Diagram** menu item from the background menu of the diagram.

For structure diagrams, you can also optionally select the border if you want to show interactions between the capsule and its roles.

**Note:** There are no borders to select in collaboration diagrams.

- 4 Draw messages among instances using the toolbox.

## Adding Instances

The instances or roles in the Sequence diagram should generally be drawn from the instances or roles in the collaboration diagram. Collaboration and Sequence diagrams can show interactions among object instances or among roles. In most cases, they are more useful demonstrating interactions among roles, because a role demonstrates a part played in the scenario, which could be played by more than one instance.

Sequence diagrams are not automatically populated with instances. The instances must be added, either by dragging classes from the model browser or by using the instance tool from the toolbox.

Instances added using the instance tool are unspecified by default, which means that the tool does not know what actual design element the instance corresponds to. You can specify a role or create a new one using the drop-down model box that appears when you create a new instance. The Sequence diagram is not a complete specification until all instances are mapped to actual design elements. Use the Path field on the interaction instance specification dialog to specify which design instance the sequence instance maps to.

A Sequence diagram created under a protocol is pre-populated with two instances: base and conjugate. These instances cannot be removed and other instances cannot be added.

## Defining Messages

Messages are created between instances or between instances and the environment on the diagram to show interaction. Messages can represent: asynchronous sends, synchronous sends, function calls, instantiations, destructions, FOC (Focus of Control) blocks, local states, local actions, and coregions. There is a separate message tool for each of these.

## Specifying Message Details

Sequence diagrams act as design specifications. A complete Sequence diagram within a capsule structure can be verified by model execution. To verify a Sequence diagram, the sequence instances must be mapped to design instances, and the send messages among capsule instances must be specified. The specification of the message includes identifying the source and destination ports, signal names, and possibly data types.

## Cloning a Sequence Diagram

---

**To clone a Sequence diagram:**

- 1 In the browser, select the Sequence Diagram you want to clone.
- 2 Again in the browser, Control-drag it onto a Collaboration.

Note that you can select the same Collaboration that contains the original Sequence Diagram.

A new Sequence Diagram is created for you under this Collaboration.

## Using Copy and Paste within Sequence Diagrams

---

You can use **Copy** and **Paste** commands for elements within a single Sequence diagram, and to copy and paste from one Sequence Diagram to another. You can copy and paste the following Sequence diagram elements:

- Interaction instances
- Certain types of Messages (synchronous and asynchronous send messages, call messages, states, and actions)
- Standard diagram objects (constraints, notes, and text boxes)

## Interaction Instances

You can use the **Copy** and **Paste** commands on interaction instances and the properties associated with an interaction instance are preserved. You can select one or more interaction instances at the same time. When pasting interaction instances, they appear vertically at the top level and in the center of the visible area of the diagram.

If the diagram you are copying an interaction instance to (the destination diagram) resides under a collaboration that does not have classifier roles listed in the interaction instance, the roles on the pasted interaction instance will be reset.

**Note:** You cannot copy the Environment, or create and destroy messages.

## Messages

Messages are created between instances or between instances and the Environment on the diagram to show interaction. Messages can represent: asynchronous sends, synchronous sends, function calls, instantiations, destructions, FOC (Focus of Control) blocks, local states, local actions, and coregions.

You can use **Copy** and **Paste** commands on the following messages:

- Synchronous Send Message
- Asynchronous Send Message
- Call Messages
- Local Actions
- Local States

You cannot use **Copy** and **Paste** commands on the following elements:

- Focus of Control Blocks (FOCs)
- Return Messages
- Reply Messages
- Coregions
- Create message
- Destroy/Terminate Messages

You can copy and paste multiple messages at the same time, and those copied messages will preserve attributes of the original message when pasted.

If you have the **Auto-Create FOC's** option selected on the destination diagram:

- New FOCs are created on the pasted messages
- Return messages are created for pasted Call messages
- Reply messages are created for pasted synchronous send messages.

**Note:** You cannot paste existing FOC blocks and Return/Reply messages. To create new FOCs and Return/Reply messages, prior to using the **Copy** and **Paste** commands, right-click on the Sequence diagram and select **Auto-Create FOC's**. Auto-created FOCs and Return/Reply messages are not copies of the original FOCs and Return/Reply messages. Copying creates new FOCs and Return/Reply messages.

### **Message Positioning in the Destination Diagram**

Pasted message appear after all existing messages and FOCs on the interaction instances, but before any terminate and destroy messages if any exist for the interaction instance.

**Note:** When pasting messages, the display on the diagram in the order they were selected for when copied. When pasting several messages at the same time, positioning applies to messages one-by-one, possibly creating a vertical separation between messages that did not have any on the originating diagram.

### **Pasting Messages into Originating Diagram**

If the destination diagram is the same as the source diagram, a pasted message appears between the same interaction instances as the original message.

### **Pasting Messages into Another Diagram**

If the destination diagram is different from the source diagram, a pasted message appears between interaction instances that have the same internal numbers as the interaction instances. Internal numbers do not necessarily correspond to the visual order of interaction instances on a Sequence diagram.

If no interaction instance exists in the destination diagram (excluding the Environment itself), a new "Unspecified" interaction instance is created before the messages are pasted on to the diagram.



## Standard Diagram Elements

The following standard diagram elements will support copy/paste on sequence diagrams:

- Constraints
- Notes
- Text boxes

**Note:** You cannot copy Anchors on Sequence diagrams.

### Enabling Standard Functionality

**Cut** and **Duplicate** commands are available for the elements that support the **Copy** and **Paste** commands.

### Automatic Scrolling

If pasted elements appear outside of the visible area, the Sequence diagram scrolls so that visible area for the pasted element is centered.

## Known Limitations

Pasting messages on an Interaction Instance that has limited space between the next-to-last message and the instance's terminate/destroy message may create a pasted message overlapping either the next-to-last message or the terminate/destroy message. You must manually adjust the placement of pasted messages.

You cannot copy model elements from a Sequence diagram if that diagram was created from a trace with **Don't save with the model** selected.

## Using the Sequence Diagram Editor

---

A Sequence diagram is a graphical view of a scenario that shows an object interaction in a time-based sequence. Sequence diagrams establish the roles of objects and help provide essential information to determine class responsibilities and interfaces.

A Sequence diagram has two dimensions: vertical placement represents time and horizontal placement represents different objects.

Elements of the Sequence diagram, such as instances and messages are added using the toolbox.

The window title bar shows the full name of the Sequence diagram.

You can access the following Specification dialogs from elements on the Sequence diagram:

- Instance Specification dialog
- Interaction Specification dialog
- Send Message Specification dialog
- Call Message Specification dialog
- Create Message Specification dialog
- Return Message Specification dialog
- Destroy Message Specification dialog
- Local State Specification dialog
- Coreion Specification dialog
- Local Action Specification dialog
- Reply Message Specification dialog

The popup menu for the Sequence diagram editor includes a validate command that opens the validation dialog. It also contains an **Auto-generate FOC** entry, which controls whether new send or call messages automatically get an FOC (Focus of Control) - and a return message, if appropriate - when they are created.

**Note:** If you are not interested in message activation, turn the **Auto-generate FOC** entry off. This simplifies the diagram display considerably.

## Opening Collaboration Diagrams

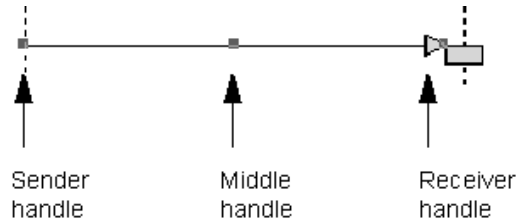
To open the Collaboration diagram associated with a particular Sequence diagram, select **Open Collaboration Diagram** from the popup menu for the background of the Sequence diagram.

## Reorienting Messages

You can reorient messages to make semantic changes in the diagram, for example, to change the sender or receiver for a message.

Using the sender or receiver handles, you can change the sender or receiver. Using the Re-order handle (Middle handle), you can change the order of this message on the sender and receiver.

**Figure 77 Message Handles**



## **Moving Messages**

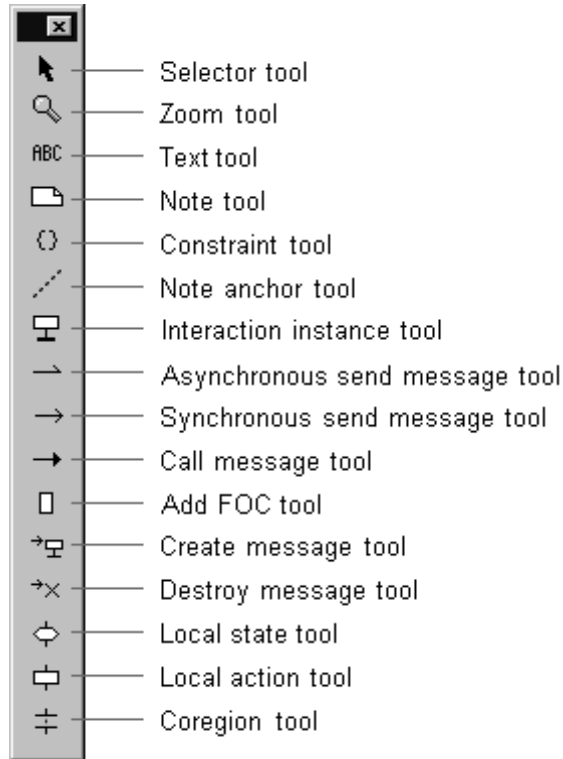
You can move messages in a Sequence diagram to create more or less space between them. Message movement is restricted to avoid accidentally altering the semantics of the diagram. When moving a message, do not drag it using one of the handles.

## **Sequence Diagram Toolbox**

---

The Sequence diagram toolbox contains tools for adding elements to the Sequence diagram.

**Figure 78 Sequence Diagram Toolbox**



### **Selector Tool**

Select objects for moving, resizing, and so forth.

### **Zoom Tool**

Zooms in on a portion of the diagram. Click on the **Zoom** tool and then click on the area of the diagram to zoom in on.

### **Text Tool**

Adds text anywhere in the diagram.

### **Note Tool**

Annotates the diagram with textual notes. This is useful for marking up the diagram with explanations, review comments, and so forth. You can drag and drop a diagram or external document from the browser onto a note. Notice that the name of the diagram or external document is underlined. If you double-click on the note, the diagram or external document is opened. You can undo and redo this command.

### **Constraint Tool**

Adds UML constraints to the diagram. A constraint can be anchored to a view element by using the anchor tool. Currently, constraints do not have any semantic meaning to the tool.

### **Note Anchor Tool**

Anchors a note or constraint to a particular element on the diagram.

### **Interaction Instance Tool**

Adds an object instance to the diagram. Instances are added along the horizontal axis at the top of the diagram.

### **Synchronous Send Message Tool**

Adds a message between two instances. Synchronous send messages block the sender waiting for a return (like a function call). This corresponds to the 'invoke' operation.

### **Asynchronous Send Message Tool**

Adds a message between two instances. Asynchronous messages do not block the sender.

### **Call Message**

Adds a message between two instances. Call messages are like function calls, so the sender is blocked waiting for a return.

### **Add FOC**

Adds an FOC (Focus of Control) block to the selected message. Select the tool and click on the send or call message you want to add an FOC to.

**Note:** If the selected message can have a reply or return (that is, a synchronous send or call), it is automatically generated.

### **Create Message Tool**

Indicates that one instance creates another instance dynamically. The create message indicates the moment of creation of the destination instance. The new instance opens at the end of the create message.

### **Destroy Message Tool**

Indicates that one instance destroys (deletes) another instance dynamically. The destroy message indicates the moment of destruction of the destination instance.

### **Local State Tool**

Indicates a state change in one of the instances. Click on one of the instances in the diagram to place a new state at the selected location.

### **Local Action Tool**

Indicates an action carried out by one of the instances. The action represents a significant activity or operation being performed by the instance at that time.

### **Coregion Tool**

Indicates a set of events/messages whose ordering is undefined. That is, although the messages appear in a particular order (as indicated by their vertical placement on the instance line), the actual run-time ordering may vary.

## **Interaction Instance Specification**

The interaction instance specification has information about an instance on a Sequence diagram.

It contains two tabs: **General** and **Files**.

### **General Tab**

#### **Name**

Specifies the name of this instance. Instances are unnamed by default. The name is displayed as part of the instance label on the diagram.

#### **Path**

Identifies the role path for an instance in a collaboration. The drop-down menu allows you to choose from the available roles in the immediate collaboration associated with the Sequence diagram.

For Sequence diagrams under structure diagrams, it is also possible to show interactions between the capsule and its roles. To do this, pick the capsule from the path pull-down menu.

**Note:** If the **Path** box contains a role, the corresponding label becomes a hot link to the Specification dialog for that classifier role.

### **Stereotype**

Specifies the (optional) stereotype of this instance.

### **Documentation**

Use the Documentation field to describe this instance.

### **Files Tab**

The **Files** tab allows for linking external files.

## **Interaction Specification**

The Interaction Specification is used to describe interactions on a Sequence diagram.

It has two tabs: General and Files

### **General Tab**

#### **Name**

Specifies the name of the interaction.

#### **Stereotype**

Specifies the (optional) stereotype of this instance.

#### **Documentation**

Describes this interaction.

### **Files Tab**

The **Files** tab allows for linking external files.

## Local Action Specification

**The Local Action Specification is used to describe actions in Sequence diagrams.**

It contains General, Detail, and Files tabs.

### General Tab

#### Name

A name for the local action, which is displayed on the Sequence diagram.

#### Stereotype

Specifies a stereotype for the local action.

### Detail Tab

#### Sender

Non-editable field with the name of the instance where the local action is defined.

#### Receiver

Not applicable to a local action.

#### Time

Capture the time of the action.

#### Effect

A textual description of the effect of the local action.

## Local State Specification

The Local State Specification contains General, Detail, and File tabs.

### General Tab

#### Name

A name for the local state. The name is displayed on the Sequence diagram.



**Stereotype**

Specify a stereotype for the local state.

**Detail Tab****Sender**

Non-editable field with the name of the instance where the local state is defined.

**Receiver**

Not applicable for the local state

**Time**

Capture the time of the state change.

**Message Specification**

There are several different kinds of messages, but all have similar controls in the Message Specification dialog.

The Message Specification dialog contains the following tabs: General, Detail, Port Detail (only for Send messages), and Files.

**General Tab****Name**

A name for the message. The name is displayed on the Sequence diagram.

**Stereotype**

Specify a stereotype for the message.

**Documentation**

Specify documentation for this element.

**Detail Tab****Sender**

Non-editable field with the name of the instance where the message originated.

## **Receiver**

Non-editable field with the name of the instance where the message ends.

## **Time**

Capture the time that the message was sent.

## **Data**

A textual description of the message data.

## **Port Detail Tab**

This tab is only significant for messages between capsule roles. The data on this tab can be filled in by selecting from the pull-down menus, which include data from the collaboration diagram.

If the fields on this tab are filled in for a Sequence diagram that acts as a behavior specification, then the data can be compared to the actual data captured from a run-time execution trace to verify the behavior at execution time against the specification.

## **From Port**

The name of the port on the sender capsule.

**Note:** If the **From Port** box contains a port, the corresponding label becomes a hot link to the Specification dialog for that port.

## **To Port**

The name of the port on the receiver capsule.

**Note:** If the **To Port** box contains a port, the corresponding label becomes a hot link to the Specification dialog for that port.

## Signal

The name of the signal from the ports' protocol.

**Note:** If the **Signal** box contains a signal, the corresponding label becomes a hot link to the Specification dialog for that signal.

## Delivered

Capture the time the message was delivered to the receiver.

## Priority

The priority at which the message is sent. (Applies only to an Asynchronous Send Message.)

# Send Message Specification - Adding Ports to Capsule Classes

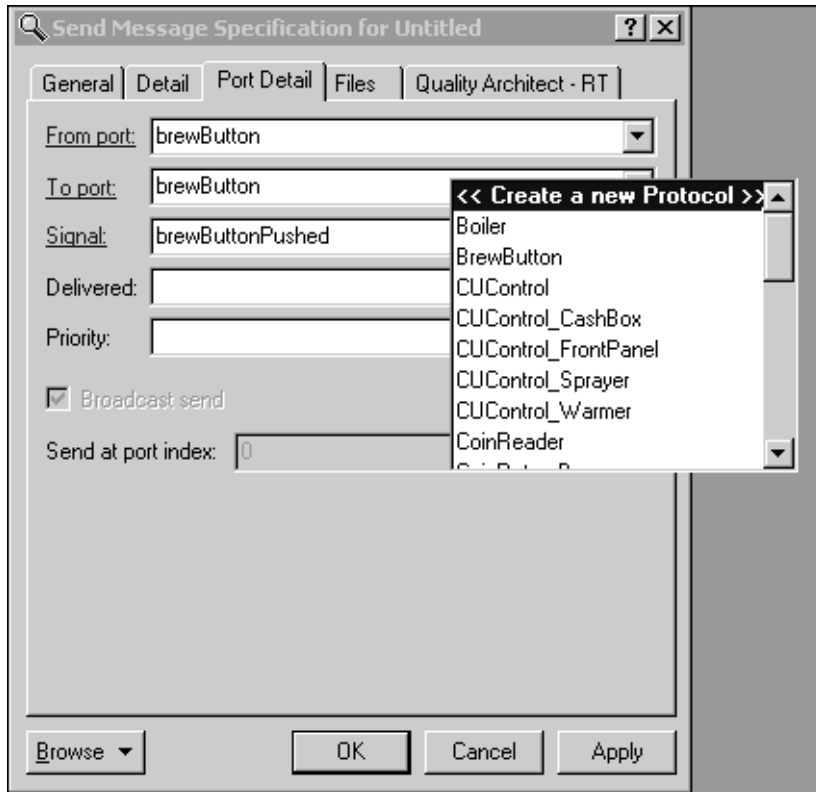
---

The drop-down list for the **From port** (see Figure 80) and **To port** (see Figure 81) boxes in the **Port Detail** tab in the **Send Message Specification** dialog box have a new item called <<Create a new Port>>.

The <<Create a new Port>> option allows you to:

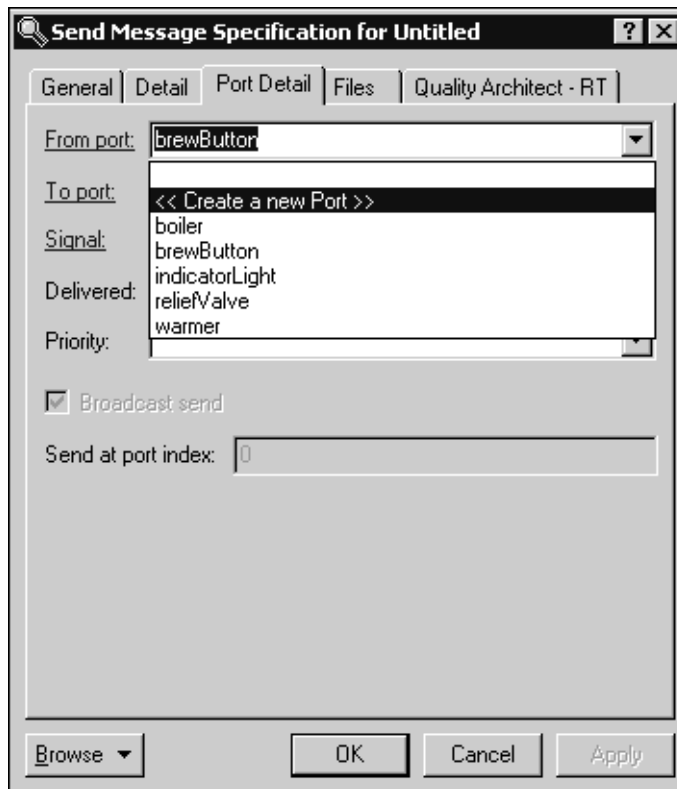
- create a new protocol if the model does not currently contain any
- select a existing protocol

**Figure 79 Send Message Specification Dialog Box**



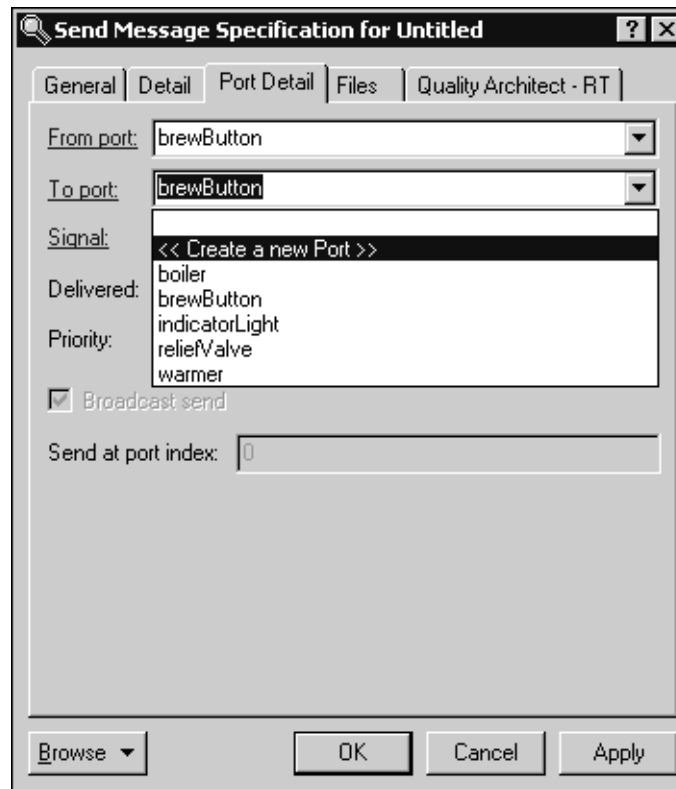
**Note:** After you select <<Create a new Port>>, the <<Create a new Protocol>> list appears. To cancel, click ESC.

Figure 80 Send Message Specification Dialog - From Port



In the **From port** box, the <<Create a new Port>> item appears in the list only when the sender is a capsule.

Figure 81 Send Message Specification Dialog Box- To Port



In the **To port** box, the **<<Create a new Port>>** item appears in the list only when the receiver is a capsule.

## Sequence Validation Dialog

---

A Sequence diagram can be used as a specification of the interaction among object roles and/or instances. Sequence diagrams are very useful as specifications of design intent to be checked against actual model execution results.

The **Sequence Diagram Validation Dialog** allows you to check the Sequence diagram specification for missing elements. It provides control over what aspects of the Sequence diagram should be checked for completeness.

The options to control what to check during verification are:

- **Instance** - Checks that the path of each instance in the interaction is defined.
- **Sender port** - Verifies that the sender port names in the sequence are defined and, possibly, resolved to an existing port.
- **Receiver port** - Verifies that the receiver port names in the sequence are defined and, possibly, resolved to an existing port.
- **Signal/Operation** - Verifies that the signal names for send or operation names for a call are defined and, possibly, resolved to an existing signal.
- **Data** - Verifies that the data types of all messages in the sequence are defined.
- **Validate** - Performs the validation. Results appear in the **Error** log.

### **Validation Error Log**

Contains the results of the validation. Each item in the list indicates an undefined or unresolved sequence element.

## **Focus of Control**

---

Focus of Control (FOC) is an advanced notational technique that enhances Sequence diagrams. This technique shows the period of time during which an object is performing an action, either directly or through an underlying procedure.

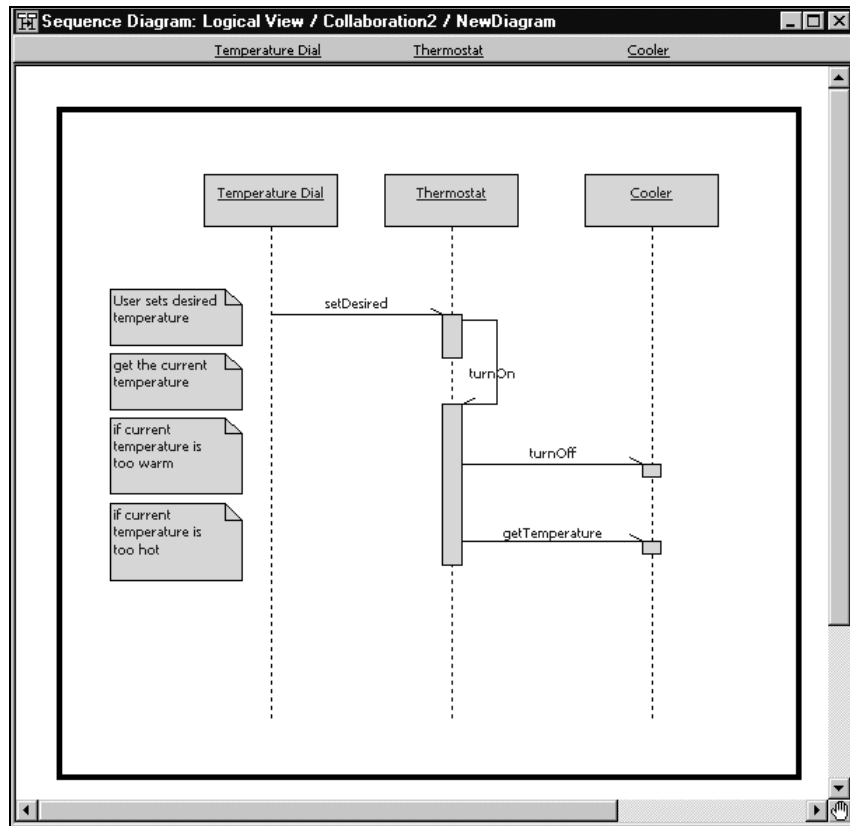
FOC is portrayed through narrow rectangles that adorn lifelines (the vertical lines descending from each object). The length of a FOC indicates the amount of time it takes for a message to be performed. When you move a message vertically, each dependent message moves vertically as well. Also, you can reorient a message vertically off the source FOC to make it detached and independent.

### **Activators**

Messages that originate from an FOC are said to have been activated by the message that started that FOC.

A Sequence diagram with FOC notation and scripts follows:

**Figure 82 Focus of Control Diagram Example**



## Coloring a Focus of Control

To help distinguish a particular FOC from other items in a Sequence diagram, you can fill a FOC with a color.

### To color a FOC:

- 1 Select the FOC you want to color.
- 2 Click Diagram Object Properties from the **Edit** menu and then click **Fill Color**.
- 3 Click on the color you want to make the selected FOC.
- 4 Click **OK**.



## Navigating Sequence Diagrams

---

You can easily navigate a sequence diagram by using the arrow keys to move the selection from one view to another. This type of navigation provides you with an integrated view of the environment.

The only views supported for navigation are:

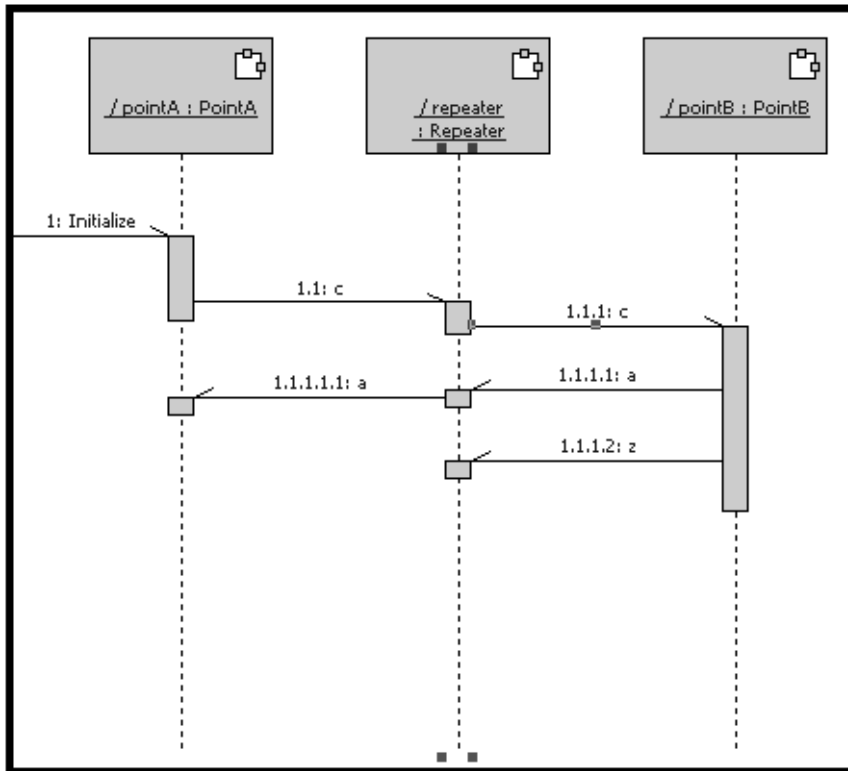
- local actions
- local states
- all message types

**Note:** Focus of control blocks, interaction instance views (the rectangle at the top of the lifeline), lifeline views, and coregions do not support this type of navigation.

### Selecting a Current Lifeline

The current lifeline is set implicitly the first time that you start to navigate, or by selecting an element using the mouse (an element not connected to the current lifeline), and then starting to navigate from that selection. For example, you can select the message **1.1:c** in Figure 85 and press CTRL and the down arrow. This action makes the lifeline extending from **repeater** the current lifeline.

**Figure 83** Current Lifeline Selection

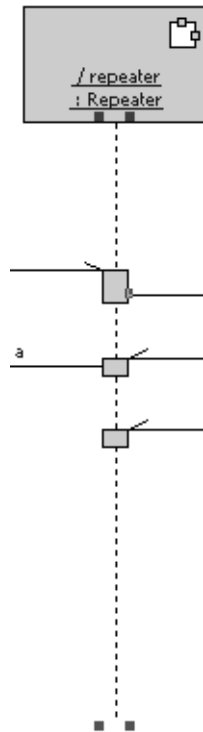


**Note:** The up and down arrow keys move the selection to the next selectable element on the current lifeline (the vertical lines descending from each object).

You can identify the current lifeline by the selection marks on the lifeline (Figure 84). If none of the lifelines in the sequence diagram have selection marks, the initial navigation was not successful.

Once you select a current lifeline, you can make another lifeline current using the left and right arrow keys to move horizontally to the nearest lifeline. When following a message, such as moving to the right from the left end of a message between adjacent lifelines, the current lifeline will change, and the selected message remains selected. If the nearest lifeline in the appropriate direction is selected, the message on that lifeline nearest the intersection point will be selected.

**Figure 84 Lifeline Selection**

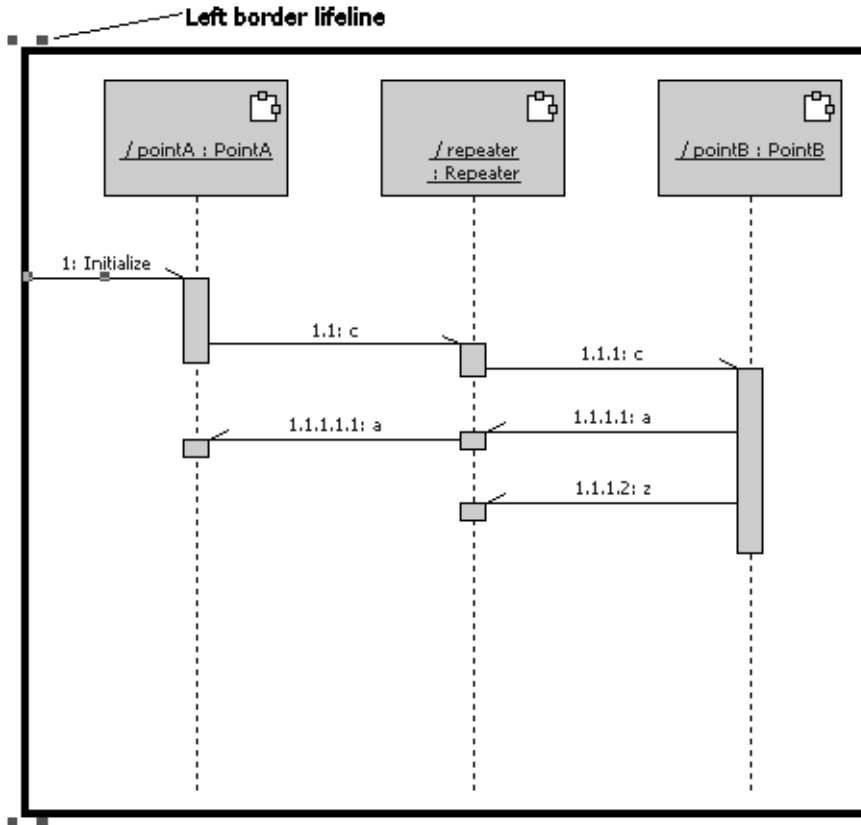


After you select a current lifeline, you can make another lifeline current using the left and right arrow keys to move horizontally to the nearest lifeline. When following a message, such as moving to the right from the left end of a message between adjacent lifelines, the current lifeline will change, and the selected message remains selected. If the nearest lifeline in the appropriate direction is selected, the message on that lifeline nearest the intersection point will be selected.

By default, when you first select a message and attempt to navigate from it, the current lifeline is set to the left end of the message (regardless of the message orientation). Press CTRL and an arrow key to select the right end. When you establish the current lifeline, pressing CTRL while using the arrow key toggles the current lifeline from one end of the message to the other.

The environment in a sequence diagram has two visual lifelines at the left and right borders of the diagram (see Figure 85). However, these border lifelines are modeled by one interaction instance and each represent a single timeline.

Figure 85 Selected Lifeline for Left Border



**Note:** If you set the left or right border as the current lifeline, selecting the up or down arrow selects the next element in the integrated view. This may result in the view moving from the left border to the right border, or vice versa. This type of navigation sequence illustrates the actual timeline of events for the environment.

## Saving Sequence Diagrams as Controlled Units

You can save Sequence Diagrams as controlled units. A sequence diagram is owned by a Collaboration diagram (an interaction). The controllable unit will be the interaction and has the file extension `.rtintractn`. Sequence Diagrams are owned by Collaborations. This includes the special case of the Capsule Collaboration which owns the Capsule Structure Diagram. Collaborations are only controllable if they are

directly owned by a package. They are not controllable if they are owned by a capsule or class. Sequence Diagrams are controllable only if they are owned by a controllable Collaboration.

**Note:** After saving a controlled Collaboration with the Rational Rose RealTime toolset, you will not be able to load the petal file with any previous version of the toolset, even if you have no controlled Sequence Diagrams.

For controllable Collaboration diagrams, you can use the two new context menu entries, **Control Child Units** and **Uncontrol Child Units**.

**Control Child Units** is enabled when there is at least one uncontrolled child Sequence Diagram. **Uncontrol Child Units** is enabled when there is at least one controlled child Sequence Diagram.

There are two new options on the **Unit Information** tab for controlled interactions: **Control new child units** and **Disallow model-relative pathnames**.

### **Owned by model**

Indicates whether the unit is owned by this collaboration, or whether it is owned by another model and shared into this model. When selected, it indicates that this collaboration is owned by another model.

### **Under source control**

Indicates whether this element has been added to source control.

### **Control new child units**

Controls whether newly created sequence diagrams in this collaboration will be individually controlled, by default. When selected, any new child units (sequence diagrams) for this element will be controlled.

### **Disallow model-relative pathnames**

Informs Rational Rose RealTime not to use the implicit and virtual pathmap symbol when saving units.

### **Filename**

Specifies the fully-qualified file name for the collaboration.

### **Version**

Indicates the version information as defined by your configuration management application.

## Uncontrolling Sequence Diagrams

You can uncontrol sequence diagrams by:

- Using the context menus directly
- Controlling and then uncontrolling an ancestor unit such as the parent collaboration, and the owning package of the collaboration

## Importing and Exporting Sequence Diagrams

You cannot export sequence diagrams (or collaborations) directly; however, you can export the containing package.

### RRTEI

The RRTEI allows the new attributes for Interactions and Collaborations to be manipulated through the methods exposed by ControllableElement. To automate the process of making sequence diagrams controllable units for existing models, you can use, and modify, the Control Interaction scripts in the following location:

`$ROSERT_HOME/Scripts/ControlInteractions`

We recommend that you make a copy of these scripts before modifying them.

## Control Interaction Scripts

There are three SummitBasic scripts included with this patch. The purpose of these Control Interaction scripts is to automate the process of making sequence diagrams controllable units for existing models. This process is divided into three scripts to provide you with the opportunity to take some intermediate actions at various stages within the conversion process.

After you install this patch, you will find the Control Interaction scripts in the following location:

`$ROSERT_HOME/Scripts/ControlInteractions`

The Control Interaction scripts are:

- `ControlInteractions_CheckOut.ebs`
- `ControlInteractions_AddSequenceDiagrams.ebs`
- `ControlInteractions_CheckIn.ebs`

## **ControlInteractions\_CheckOut.ebs**

This script is the first of three scripts used to make Sequence Diagrams controllable. This script searches the currently open model file for Collaborations that are a child of a Package. The following activities occur for each Collaboration:

- If it is not currently controlled, the Collaboration will be controlled.
- If not currently under source control, the Collaboration is added to source control.
- The Collaboration file (.rtcollab) is checked out.
- The **Control New Child Units** property is set to TRUE.
- A child directory associated with the Collaboration is created and added to source control.
- If required, the containing parent package for a Collaboration is also checked out.

### **Before Starting This Script:**

- The model must be under source control.
- All packages containing a Collaboration that will be modified must already be under source control.
- Rational Rose RealTime must be at patch level 6.4.353 or higher.

## **ControlInteractions\_AddSequenceDiagrams.ebs**

This is the second of three scripts used to make Sequence Diagrams controllable. This script searches the model for Collaborations that are checked out and are a child of a package. For each Collaboration, its contained Sequence Diagrams that are not controlled are made controllable, and may be added to source control, if required.

### **Before Starting This Script:**

- The model must be under source control.
- The first script, ControlInteractions\_CheckOut.ebs, ran and it completed successfully.
- Rational Rose RealTime must be at patch level 6.4.353 or higher.

## **ControlInteractions\_CheckIn.ebs**

This is the third of three scripts used to make some Sequence Diagrams controllable. This script searches the model for Collaborations that are checked out and are children of a Package. Each Collaboration will be checked in, and if necessary, its parent package will be checked in.

### **Before Starting This Script:**

- The model must be under source control.
- The first two scripts, `ControlInteractions_CheckOut.ebs` and `ControlInteractions_AddSequenceDiagrams.ebs`, were run and they completed successfully.
- Rational Rose RealTime must be at patch level 6.4.353 or higher.

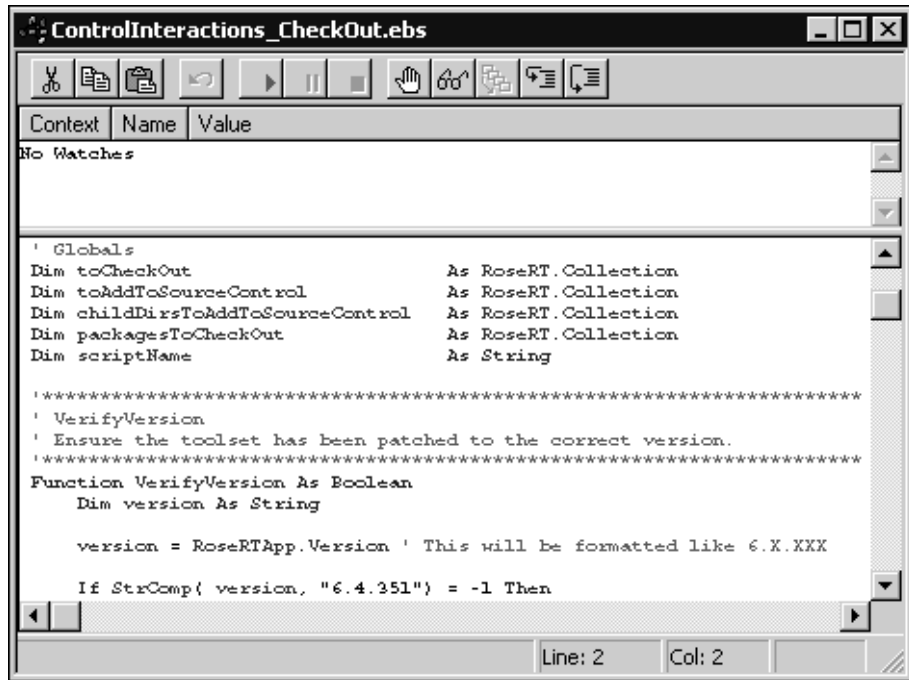
## **Running Scripts to Make Sequence Diagrams Controllable**


### **To make Sequence Diagrams controllable:**

- 1 Ensure that Rational Rose RealTime is at patch level 6.4.353 or higher.
- 2 Start Rational Rose RealTime.
- 3 Open your model.
- 4 Ensure that the model is under source control.
- 5 Click **Tools > Open Script**.
- 6 Browse to the following directory:  
\$ROSERT\_HOME/Scripts/ControlInteractions

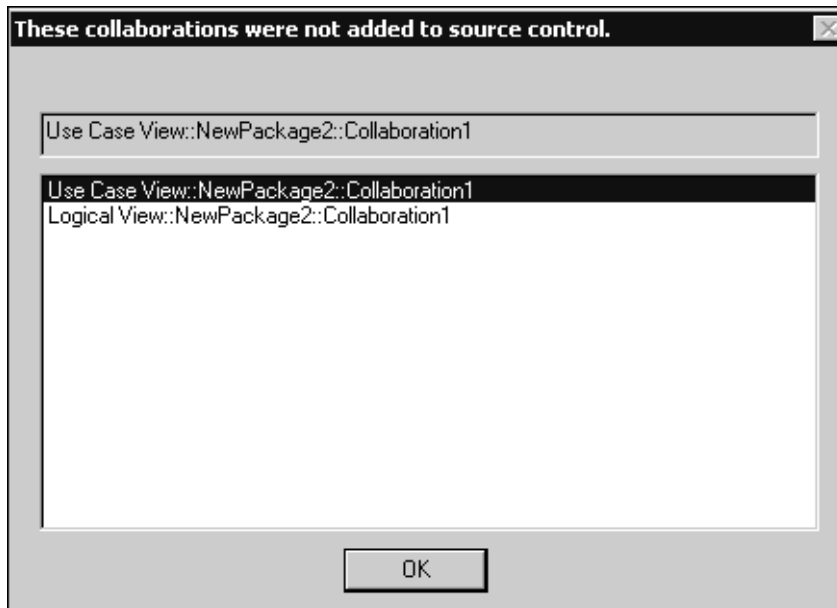


- 7 Select ControlInteractions\_CheckOut.ebs.



- 8 Click Start (  ) to execute the first script.

If this script determines that no changes are required, a dialog lists the Collaborations that were not added to source control, and the ControlInteractions\_CheckOut.ebs script will terminate.



**9** Click **OK**.

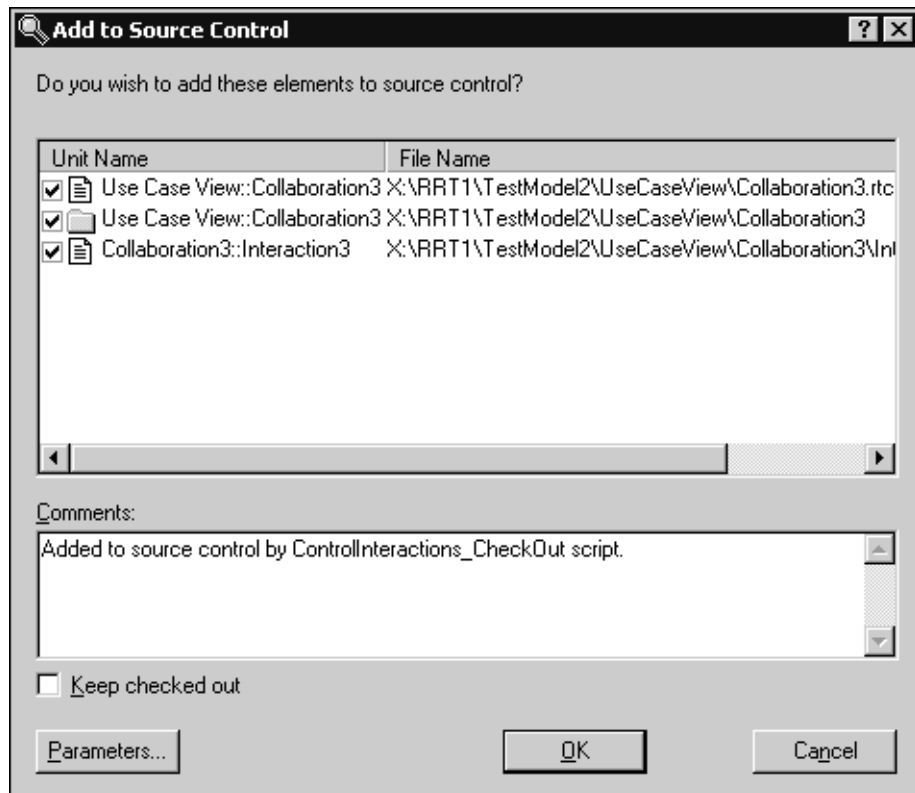
If one or more Collaborations need to be controlled, you are prompted for permission to check out the parent package for the Collaboration.

You must check out the parent package for the Collaboration because the definition of an uncontrolled collaboration is part of the parent package. This definition must be removed from the parent package and put into its own controllable unit.

**10** Click **Yes** to permit the checkout of the required packages.

**11** In the **Checkout** dialog, select the packages to check out. If you did not allow permission to check out the packages, the affected collaborations are ignored. After the checkout completes, the uncontrolled collaborations are made controllable.

If the script detects that some collaborations need to be added to source control, you are prompted for permission. If you click **Yes**, the **Add To Source Control** dialog appears.



**12** Select the desired collaboration to add to source control.

**13** Click **OK**.

Next, the script attempts to check out those collaborations that require changes.

The **CheckOut** dialog opens.

**14** Select the collaborations to check out.

**15** If the script detects that there are collaborations under source control that do not have a child directory which is also under source control, it will prompt you to add child directories to source control. Click **Yes** to open the **Add To Source Control** dialog to select the directories to add to source control.


**16** Optional. You can make manual changes to the script, if required. For example, if a collaboration was checked out and you do not want it modified, you can click **Undo checkout** for the collaboration, or if a collaboration you want modified is not checked out, you can manually check it out.

**17** Click **Tools > Open Script**.

**18** Browse to the following directory:

`$ROSERT_HOME/Scripts/ControlInteractions`

**19** Select `ControlInteractions_AddSequenceDiagrams.ebs`.

**20** Click Start (  ) to execute the second script.

The script will control all sequence diagrams in any checked-out collaborations.

**21** If the script detects that some elements need to be added to source control, you are prompted for permission. If you click **Yes**, the **Add To Source Control** dialog appears.

**22** The script will prompt you for permission to add newly controlled sequence diagrams to source control. If you click **Yes**, they are added to source control.


**23** Optional. Modify the script, if required. For example, if the script reported that a sequence diagram was not added to source control, you can investigate why it wasn't added, fix the problem and then manually add it to source control.

**24** Click **Tools > Open Script**.

**25** Browse to the following directory:

`$ROSERT_HOME/Scripts/ControlInteractions`

**26** Select the `ControlInteractions_CheckIn.ebs` script.

**27** Click Start (  ) to execute the third script.

The script will search for checked out collaborations and will check them in to your source control tool.

## Contents

This chapter is organized as follows:

- *Creating a Class* on page 333
- *Creating New Attributes* on page 334
- *Creating New Operations* on page 334
- *Class Specification* on page 335
- *Attribute Specification Dialog* on page 345
- *Operation Specification Dialog* on page 347
- *Creating a Capsule Class* on page 353
- *Capsule Diagrams* on page 354
- *Capsule Specification* on page 354

## Creating a Class

---

Classes can be created in either the **Logical View** or the **Use Case View** in the Model View Tab in the browser.

### To create a class:

- 1 Right-click on the **Logical View** package in the Model View Tab in the browser (or on the **Use Case View** package).
- 2 Select **New >Class** from the menu.  
A new class is created with a default name of 'NewClass1'.
- 3 Type over the name to change it.

## Creating New Attributes

---

### To create a new attribute on a class:

- 1 Right-click on the class in the Model View Tab in the browser.
- 2 Select **New > Attribute** from the menu.
- 3 Double-click on the attribute to open the Attribute Specification Dialog to set the name, class or type, code generation properties (for example, virtual), and so forth.

### Alternatively:

- 1 Open the Class Specification.
- 2 Select the **Attributes** tab.
- 3 Right-click and select **Insert** from the popup menu.

You can reorder attributes in the Specification dialog using drag-and-drop. You can undo and redo this action.

## Creating New Operations

---

### To create a new operation on a class:

- 1 Right-click on the class in the Model View Tab in the browser.
- 2 Select **New > Operation** from the menu.
- 3 Double-click on the operation to open the Operation Specification Dialog to set the name, parameters, return values, and so forth.

### Alternatively:

- 1 Open the Class Specification.
- 2 Select the **Operations** tab.
- 3 Right-click and select **Insert** from the popup menu.
- 4 Double-click on the new operation and use the Operation Specification Dialog to specify the operation details.

You can reorder operations in the Specification dialog using drag-and-drop. You can undo and redo this action.

## Class Specification

---

Use the Class Specification dialog to edit the properties of a class. The dialog provides access to all member attributes and operations as well.

### Class Specification Content

The class specification contains the following tabs:

- Class Specification - General Tab
- Class Specification - Detail Tab
- Class Specification - Operations Tab
- Class Specification - Attributes Tab
- Class Specification - Nested Tab
- Class Specification - Components Tab
- Class Specification - Relations Tab
- Class Specification - Files Tab

### Browse Button

Clicking **Browse** has the following options:

- **Select in Browser** - Highlights the selected element in the browser.
- **Open Diagram** - Opens the diagram associated with the object.
- **Browse Parent** - Opens the specification for the parent of the selected element.
- **Browse Selection** - Opens the specification for the currently selected element.
- **Show Usage** - Displays a list of all diagrams in which the currently selected element is the supplier, or in the case of a collaboration diagram, a list which shows the usage of a message.
- **Find References** - Finds all references of the item in the model by searching all fields - excluding Documentation - and searching all objects - excluding Diagrams, Component Instances, devices, Instances, Interactions, Messages, Packages, Probes, and Processors. The results appear in the **Find** tab in the **Output** window.

## Class Specification - General Tab

### Name

The name of the class.

### Parent

The parent the class belongs to (its package, or class in the case of a nested class) is displayed in this static field.

### Type

The **Type** choices for the selected model element are:

- Class
- Parameterized class
- Instantiated class
- Utility class
- Parameterized class utility
- Instantiated class utility
- Metaclass

### Stereotype

A stereotype represents the subclassification of an element. It represents a class within the UML metamodel itself, that is, a type of modeling element. Some stereotypes are already predefined, but you can also define your own to add new kinds of modeling types.

Stereotypes can be shown in the browser and on diagrams. The name of the stereotype may appear in angle brackets <<>>, depending on the settings found in either the Diagram or Browser tabs of the Options dialog box located under the **Tools** menu. Refer to the Stereotype chapter for more information on stereotypes.

To show stereotypes on the diagrams, click **Options** from the shortcut menu and click Stereotype Name or Stereotype Icon. Stereotype Name displays the name in angle brackets (that is, <<stereotype>>). Stereotype Icon displays the graphical representation.



## Language

Select the implementation language for the class from the available languages. The analysis selection indicates that no code will be generated for the class.

## Visibility

The Visibility field specifies how a class and its elements are viewed outside of the defined package

Select:	To Indicate:
<b>Public</b>	The element is visible outside of the enclosing package and you can import it to other portions of your model. Operations are accessible to all clients.
<b>Protected</b>	The element is accessible only to subclasses, friends, or the class itself.
<b>Private</b>	The element is accessible only to its friends or to the class itself.
<b>Implementation</b>	The element is visible only in the package in which it is defined. An operation is part of the implementation of the class.

The Visibility field can be set only in the specification. No special annotation is related to access control properties.

To change the visibility type for the class, click on the appropriate option in the visibility field. You can display the implementation visibility in the component compartment. You can display visibility in an icon through the shortcut menu.

## Documentation

Specifies documentation on this element.

## Class Specification - Detail Tab

### Multiplicity

The Multiplicity field specifies the number of expected instances of the class. In the case of relationships, this field indicates the number of links between each instance of the client class and the instance of the supplier. See Cardinality Options for more information.

## Space

Specifies the amount of storage required by objects of the class during execution.

## Persistence

Defines the lifetime of the instances of a class. A persistent element is expected to have a life span beyond that of the program or one that is shared with other threads of control or other processes.

The persistence of an element must be compatible with the persistence that you specified for its class. If a class persistence is set to Persistent, then the object persistence is either persistent, static or transient. If a class persistence is set to Transient, then the object persistence is either static or transient.

You can set the persistence only through the specification.

To set the persistence, click on the applicable option in the Persistence field. You can display the persistence in the diagram by selecting **Show Persistence** from the popup menu.

## Concurrency

Denotes the semantics in the presence of multiple threads of control. The Concurrency field shows the concurrency for the elements of a class. The concurrency of an operation should be consistent with its class.

Type Description:

- **Sequential** (default) - The semantics of the class are guaranteed only in the presence of a single thread of control. Only one thread of control can be executing in the method at any one time.
- **Guarded** - The semantics of the class are guaranteed in the presence of multiple threads of control. A guarded class requires collaboration among client threads to achieve mutual exclusion.
- **Active** - The class has its own thread of control.
- **Synchronous** - The semantics of the class are guaranteed in the presence of multiple threads of control; mutual exclusion is supplied by the class.

To change the concurrency, click on an applicable option button in the **Concurrency** box. You can display the concurrency in the class diagram by selecting **Show Concurrency** from the popup menu.

## Abstract

The **Abstract** field identifies a class that serves as a base class. An abstract class defines operations and states that will be inherited by subclasses. This field corresponds to the abstract class adornment displayed inside the class icon.

To toggle the abstract adornment, click on its check box.

When you click **Abstract**, the abstract class adornment is displayed in the lower left corner of the class icon. You can change the abstract class adornment only through the specification.

## Formal Arguments

In the **Parameterized Class Specification** or **Parameterized Class Utility Specification**, the formal, generic parameters declared by the class or class utility are listed.

In the **Instantiated Class Specification** or **Instantiated Class Utility Specification**, the actual arguments that match the generic parameters of the class being instantiated are listed.

You can add, update, or delete parameters only through the **Class Specification**. This field applies only to parameterized classes, parameterized class utilities, instantiated classes, and instantiated class utilities.

To define the parameters for a class, position the pointer within the **Parameters** box and click **Insert** from the shortcut menu or press the insert key.

Parameters are displayed on class diagrams.

## Class Specification - Operations Tab

Operations denote services provided by the class. Operations are methods for accessing and modifying Class fields, or methods that implement characteristic behaviors of a class.

The **Operations** tab lists the operations that are members of this class. The actual definition of the operation is accessible from the **Operation Specification Dialog**.

The **Operations** list has the following columns:

- **Visibility Adornment** (Unlabeled); the visibility of the operation is indicated with an icon. The visibility options are:
  - **Public** - The operation is accessible to all clients.
  - **Protected** - The operation is accessible only to subclasses, friends, or to the class itself.
  - **Private** - The operation is accessible only to the class itself or to its friends.
  - **Implementation** - The operation is accessible only by operations of this class.
- **Stereotype** - Displays the name of the stereotype.
- **Return Type** - Identifies the type of value returned from the operation.
- **Name** - Displays the name of the operations.
- **Parameters** - Shows the argument list for the operation. The information in this column can not be edited from this dialog. Double-click the argument list to open the **Operation Specification** dialog, then click the **Detail** tab to modify the arguments.
- **Class** - Identifies which class defines the operation.

The **Operations** tab is active for all class types. In the class diagram, you can display operation names in the class compartment.

### Show Inherited

Click this option to see operations inherited from other classes. If an operation is inherited, a blue arrow prefixes the Operation type symbol, ↵ ◆ .

If this option is not selected, you can view only operations associated with the selected class.

**Note:** Rational Rose RealTime allows you to directly modify any operation shown in the Operations list by displaying the Operations Specification dialog. Use caution when modifying base class operations because any changes may have implications on other elements in your model which reference, or are subclassed from, the base class.

### Creating New Operations

To enter an operation in the Class Specification, select **Insert** from the popup menu. A new operation with a default name is added to the operations list.

## Moving and Copying Operations

To move an operation from one Specification sheet to another, drag and drop it. From the **Edit** menu of the main window, you can select **Undo** and **Redo**.

To copy an operation from one Specification sheet to another, drag and drop it while holding down the Ctrl key. From the **Edit** menu of the main window, you can select **Undo** and **Redo**.

## Class Specification - Attributes Tab

The UML asserts that attributes are data values (string or integer) held by objects in a class. Thus, the Attributes tab lists attributes defined for the class. The attribute definition can be modified through the Attribute Specification Dialog.

**Note:** Attributes and relationships created using this technique are added to the model, but do not automatically appear in any diagrams. That is, adding an attribute affects the code generation for the class and a compilation dependency between the class of the container and the class of the attribute, but these relationships are not graphically visible in the model.

The descriptions for each field follow:

- **Visibility Adornment** (Unlabeled):
  - **Public** - The attribute is publicly visible, and is accessible to all clients.
  - **Protected** - The attribute may be accessed only by subclasses, friends, or by operations of this class.
  - **Private** - The attribute is accessible only by the class itself or by its friends.
  - **Implementation** - The attribute is accessible only by other operations in this class.
- **Stereotype** - Displays the name of the stereotype.
- **Name** - Displays the name of the attribute.
- **Class** - Identifies where the attribute is defined.
- **Type** - This can be a class or a traditional type, such as **int**.
- **Initial** - Displays the initial value of an object.

The **Attribute** tab is active for all class types.

## Show Inherited

Click this option to see attributes inherited from other classes. If there is no check mark in this field, you can view only attributes associated with the selected class.

**Note:** Rose RealTime allows you to directly modify any attribute shown in the attributes list by displaying the **Attribute Specification** dialog. You should be careful when modifying base class attributes for it may have implications on other elements in your model which reference or are subclassed from the base class.

## Creating New Attributes

You can add an attribute relationship by selecting **Insert** on the popup menu or by pressing the insert key. A new attribute with a default name is added.

## Moving and Copying Attributes

To move an attribute from one Specification sheet to another, drag and drop it. From the **Edit** menu of the main window, you can select **Undo** and **Redo**.

To copy an attribute from one Specification sheet to another, drag and drop it while holding down the Ctrl key. From the **Edit** menu of the main window, you can select **Undo** and **Redo**.

## Class Specification - Nested Tab

A nested class is a class that is enclosed within another class. Classes may contain instances of, inherit from, or use a nested class.

Enclosing classes are referred to as parent classes, and a class that lies underneath the parent class is called a nested class.

A nested class is typically used to implement functionality for the parent class. In many designs, a nested class is closely coupled to the parent class and is often not visible outside of the parent class. For example:

Think of your computer as a parent class and its power supply as a nested class. While the power supply is not visible outside the computer, the task it completes is crucial for the overall functionality of the computer.

## Moving and Copying Nested Classes

To move a Nested class from one Specification sheet to another, drag and drop it. From the **Edit** menu of the main window, you can select **Undo** and **Redo**.

To copy a Nested class from one Specification sheet to another, drag and drop it while holding down the Ctrl key. From the **Edit** menu of the main window, you can select **Undo** and **Redo**.

**To add a Nested Class from a Class Specification:**

- 1 Create and name a class.
- 2 Display the Class Specification.
- 3 Click on the Nested tab.
- 4 Right-click to display the shortcut menu, then click **Insert**.

A nested class entry with a default class name is inserted.

**To display a nested class:**

- 1 On the **Query** menu, select **Add Classes**.
- 2 Select the nested class and place it in the Selected Classes list box.

You can undo and redo the addition of nested classes.

**To delete a Nested Class from a Class Specification:**

- 1 Select the nested class from the Nested tab in the Class Specification.
- 2 Right-click on the class to display the popup menu.
- 3 From the popup menu, select **Delete**.

Or, use the following steps to delete a nested class:

- 1 Select the name of the nested class from the Nested Classes list on the Nested Classes tab.
- 2 Press the Delete key.

If you delete a nested class that is also a parent to other nested classes, all the nested classes are deleted.

You can undo and redo the deletion of nested classes.

**Note:** When you attempt to delete a nested class from a Class Specification, a warning dialog appears to verify the deletion.

## Relocating Nested Classes from the Browser to a Specification

Classes and Nested Classes can be moved from the browser to the Class Specification Nested tab. If you move a class (NewClassA) from the browser and place it directly on top of a class (NewClassB) on the Nested tab, NewClassA becomes nested underneath NewClassB. However, only one level of class nesting appears on the Nested tab. You can view all levels of nesting in the browser.

## Moving Nested Classes Between Class Specifications

Nested classes can be dragged and dropped between Class Specification Nested tabs.

## Class Specification - Components Tab

### Components List

The components list displays a list of components to which this class has been assigned. Components can be inserted, deleted, and moved up and down in the list. Each component has a corresponding Component Specification for editing the component attributes.

A check-box provides filtering control over which components are displayed:

**Show all components** displays the list of all components in the model.

Right-clicking on a component open the Component shortcut menu.

## Class Specification - Relations Tab

### Relations List

The relations list displays relations between the class and other model classes as specified in class diagrams. The relations list simply displays the relationships involving this class that appear on class diagrams in the model.

Each relation has a corresponding Association Specification for editing the relation attributes.

A check-box provides filtering control over which relations are displayed:

**Show Inherited** shows any relations inherited from a superclass protocol.



## Class Specification - Files Tab

A list of referenced files is provided here. You can link external files to model elements for documentation purposes.

## Class Specification - Diagrams Tab

The **Diagrams** area lists the diagrams for the selected class. For classes, you can add Collaboration and State diagrams.

**Note:** You can delete State and Collaboration diagrams.

The first column contains the icon that corresponds to the diagram's type. The **Title** column contains the title for the diagram. You can modify the name of the title.

To add a new diagram, use the shortcut menu and select the desired **Insert** option.

## Attribute Specification Dialog

---

The **Attribute Specification** dialog lets you display and modify the properties of a class or capsule attribute in the current model.

To display an Attribute Specification, select the entry on the Attribute tab of the Class or Capsule Specification and click **Specification** from the shortcut menu. Alternatively, double-clicking on the entry displays the Attribute Specification.

### Specification Content

The **Attribute Specification** dialog has of the following tabs: **General**, **Detail**, **Files**, and other language-specific tabs.

### Browse Button

Clicking **Browse** has the following options:

- **Select in Browser** - Highlights the selected element in the browser.
- **Open Diagram** - Opens the diagram associated with the object.
- **Browse Parent** - Opens the specification for the parent of the selected element.
- **Browse Selection** - Opens the specification for the currently selected element.
- **Show Usage** - Displays a list of all diagrams in which the currently selected element is the supplier, or in the case of a collaboration diagram, a list which shows the usage of a message.

- **Find References** - Finds all references of the item in the model by searching all fields - excluding Documentation - and searching all objects - excluding Diagrams, Component Instances, devices, Instances, Interactions, Messages, Packages, Probes, and Processors. The results appear in the **Find** tab in the **Output** window.

## General Tab

### Name

A name for the attribute. This name will be the name for the generated attribute.

### Stereotype

A stereotype value.

### Class

The class the attribute belongs to is displayed in this non-editable field.

### Visibility

- **Public** - The attribute is visible to any other classes.
- **Protected** - The attribute is visible only to subclasses and friend classes.
- **Private** - The attribute is not visible to any other classes, except designated friend classes.
- **Implementation** - The attribute is never visible to other classes.

### Scope

- **Class** - There is a single instance of the attribute for all instances of the class (for example a static member in C++ terminology).
- **Instance** - Each instance of the class will have a separate attribute instance.

## Detail Tab

### Type

Attribute types can either be classes or language-specific types. When the attribute is a data value, the type is defined as a language-specific type. You can enter the type in the Type field of the Class Attribute Specification. Rational Rose RealTime displays the type beside the attribute name in the class icon and updates the information in the model.

**Note:** If the **Type** box contains a class, the corresponding label becomes a hot link to the Specification dialog for that class.

### Initial Value

You can assign an initial value to your class attribute through this box. You can specify multi-line expressions for initial data.

### Changeability

- Changeable - The attribute can be modified.
- Frozen - The attribute cannot be modified.
- Add-only - The attribute can only be updated in an additive way. This is not enforceable in most programming languages.

### Derived

The **Derived** check box indicates whether the element was computed or implemented directly.

To define a element as derived, select the **Derived** check box. The element name is adorned by a "/" in front of the name.

If the derived box is checked, no code is generated for the attribute.

## Operation Specification Dialog

---

Complete one Operation Specification for each operation that is a member of a class.

If you change a class operations property by editing its specification, Rational Rose RealTime updates all class diagrams containing icons representing that class.

To access the Operation Specification, select an entry on the **Operations** tab of the Class Specification and double-click the entry or click Insert from the popup menu. You can also open the Specification dialog using the shortcut menu.

## Specification Content

The **Operation Specification** dialog has the following tabs: **General**, **Detail**, **Validations**, **Semantics**, **Files**, <*language-specific tab*>.

### Browse Button

Clicking **Browse** has the following options:

- **Select in Browser** - Highlights the selected element in the browser.
- **Open Diagram** - Opens the diagram associated with the object.
- **Browse Parent** - Opens the specification for the parent of the selected element.
- **Browse Selection** - Opens the specification for the currently selected element.
- **Show Usage** - Displays a list of all diagrams in which the currently selected element is the supplier, or in the case of a collaboration diagram, a list which shows the usage of a message.
- **Find References** - Finds all references of the item in the model by searching all fields - excluding Documentation - and searching all objects - excluding Diagrams, Component Instances, devices, Instances, Interactions, Messages, Packages, Probes, and Processors. The results appear in the **Find** tab in the **Output** window.

## General Tab

### Name

The name of the operation. The named operation will be generated as a member of the containing class.

### Stereotype

Specifies a stereotype for the operation.

### Class

A non-editable field that displays the class to which the operation belongs.

## Visibility

- **Public** - Indicates that the operation is visible to other classes.
- **Protected** - Indicates that the operation is not part of the public interface of the class, but is visible to subclasses.
- **Private** - Indicates that the operation is not visible to other classes, including subclasses. May be visible to specific classes designated as friend classes.
- **Implementation** - Indicates the operation is not visible to any other classes, including subclasses and friends.

## Options

- **Polymorphic** - Indicates that the operation should be inherited by all subclasses.
- **Query** - Indicates that the operation is read-only and does not modify the object's state.
- **Abstract** - Indicates that the operation is an abstract definition that should be overridden by specific implementations in subclasses.

## Scope

- **Instance** - Indicates that the operation operates on individual class instances, usually because its calculations are based on the object state, or because it modifies the object state.
- **Class** - Indicates that the operation operates the same way regardless of the state of any individual object in the class.

## Detail Tab

### Return Type

For operations that are functions, set this field to identify the class or type of the function's result. If show classes is set, the list box displays all the classes in the package. If **Show classes** is not set, only the predefined set of return class types is displayed.

If you enter a class name and it does not exist in your model, the application does not create one.

**Note:** If the **Return Type** box contains a class, the corresponding label becomes a hot link to the Specification dialog for that class.

## Parameters

Contains a list of the arguments of the operation. You may express these arguments in your selected implementation language.

The argument list can be rearranged with the click and drag technique. Select an argument from the list, drag it to the location, and release. The list reflects the new order.

**Note:** Double-click on a name in the **Type** column to open the appropriate Specification dialog for that type.

### Parameter Specification Dialog

To open the **Parameter Specification** dialog, double-click a parameter. The dialog has two tabs: **General** and **Files**.

The **General** tab contains fields for **Name**, **Type**, **Default**, and **Documentation**. It also provides the name of the owner of the parameter, that is, the operation that the parameter belongs to.

The **Files** tab provides a list of referenced files. You can link external files to model elements for documentation purposes.

You can **Undo** and **Redo** any changes from the **Edit** menu.

### Moving and Copying Parameters:

To move a parameter from one specification sheet to another, drag and drop it. From the **Edit** menu of the main window, you can select **Undo** and **Redo**.

To copy a parameter from one specification sheet to another, drag and drop it while holding down the Ctrl key. From the **Edit** menu of the main window, you can select **Undo** and **Redo**.

## Code

A code editor allowing you to enter the detailed implementation code for the operation.

## Validation Tab

### Protocol

This field lists a set of operations that a client can perform on an object and the legal orderings in which they might be invoked. The protocol of an operation has no semantic impact.

## Qualifications

This field identifies language-specific features that qualify the method.

## Exceptions

This field contains a list of the exceptions that can be raised by the operation. Enter the name of one or more classes identifying the exception.

## Size

This field identifies the relative or absolute amount of storage consumed by the invocation of the operation.

## Time

This field contains a statement about the relative or absolute time required to complete an operation. Use this field to budget time for the operation.

## Concurrency

This field denotes the semantics in the presence of multiple threads of control. The Concurrency field shows the concurrency for the elements of a class. The concurrency of an operation should be consistent with its class.

**Table 1    Concurrency Field Options**

Type	Description
<b>Sequential</b> (default)	The semantics of the operation are guaranteed only in the presence of a single thread of control. Only one thread of control can be executing in the method at any one time.
<b>Guarded</b>	The semantics of the operation are guaranteed in the presence of multiple threads of control. A guarded class requires collaboration among client threads to achieve mutual exclusion.
<b>Synchronous</b>	The semantics of the operation are guaranteed in the presence of multiple threads of control; mutual exclusion is supplied by the class.

You can set the concurrency of a class only through the **Class Specification**. To change the concurrency, click on an applicable option in the **Concurrency** box. You can display the concurrency in the class diagram by clicking **Show Concurrency** from the context menu.

## Semantics Tab

### Preconditions

Invariants that are assumed by the operation (the entry behavior of an operation) are listed.

### Semantics

The action of the operation is shown in this area.

### Postcondition

Invariants that are satisfied by the operation (the exit behavior of an operation) are listed in this area.

### Interaction Diagram

Select an interaction diagram from the list box that illustrates the appropriate semantics. Selecting <New> brings up the New Interaction Diagram dialog, in which you can specify the diagram type and title.

## Parameter Specification Dialog

---

To open the **Parameter Specification** dialog, double-click a parameter. The dialog has two tabs: **General** and **Files**.

### Name

Displays the name of the parameter.

### Owner

Displays the owner of the parameter, that is, the operation to which the parameter belongs.

### Type

Specifies the type for this parameter.

**Note:** If the **Type** box contains a class, the corresponding label becomes a hot link to the Specification dialog for that class.



### **Default**

Specifies the default value for this parameter.

### **Documentation**

Describes any details about the use of this parameter.

### **Files Tab**

The **Files** tab provides a list of referenced files. You can link external files to model elements for documentation purposes.

You can **Undo** and **Redo** any changes using the **Edit** menu.

## **Creating a Capsule Class**

---

Capsule classes are created in the Logical View of the Model View Tab in the browser.

### **To create a new capsule class:**

- 1** Right-click on the Logical View package (or another package of your choice) in the model browser.
- 2** Select the **New > Capsule** menu option. A new capsule class is created with a default name of 'NewCapsule1'.
- 3** Type over the name to change it.

You can also create new capsule classes using the capsule tool in the class diagram.

Each capsule has an associated structure diagram and state diagram.

The capsule class attributes, operations and other properties can be modified through the **Capsule Specification**. Open the specification dialog by double-clicking on the capsule in the model browser.

## Capsule Diagrams

---

There are two diagrams associated with capsules:

### State Diagram

The state diagram captures the high-level behavior of the capsule.

### Structure Diagram

The structure diagram captures the interface and internal structure of the capsule in terms of its contained capsules and ports.

### Undocking the Capsule Diagrams

These two diagrams can be docked together or viewed separately. To separate the diagrams into separate windows, grab one of the diagram tabs at the bottom of the window and drag it away to create a new window.

## Capsule Specification

---

The capsule specification is used to edit the properties of a capsule.

The capsule specification dialog contains the following tabs:

- Capsule Specification - General Tab
- Capsule Specification - Operations Tab
- Capsule Specification - Attributes Tab
- Capsule Specification - Capsule Roles Tab
- Capsule Specification - Ports Tab
- Capsule Specification - Connectors Tab
- Capsule Specification - Relations Tab
- Capsule Specification - Components Tab
- Capsule Specification - Files Tab

## Browse

Clicking **Browse** has the following options:

- **Select in Browser** - Highlights the selected element in the browser.
- **Open Diagram** - Opens the diagram associated with the object.
- **Browse Parent** - Opens the specification for the parent of the selected element.
- **Browse Selection** - Opens the specification for the currently selected element.
- **Show Usage** - Displays a list of all diagrams in which the currently selected element is the supplier, or in the case of a collaboration diagram, a list which shows the usage of a message.
- **Find References** - Finds all references of the item in the model by searching all fields - excluding Documentation - and searching all objects - excluding Diagrams, Component Instances, devices, Instances, Interactions, Messages, Packages, Probes, and Processors. The results appear in the **Find** tab in the **Output** window.

## Capsule Specification - General Tab

### Name

The name of the capsule class. The capsule class name may be referenced in the detailed code of other capsule classes. One reason for this is for a container capsule to instantiate an optional capsule role (see the Frame service incarnate function).

### Stereotype

A stereotype represents the subclassification of an element. It represents a class within the UML metamodel itself, that is, a type of modeling element. Some stereotypes are already predefined, but you can also define your own to add new kinds of modeling types.

Stereotypes can be shown in the browser and on diagrams. The name of the stereotype may appear in angle brackets <<>>, depending on the settings found in either the Diagram or Browser tabs of the Options dialog located under the **Tools** menu. Refer to the Stereotype chapter for more information on stereotypes.

To show stereotypes on the diagrams, click **Options** from the shortcut menu and click Stereotype Name or Stereotype Icon. Stereotype Name displays the name in angle brackets (that is, <<stereotype>>). The Stereotype Icon displays the graphical representation.

## Language

Specifies the language to use for detailed coding and code generation.

## Documentation

Specifies documentation for this element.

## Capsule Specification - Diagrams Tab

The **Diagrams** area lists the diagrams for the selected capsule. For capsules, you can add Collaboration, Sequence, and State diagrams.

**Note:** There can be only one State diagram, and you can only delete Collaboration diagrams.

The first column contains the icon that corresponds to the diagram's type. The **Title** column contains the title for the diagram. You can modify the name of the title.

To add a new diagram, use the shortcut menu and select the desired **Insert** option.

## Capsule Specification - Operations Tab

Operations denote services provided by the class. Operations are methods for accessing and modifying Class fields or methods that implement characteristic behaviors of a class.

The **Operations** tab lists the operations that are members of this class. The actual definition of the operation is accessible from the Operation Specification Dialog.

The operations are listed with the following fields:

- **Visibility Adornment** (Unlabeled); the visibility of the operation is indicated with an icon. These are the visibility options:
  - **Public** - The operation is accessible to all clients.
  - **Protected** - The operation is accessible only to subclasses, friends, or to the class itself.
  - **Private** - The operation is accessible only to the class itself or to its friends.
  - **Implementation** - The operation is accessible only by the implementation of the package containing the class.
- **Stereotype** - Displays the name of the stereotype.

- **Signature** - Displays the name of the operation.
- **Class** - Identifies which class defines the operation.
- **Return Type** - Identifies the type of value returned from the operation.

The **Operations** tab is active for all class types. In the class diagram, you can display operation names in the class compartment.

### Show Inherited

Set this option to display operations inherited from other classes. If an operation is inherited, a blue arrow prefixes the Operation type symbol, ↵ ◆ .

If this option is not selected, you can view only operations associated with the selected class.

**Note:** Rational Rose RealTime allows you to directly modify any operation shown in the operations list by displaying the Operations Specification dialog. Use caution when modifying base class operations because any changes may have implications on other elements in your model which reference, or are subclassed from, the base class.

### Creating New Operations

To enter an operation in the **Class Specification**, select **Insert** from the popup menu. A new operation with a default name is added to the operations list.

## Capsule Specification - Attributes Tab

The UML asserts that attributes are data values (string or integer) held by objects in a class. Thus, the Attributes tab lists attributes defined for the class. The attribute definition can be modified through the Attribute Specification Dialog.

**Note:** Attributes and relationships created using this technique are added to the model, but do not automatically appear in any diagrams. That is, adding an attribute affects the code generation for the class and a compilation dependency between the class of the container and the class of the attribute, but these relationships are not graphically visible in the model.

The descriptions for each field follow:

- **Visibility Adornment** (Unlabeled):
  - **Public** - The attribute is publicly visible, and is accessible to all clients.
  - **Protected** - The attribute may be accessed only by subclasses, friends, or by operations of this class.
  - **Private** - The attribute is accessible only by the class itself or by its friends.
  - **Implementation** - The attribute is accessible only by operations in this class.
- **Stereotype** - Displays the name of the stereotype.
- **Name** - Displays the name of the attribute.
- **Class** - Identifies where the attribute is defined.
- **Type** - This can be a class or a traditional type, such as **int**.
- **Initial** - Displays the initial value of an object.

The Attribute tab is active for all class types.

### Show Inherited

Set this option to display attributes inherited from other capsules. If there is no check mark in this field, you can view only attributes associated with the selected capsule.

**Note:** Rose RealTime allows you to directly modify any attribute shown in the attributes list by displaying the **Attribute Specification** dialog. You should be careful when modifying base class attributes for it may have implications on other elements in your model which reference or are subclassed from the base class.

### Creating New Attributes

You can add an attribute relationship by selecting **Insert** on the popup menu or by pressing the insert key. A new attribute with a default name is added.

## Capsule Specification - Capsule Roles Tab

The capsule roles list displays all contained capsule roles within the immediate capsule decomposition. Capsule roles can be inserted, deleted, and moved up and down in the list. Each capsule role has a corresponding Capsule Role Specification for editing the capsule role attributes.

Inserting Capsule Roles through the Capsule Roles list is the same as adding capsule roles through the Structure diagram editor. When inserting a new capsule role, a pick-list appears allowing you to select the class for the capsule role. The new capsule role is given a default name, which can be changed by double-clicking on it.

Three check-boxes provide filtering control over which capsule roles are displayed:

- **Inherited Values** - Shows any elements inherited from a superclass protocol.
- **Local Values** - Shows any elements defined within this capsule (not inherited).
- **Excluded Values** - Shows elements defined in the superclass and deleted from the subclass.

Right-clicking on a capsule role brings up the Capsule Role popup menu.

## Capsule Specification - Ports Tab

The ports list displays all contained ports within the immediate capsule decomposition. Ports can be inserted, deleted, and moved up and down in the list. Each port has a corresponding Port Specification for editing the port attributes.

Inserting ports through the list is the same as adding ports through the Structure diagram editor. When inserting a new port, a pick-list appears allowing you to select the protocol for the port. The new port is given a default name, which you can change by double-clicking on it.

Three check-boxes provide filtering control over which ports are displayed:

- **Inherited Values** - Shows any elements inherited from a superclass protocol.
- **Local Values** - Shows any elements defined within this capsule (not inherited).
- **Excluded Values** - Shows elements defined in the superclass and deleted from the subclass.

Right-clicking on a signal displays the **Port** context menu.

## Capsule Specification - Connectors Tab

The connectors list displays all connectors contained within the immediate capsule decomposition. Connectors can be deleted, and moved up and down in the list. Connectors cannot be inserted through this list: they can only be defined through the Capsule Collaboration Diagram Editor. Each connector has a corresponding Connector Specification for editing the connector attributes.

Three check-boxes provide filtering control over which connectors are displayed:

- **Inherited Values** - shows any elements inherited from a superclass protocol.
- **Local Values** - shows any elements defined within this capsule (not inherited).
- **Excluded Values** - shows elements defined in the superclass and deleted from the subclass.

Right-clicking on a signal opens the **Connector** context menu.

## Capsule Specification - Relations Tab

### Relations List

The relations list displays relations between the class and other model classes as specified in class diagrams. The relations list displays the relationships involving this class that appear on class diagrams in the model.

Each relation has a corresponding Association Specification for editing the relation attributes.

A check-box provides filtering control over which relations are displayed:

**Show Inherited** shows any relations inherited from a superclass protocol.

## Capsule Specification - Components Tab

### Components List

The components list displays a list of components to which this class has been assigned (a red check mark on the icon). Components can be inserted, deleted, and moved up and down in the list. Each component has a corresponding Component Specification for editing the component attributes.

A check-box provides filtering control over which components are displayed:

**Show all components** displays all the components in the model.

Right-clicking on a component opens the **Components** context menu.

## Capsule Specification - Files Tab

A list of referenced files is provided here. The files list popup menu allows you to insert and delete references to files or URLs.

You can link external files to model elements for documentation purposes.



## Contents

This chapter is organized as follows:

- *Protocol Specification* on page 361
- *Signal Specification* on page 364

## Protocol Specification

---

The protocol specification provides control over the definition of a protocol class. The dialog includes the following tabs:

- Protocol Specification - General Tab
- Protocol Specification - Signals Tab
- Protocol Specification - Relations Tab
- Protocol Specification - Components Tab
- Protocol Specification - Files Tab

### Browse Button

Clicking **Browse** displays the following options:

- **Select in Browser** - Highlights the selected element in the browser.
- **Open Diagram** - Opens the diagram associated with the object.
- **Browse Parent** - Opens the specification for the parent of the selected element.
- **Browse Selection** - Opens the specification for the currently selected element.
- **Show Usage** - Displays a list of all diagrams in which the currently selected element is the supplier, or in the case of a collaboration diagram, a list which shows the usage of a message.
- **Find References** - Finds all references of the item in the model by searching all fields - excluding Documentation - and searching all objects - excluding Diagrams, Component Instances, devices, Instances, Interactions, Messages, Packages, Probes, and Processors. The results appear in the **Find** tab in the **Output** window.

## Protocol Specification - General Tab

### Name

The name of the Protocol Class.

### Language

Select the implementation language for the class from the available languages. The analysis selection indicates that no code will be generated for the class.

### Stereotype

A stereotype represents the subclassification of an element. It represents a class within the UML metamodel itself, that is, a type of modeling element. Some stereotypes are already predefined, but you can also define your own to add new kinds of modeling types.

Stereotypes can be shown in the browser and on diagrams. The name of the stereotype may appear in angle brackets <<>>, depending on the settings found in either the Diagram or Browser tabs of the Options dialog located under the **Tools** menu. Refer to the Stereotype chapter for more information on stereotypes.

To show stereotypes on the diagrams, click **Options** from the shortcut menu and click Stereotype Name or Stereotype Icon. Stereotype Name displays the name in angle brackets (that is, <<stereotype>>). Stereotype Icon displays the graphical representation.

### Documentation

Specifies documentation for this element.

## Protocol Specification - Signals Tab

This tab provides a list of signals that can be received (the **In** list) and sent (the **Out** list) by ports using this protocol.

**Note:** A tilde character, "~", opposite an item indicates that the item is conjugated.

### In/Out Signal List

The signal list allows signals to be inserted, deleted, and moved up and down in the list. Each signal has a corresponding Signal Specification for editing the signal attributes.

Three check-boxes provide filtering control over which signals are displayed:

- **Show inherited** - Shows any signals inherited from a superclass protocol.
- **Show local** - Shows signals defined within this protocol (not inherited).
- **Show excluded** - Shows signals defined in the superclass protocol and deleted from the subclass protocol.

Right-clicking on a signal brings up the Signal popup menu, allowing you to insert new signals, delete signals, and promote/demote signals in the protocol class hierarchy. As well, you can select **Open Data Class Specification**, which brings up the Class Specification.

### Copying Signals

To copy a signal from one Specification sheet to another, drag and drop it. From the **Edit** menu of the main window, you can select **Undo** and **Redo**.

## Protocol Specification - Relations Tab

### Relations List

The relations list displays relations between the protocol class and other model classes as specified in class diagrams. Relations can be inserted, deleted, and moved up and down in the list. Each relation has a corresponding Association Specification for editing the relation attributes.

A check-box provides filtering control over which relations are displayed:

**Show Inherited** shows any relations inherited from a superclass protocol.

Right-clicking on a relation opens the Relation context menu.

## Protocol Specification - Components Tab

### Components List

The components list displays a list of components to which this class has been assigned (a red check mark on the icon). Components can be inserted, deleted, and moved up and down in the list. Each component has a corresponding Component Specification for editing the component attributes.

A check-box provides filtering control over which components are displayed:

**Show all components** displays the list of components in the model.

Right-clicking on a component opens the Components context menu.

## Protocol Specification - Diagrams Tab

The **Diagrams** area lists the diagrams for the selected protocol. For protocols, you can add Sequence and State diagrams.

The first column contains the icon that corresponds to the diagram's type. The **Title** column contains the title for the diagram. You can modify the name of the title.

To add a new diagram, use the shortcut menu and select the desired **Insert** option.

## Protocol Specification - Files Tab

A list of referenced files is provided here. You can link external files to model elements for documentation purposes.

## Signal Specification

---

The dialog shows information about a signal in a protocol class. The signal specification is opened from the **Protocol Specification - Signals Tab**.

### Browse Button

Clicking **Browse** displays the following options:

- **Select in Browser** - Highlights the selected element in the browser.
- **Open Diagram** - Opens the diagram associated with the object.
- **Browse Parent** - Opens the specification for the parent of the selected element.
- **Browse Selection** - Opens the specification for the currently selected element.

- **Show Usage** - Displays a list of all diagrams in which the currently selected element is the supplier, or in the case of a collaboration diagram, a list which shows the usage of a message.
- **Find References** - Finds all references of the item in the model by searching all fields - excluding Documentation - and searching all objects - excluding Diagrams, Component Instances, devices, Instances, Interactions, Messages, Packages, Probes, and Processors. The results appear in the **Find** tab in the **Output** window.

## Signal Specification - General Tab

### Name

Specifies a name for the signal. The name is referenced in detail code when a capsule sends a message through a port, and in the trigger event for transitions in the capsule state diagram (through the Event Editor Dialog).

### Data Class

Specifies the class of the data object that is expected as a payload of the message. The data class field has a pull-down menu, which allows you to pick from the list of available data classes and types in the model.

**Note:** If the **Data Class** box contains a class, the corresponding label becomes a hot link to the Specification dialog for that class.

## Signal Specification - Files Tab

A list of referenced files is provided here. The files list popup menu allows you to insert and delete references to files or URLs.

You can link external files to model elements for documentation purposes.



## Contents

This chapter is organized as follows:

- *Introduction to Packages* on page 367
- *Creating a Package* on page 367
- *Package Specification* on page 368
- *Moving Model Elements* on page 372

## Introduction to Packages

---

Packages are organized model elements in larger models. Packages break up large models containing hundreds or thousands of elements into smaller, more manageable conceptual units. When properly designed, packages usually represent units of work for an individual or team, and units of reusability. That is, a package represents a set of highly related (highly cohesive) model elements. In most cases when looking to reuse portions of a model across software projects, entire packages would be reused rather than individual classes.

Packages also define the directory structure of a stored model. When a model is stored as controlled units, a subdirectory is created for each package, such that the representation of the model on disk mirrors the packaging hierarchy of the model in the tool.

## Creating a Package

---

Packages can be created in the **Use Case View**, the **Logical View** or the **Component View** of the browser. Packages can contain other packages. In fact, the four main views in the model browser are themselves packages.

### To create a package:

- 1 Right-click on the package in the model browser where you want the new package to be created.
- 2 Select **New >Package** from the menu.

A new package will be created with a default name of 'NewPackage1'.

- 3 Type over the name to change it.

New model elements can be created within the package by clicking on the package and selecting the right-mouse button to access the popup menu.

Existing model elements can be moved across packages. See *Moving Model Elements* on page 372.

## Packages and Class Diagrams

Packages can be displayed in class diagrams to show dependencies among packages. It is useful in large systems to construct top-level class diagrams that just show the packages, and allow users to drill down into the individual packages for more detailed class diagrams.

## Package Specification

---

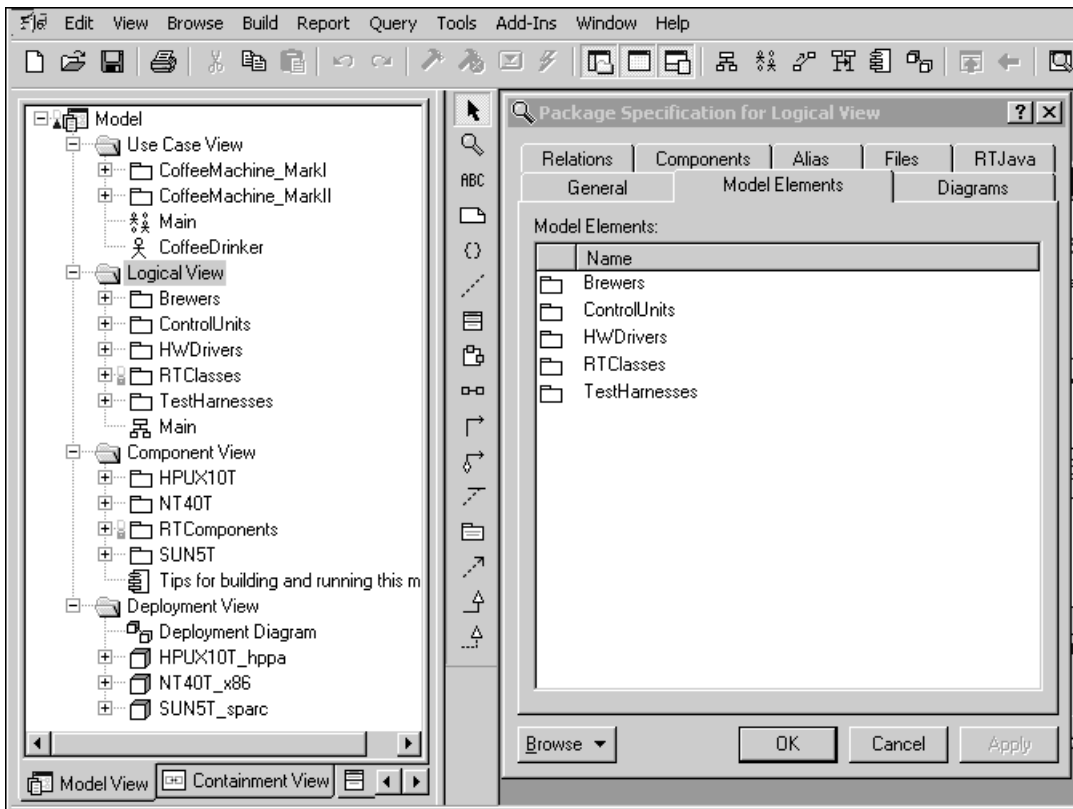
A Logical Package Specification enables you to display and modify the properties and relationships of a logical package in the current model.

If you change a package's properties or relationships by editing its specification, the application updates all class diagrams containing icons representing that logical package. If you change a logical package's properties or relationships by editing a diagram containing its icon, the application updates the logical package's specification and any other diagrams containing its icon.

The package specification dialog provides control over the definition of a package. The dialog includes the following tabs:



**Figure 86 Package Specification - Model Elements Tab**



## Package Specification - General Tab

### Name

The name of the package.

### Parent

Displays the name of the parent package. If this is one of the top-level view packages, the parent is the Model.

### Stereotype

Displays the stereotype of the package. There are no pre-defined package stereotypes.

## **Documentation**

Specifies documentation for this element.

## **Package Specification - Detail Tab**

### **Global**

The Global check-box indicates that all public classes in the logical package can be used by any other logical package.

To switch the global adornment, click on the Global check-box. When you set the global indicator, Rational Rose displays the word “global” in the lower left corner of the logical package icon.

You can change the global adornment only through the specification.

### **Diagrams**

This field lists the diagrams contained in the package. When you add a diagram to the package, Rational Rose automatically updates this list.

The first column contains the diagram’s icon. The Title field is the title of the diagram you entered (and can be modified).

To add a new diagram, use the shortcut menu and select the appropriate insert diagram option.

## **Package Specification - Relations Tab**

### **Relations List**

The relations list displays relations between the package and other model classes and packages as specified in class diagrams. Relations can be inserted, deleted, and moved up and down in the list. Each relation has a corresponding Association Specification for editing the relation attributes.

A check-box provides filtering control over which relations are displayed.

### **Show Inherited**

Shows any elements inherited from a superpackage.

Right-clicking on a relation brings up the Relation popup menu.

## Package Specification - Components Tab

### Components List

The components list displays a list of components that reference this package (red checkmark). Components can be inserted, deleted, and moved up and down in the list. Each component has a corresponding Component Specification for editing the component attributes.

A check-box provides filtering control over which components are displayed:

**Show all components** displays the list of all components in the model.

Right-clicking on a component displays the Components context menu.

## Package Specification - Files Tab

A list of referenced files is provided here. The files list popup menu allows you to Insert and Delete references to files or URLs.

You can link external files to model elements for documentation purposes.

## Package Specification - Model Elements Tab

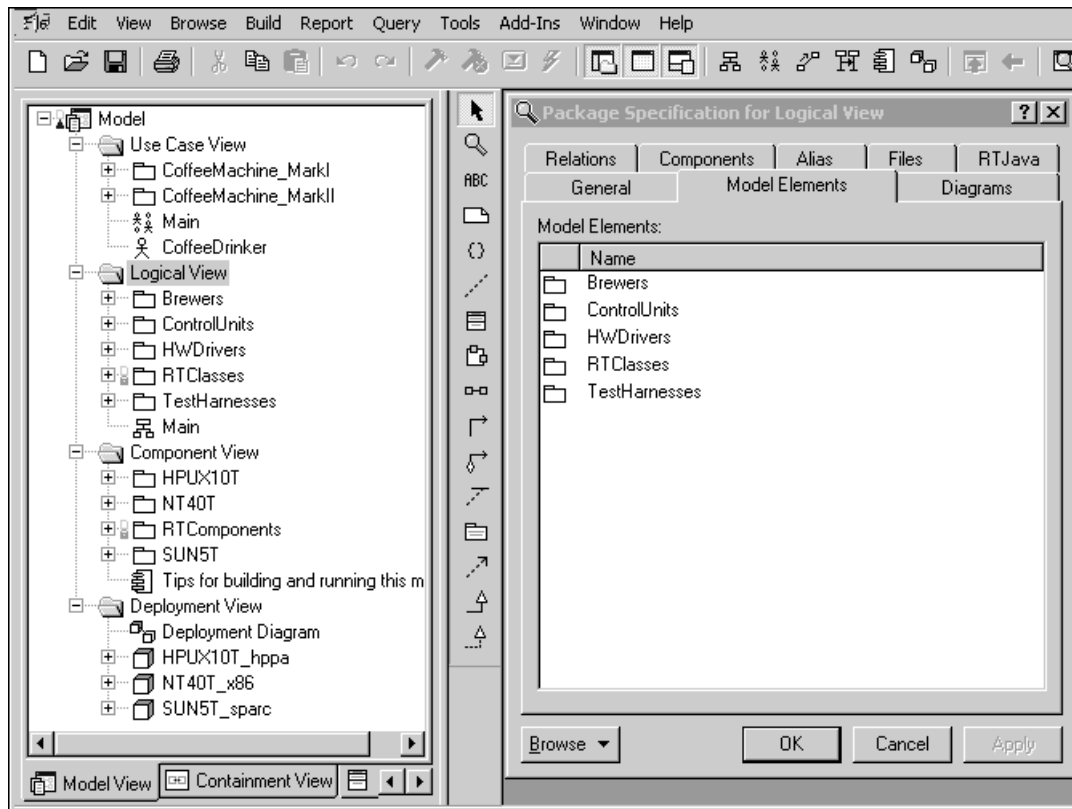
There is a new tab on the **Package Specification** dialog box called **Model Elements** (see Figure 86). The purpose of this tab is to facilitate model navigation. This tab is only available for **Use Case View** and **Logical View** packages, as well as any other similar packages contained therein. This tab does not exist for **Component Views**.

The **Model Elements** tab lists all of the model elements seen from the **Model View** tab in the browser including: actors, capsules, classes, class utilities, interfaces, packages, protocols, and use cases.

**Note:** The list on the **Model Elements** tab does not include diagrams.

In the **Model Elements** tab, you can open the **Specification** dialog box for the selected element, or delete selected items from the list.

**Figure 87 Package Specification - Model Elements Tab**



## Moving Model Elements

You can move model elements and diagrams from one package into another package on the **Model View Tab** in the browser.

### To move a model item:

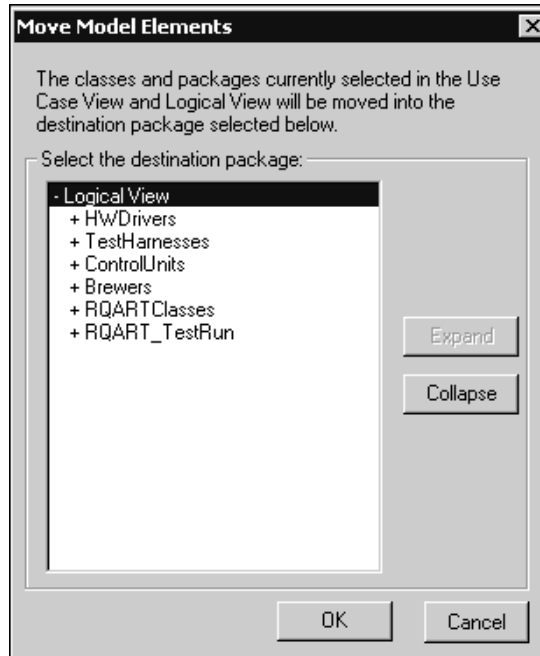
- 1 Click on the class in the model browser.
- 2 Drag it over the destination package.

### To move multiple model elements using the Move Model Elements feature:

**Note:** The **Move Model Elements** feature can move capsules, protocols, classes, and packages from the **Logical View** in the browser.

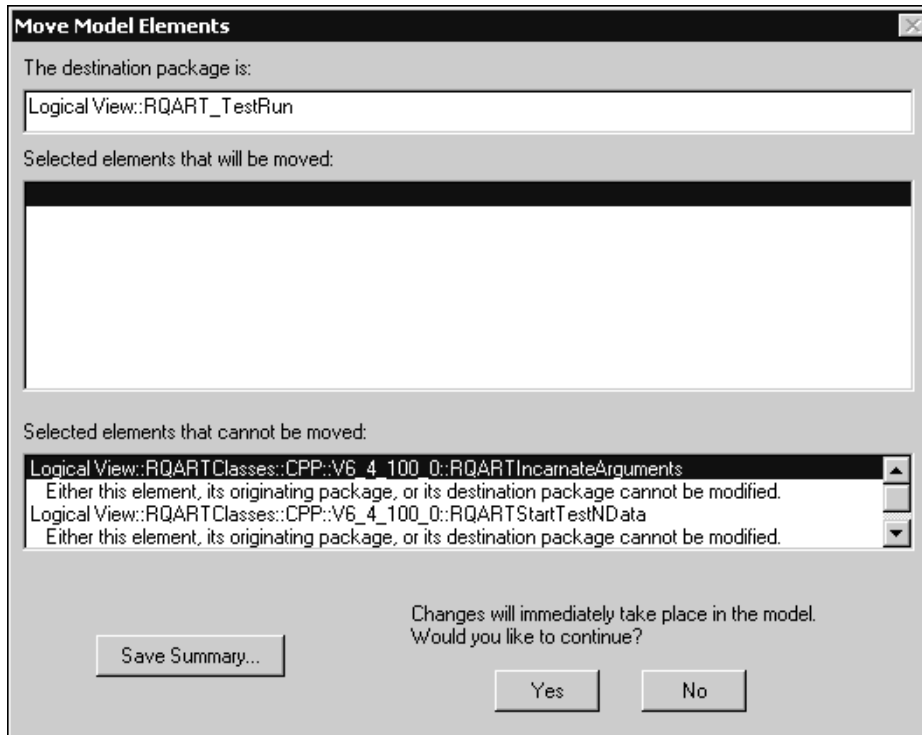
- 1 Select the model elements from the **Logical View** in the **Model View** tab in the browser, or select the model elements from a **Class** diagram.
- 2 Click **Tools > Move Model Elements**.

The **Move Model Elements** dialog shows the hierarchical list of all the packages in the **Logical View**.



- 3 Find and select the package to move the selected model elements.

4 Click **Ok**.



The **Move Model Elements** summary dialog shows the destination packages and the model elements to move, as well as any model elements that cannot be moved.

- 5 To save the information on the **Mode Model Elements** summary dialog, click **Save Summary**, and then specify a file name and location.
- 6 Click **Yes**.

The selected model elements move to the new location.

## Impact of Moving Classes or Diagrams on Configuration Management

Because classes are stored in the configuration Management (CM) system under the package directory, moving a class to another package causes a mismatch between the stored directory structure and the model packaging. The classes in the stored model directory are not automatically moved to new directories. The mismatch does not cause any problems for the toolset (Rational Rose RealTime keeps track of the stored file name for the element. For details, see the **Unit Information tab**. However, it may cause confusion for users working directly with a CM tool.

### To synchronize the system files with moved model elements:

- 1 On the **Model View** tab in the browser, select an element that you previously moved.
- 2 Right-click and select **Open Specification**.
- 3 Click the **Unit Information** tab.
- 4 Specify the location to save this file.
- 5 Click **Save As**.
- 6 Repeat Step 1 through Step 5 for each controlled unit that you moved in the model browser.
- 7 Right-click on the package containing the controlled units, and then click **File > Save Unit**.
- 8 Launch your CM tool.
- 9 Use the commands from your CM tool to move the model elements to the new location.





## Contents

This chapter is organized as follows:

- *Using the Component Diagram Editor* on page 377
- *Component Diagram Toolbox* on page 379
- *Using the Deployment Diagram Editor* on page 380
- *Deployment Diagram Toolbox* on page 382

## Using the Component Diagram Editor

---

Use the Component Diagram editor to create a diagram showing the software as releasable units, together with their interfaces and inter-dependencies. Multiple Component diagrams can exist in the same model.

A Component Diagram shows the physical dependency relationships (mapping to a file system) between components - main programs, subprograms, packages, and tasks - and the arrangement of components into component packages.

Component diagrams are contained (owned) either at the top level of the model or by a package, which means that the diagram depicts the components and packages where the diagram is contained.

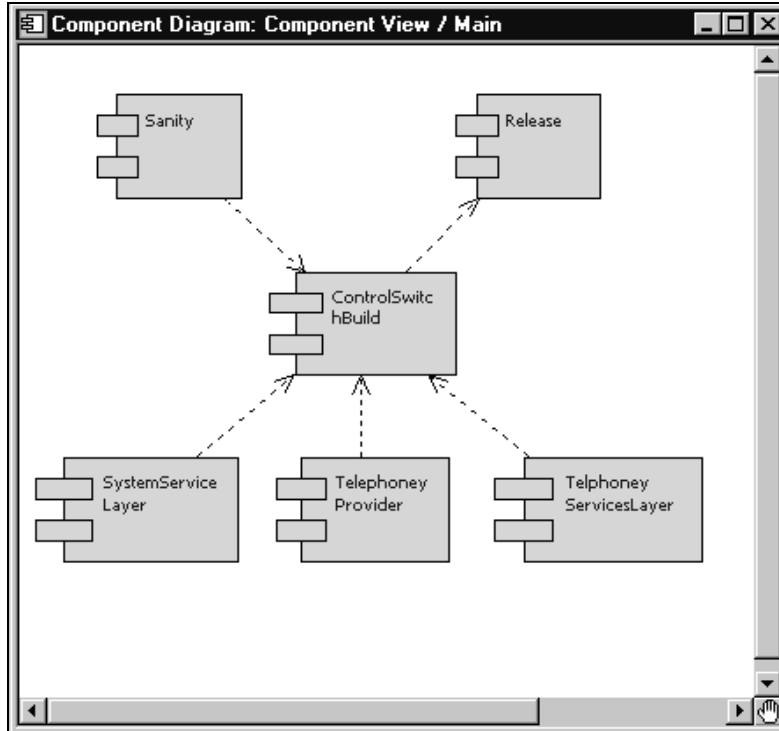
The component diagram consists of two parts:

- The diagram area
- The Component Diagram Toolbox

Elements of the component diagram, such as packages and components, are added using the toolbox or by dragging them from the browser. You can undo and redo moves from the **Edit** menu.

The window title bar shows the full name of the component diagram.

**Figure 88 Component Diagram**



### **Component**

Components can be added to the diagram using either the component tool from the toolbox, or by selecting a component from the Model View Tab in the browser and dragging and dropping it on to the diagram. Components may have dependency or aggregation relationships with other components.

The component details are specified through the Component Specification.

### **Dependency**

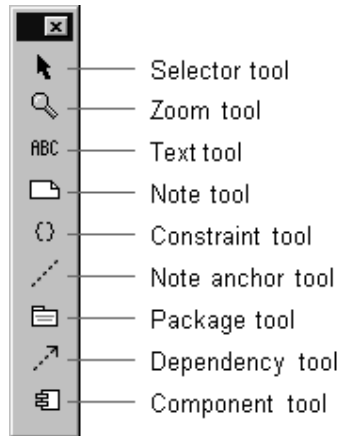
A dependency indicates a client and supplier relationship. The client depends on the supplier to provide certain services. Use this relationship to indicate that the operations of the client invoke operations of the supplier.

# Component Diagram Toolbox

---

The component toolbox contains tools for adding elements to the component diagram.

**Figure 89** Component diagram toolbox



## Selector Tool

Selects objects for moving, resizing, and so forth.

## Zoom Tool

Use to zoom in on a portion of the diagram. Click on the tool and then click on the part of the diagram you want to zoom in on.

## Text Tool

Adds text anywhere in the structure diagram.

## Note Tool

Annotates the diagram with textual notes. This is useful for marking up the diagram with explanations, review comments, and so forth. You can drag and drop a diagram or external document from the browser onto a note. Notice that the name of the diagram or external document is underlined. If you double-click on the note, the diagram or external document is opened. You can undo and redo this command.

### **Constraint Tool**

Adds UML constraints to the diagram. A constraint can be anchored to a view element by using the anchor tool. Currently, constraints do not have any semantic meaning to the tool. There are RRTEI APIs to add or remove, and enumerate constraints in a diagram.

### **Note Anchor Tool**

Anchors a note to a particular element on the diagram.

### **Package Tool**

Adds a package to the diagram. The package is given a default name such as 'NewPackage1'.

### **Dependency tool**

Indicates that a dependency is between packages or between components. A dependency indicates that some element in one package depends on (uses) some element in another package.

### **Component tool**

Adds a component to the diagram. The component is given a default name such as 'NewComponent1'. See *Building Basics* on page 439 for more information on creating and building components.

## **Using the Deployment Diagram Editor**

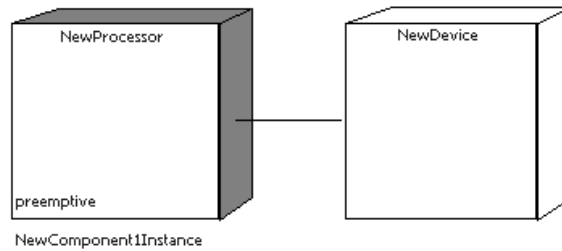
---

The deployment diagram editor is used to create a diagram showing system deployment across processing nodes. The deployment diagram shows the allocation of processes to processors in the physical design of a system. A deployment diagram may represent all or part of the process architecture of a system. Multiple deployment diagrams can exist in the same model. The deployment diagram consists of two parts:

The diagram area, and the toolbox.

The window title bar shows the full name of the deployment diagram.

**Figure 90 Deployment Diagram Editor**



## Deployment Diagram Elements

A deployment diagram shows the hardware configuration of the system under construction, and the distribution of software across that configuration.

There are four types of elements that can be placed on the diagram:

- Two types of hardware node: processors and devices
- Connections between the hardware nodes
- Software component instances deployed on the hardware nodes

### Processors

A processor is a hardware component capable of executing programs. You can further define a processor by identifying its processes and specifying the type of process scheduling it uses.

The Processor Specification Dialog dialog provides details on processor attributes.

### Devices

A device is a hardware component with no computing power. Each device must have a name. Device names can be generic, such as "modem" or "terminal."

The device specification dialog provides details on device attributes.

### Connections

A connection represents some type of hardware coupling between two nodes. The hardware coupling can be direct, such as an RS232 cable, or indirect, such as satellite-to-ground communication. Connections are usually bi-directional.

The Connector Specification provides details on connection attributes.

## Components

Components can be placed on processors for control over the distribution of the software for execution. The result is a component instance that can be specified through the Processor Specification Dialog dialog.

## Packages

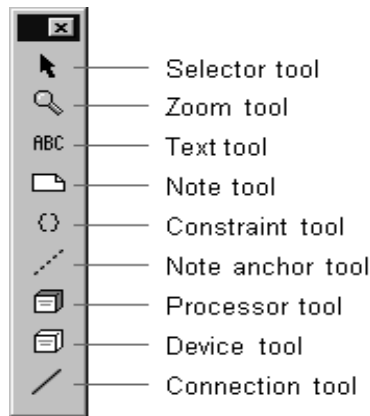
Use to add a package to the diagram. The package is given a default name such as 'NewPackage1'.

# Deployment Diagram Toolbox

---

The deployment diagram toolbox contains tools for adding elements to the deployment diagram.

**Figure 91** Deployment diagram toolbox



### Selector Tool

Selects objects for moving, resizing, and so forth.

### Zoom Tool

Use to zoom in on a portion of the diagram. Click on the tool and then click on the part of the diagram you want to zoom in on.

### Text Tool

Adds text anywhere in the structure diagram.

## **Note Tool**

Annotates the diagram with textual notes. This is useful for marking up the diagram with explanations, review comments, and so forth. You can drag and drop a diagram or external document from the browser onto a note. Notice that the name of the diagram or external document is underlined. If you double-click on the note, the diagram or external document is opened. You can undo and redo this command.

## **Constraint Tool**

Adds UML constraints to the diagram. A constraint can be anchored to a view element by using the anchor tool. Currently, constraints do not have any semantic meaning to the tool. There are RRTEI APIs to add or remove, and enumerate constraints in a diagram.

## **Note Anchor Tool**

Anchors a note to a particular element on the diagram.

## **Processor tool**

Adds a processor node to the diagram. Click on the diagram to place a new processor at the selected location.

Processors are given default names, such as 'processor1', when initially drawn. To change the name, click on the device and hit the backspace key to delete the default name, then type the new name.

## **Device Tool**

Adds a device node to the diagram. Click on the diagram to place a new device at the selected location. Devices are given default names, such as 'device1', when initially drawn.

## **Connection Tool**

Adds a connection between two nodes on the diagram. Click on the first node on the diagram and drag the connection to the second node.





## Contents

This chapter is organized as follows:

- *Importing a Petal or Package File* on page 385
- *Importing Code from Rational Rose to Rational Rose RealTime* on page 385
- *Referencing an External Library* on page 396
- *Using the Convert Rose Component Wizard* on page 397
- *Exporting a File* on page 399

To add external .class files, see *Add External Java Tool* in the *Add-in, Tool, and Wizard Guide, Rational Rose RealTime*.

## Importing a Petal or Package File

---

Rational Rose RealTime can import several different types of files. The file types are:

- .rtptl (RRT petal file)
- .ptl (Rose petal file)
- .cat (Rose package file)
- .sub (Rose component package file)

### To import an element into a Rational Rose RealTime model:

- 1 Select **File > Import**
- 2 Select the type of file you want to import.
- 3 Select the file and click **Open**.

## Importing Code from Rational Rose to Rational Rose RealTime

---

To use a Rational Rose model in Rational Rose RealTime, you need to export a Rational Rose model, and then import it into Rational Rose RealTime. In addition to converting a model and importing a Rational Rose model into Rational Rose RealTime, you will need to import any code stored outside your Rational Rose model.

Code generation is different in Rational Rose and in Rational Rose RealTime. In Rational Rose, the toolset generates skeleton code stored outside of the model. You can edit the skeleton code with an editor to add additional code, or modify existing code (such as operation bodies). In Rational Rose RealTime, you are not required to edit the code after it has been generated. In some cases, code is stored in the model itself (such as method bodies).

## Using the Code Import Process

To import the external code associated with a Rational Rose Model into Rational Rose RealTime, you must perform a series of tasks. If the tags generated by Rational Rose are good and you do not want to import code other than that which Rational Rose marks as "to be imported," you only need to complete the following steps:

- *Preparing the Rational Rose Model for Import on page 386*
- *Importing the Code on page 394*

If the generated code does not contain the tags generated by Rational Rose, or the tags were modified (perhaps when a Rational Rose user modified the source files to edit the body of an operation), or you want to specify different export options from those specified as the default by Rational Rose, such as **friend** relationships, you will need to complete the following steps:

- *Preparing the Rational Rose Model for Import on page 386*
- *Launching the C++ Analyzer on page 387*
- *Specifying Export Options and Selecting a Source File Location on page 388*
- *Analyzing the Code on page 391*
- *Using CodeCycle to Add Tags to Code on page 393*
- *Importing the Code on page 394*

**Note:** Because the model and the code are modified during the import process, we strongly recommend that you make a copy of the model and source code files prior to starting this process.

## Preparing the Rational Rose Model for Import

To import a Rational Rose model into Rational Rose RealTime, you must prepare your Rational Rose model and its associated code. For example, if the model contains a class that is not present in the source code, then the model and source code will become out of sync.

To prepare your Rational Rose model and associated code for import, we recommend that you perform the following tasks in Rational Rose:

- Click **File > Update** to synchronize your model with the code.
- Use the C++ Analyzer to check the source code to ensure that there are no errors. Errors in the model can cause problems which may make it difficult to import to Rational Rose RealTime.
- Click **Tools > Check Model** to assist with identifying problems in the model.  
**Note:** The **Check Model** tool will not find all problems; however, you should address anything it finds prior to attempting to migrate to Rational Rose RealTime.

The following steps are mandatory and must be performed prior to migrating from Rational Rose to Rational Rose RealTime.

- 1 In Rational Rose, click **File > Save As** and in the **Type** box, select one of the available formats, such as **Rose 6.1/6.5 Model**.  
**Note:** You can save your Rational Rose model in the **Rose 6.1/6.5 Model formats** or higher prior to loading it in Rational Rose RealTime.
- 2 Save your Rational Rose model to a single file.  
**Note:** If a package is controlled in a Rational Rose model, Rational Rose RealTime will not be able to load the Rational Rose model.

## Launching the C++ Analyzer

The C++ Analyzer is an executable that is an add-in to both Rational Rose and Rational Rose RealTime. The C++ Analyzer tool is responsible for checking source code for errors, and for updating the source code with tags that the Rational Rose RealTime Code Import tool recognizes.

### To start the C++ Analyzer:

- 1 In Rational Rose, click **Tools > C++ > Reverse Engineering** to start the C++ Analyzer.  
**Note:** If the C++ option is not available from the **Tools** menu, click **Add-Ins > Add-in Manager** and select **Rose C++** from the list.
- 2 Click **File > New**.
- 3 If the **\$DATA PathMap** symbol is not set, you are prompted to enter a value for **\$DATA**.

- 4 Click **OK**.
- 5 Create a **PathMap** entry.

**Note:** This directory stores the temporary data generated by the C++ Analyzer. We recommend that you create a new temporary directory and set the **PathMap** variable to that location.

Next, you want to set export options for your Rational Rose model.

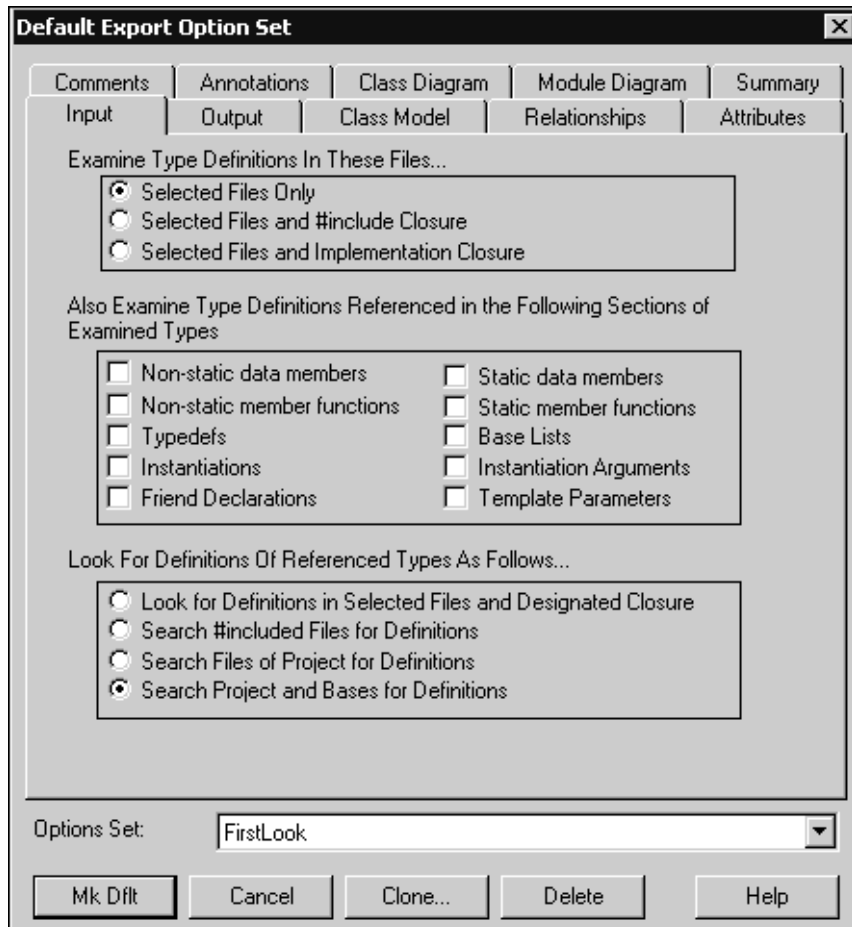
## **Specifying Export Options and Selecting a Source File Location**

The C++ Analyzer performs an export process on the Rational Rose model based on options that you specify, called **Export Options**. Setting these export options determines what source code to export from your Rational Rose model. These export options also affect the tags that the C++ Analyzer adds to the source code. Rational Rose RealTime uses these tags during the import process.

### **To specify export options:**

- 1 Open the project associated with your Rational Rose model. If a project does not exist, click **File > New**.
- 2 In the C++ Analyzer, click **Edit > Export Options** to specify what parts of the source code to examine.

**Note:** By default, many options are not selected, such as **friend** declarations.

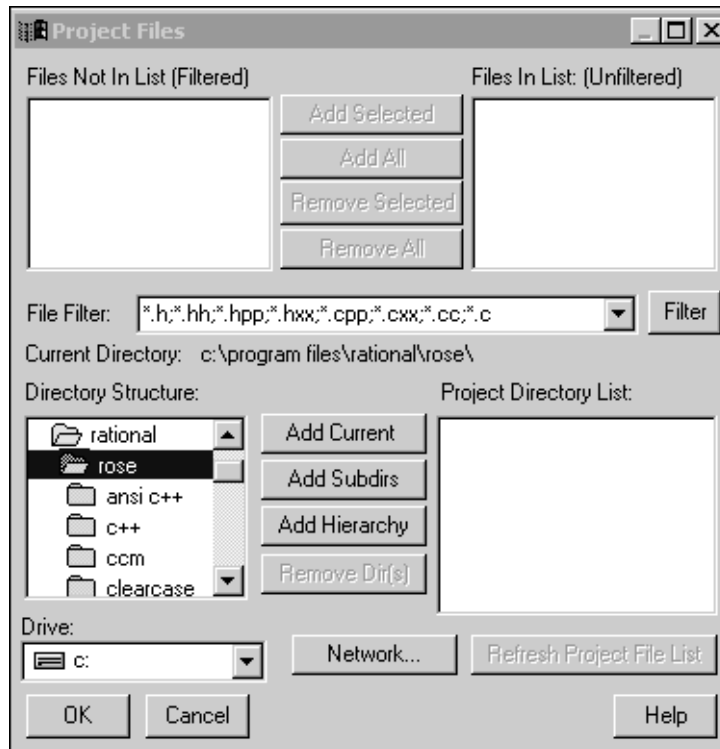


- 3 Review and set the appropriate options on all of the tabs in the **Default Export Option Set** dialog to ensure the C++ Analyzer examines the appropriate code.
- 4 Click **Update** to update the current **Export Option Set** to reflect the modified export options that you specified.

**Note:** The **Update** button is only available if you modify any export options since the current option set was selected or updated.

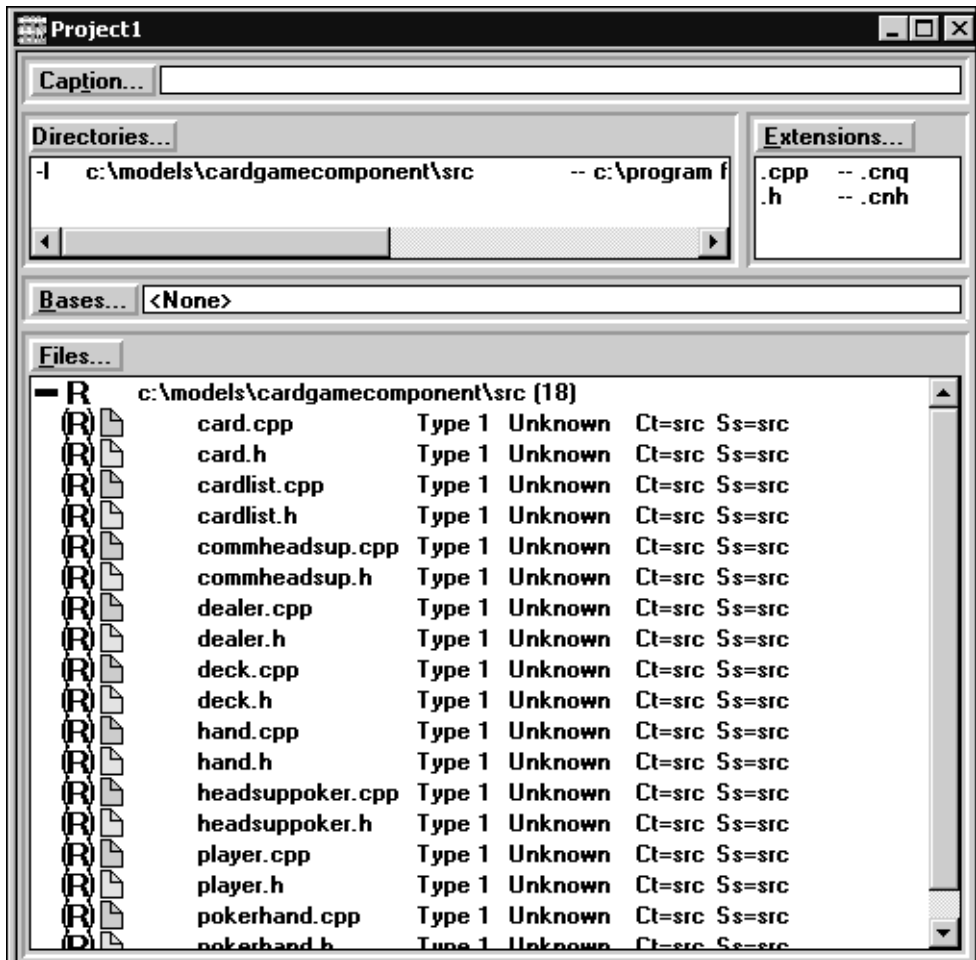
- 5 Click **Close**.

- 6 Click **Edit > Files List**.



- 7 In the **Directory Structure** area, browse to the directory that contains the source code files for your model.
- 8 Add the source files to the **Files In List** area, or click **Add All** to add all of the files at once.
- 9 Click **OK**.

The files are added to the open project in the C++ Analyzer.



Next, you want the C++ Analyzer to analyze the code you specified.

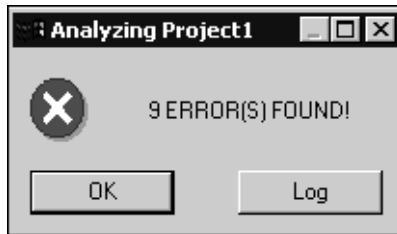
## Analyzing the Code

Use the C++ Analyzer to check the source code for errors.

### To select the files and analyze them:

- 1 In the C++ Analyzer, select individual files in the project to analyze, or click **Edit > Select All** to select all of the files in the **Project** dialog.
- 2 Click **Action > Analyze**.

When the analyze process completes, you are presented with a list of errors identified by the C++ Analyzer.

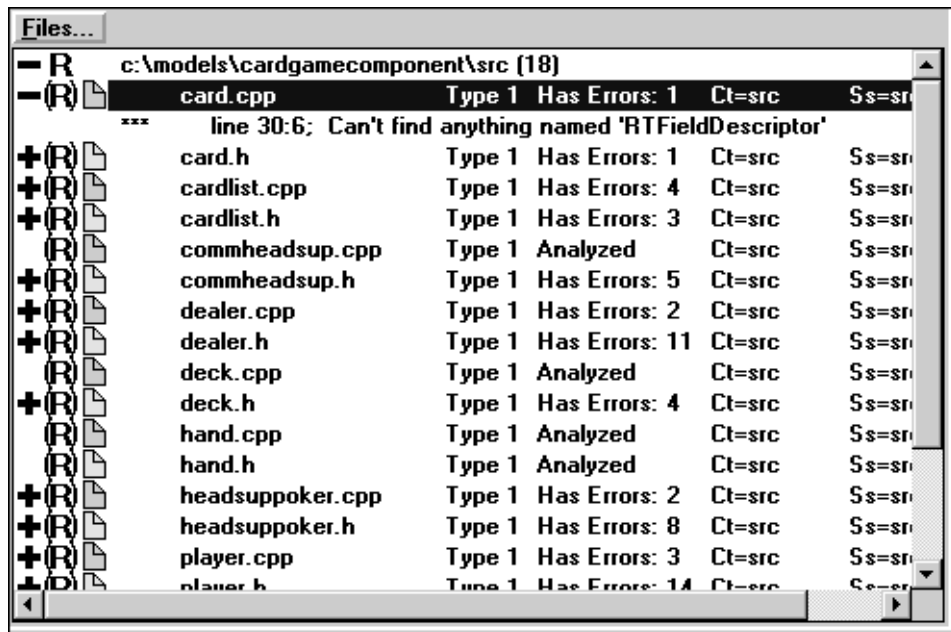


- 3 Click **OK**.

Files that contain errors have a plus sign (+) displayed next to them in the list of selected files on the **Project** dialog.

- 4 In the **Project** dialog, double-click on a problem file to view the errors.

**Note:** The plus sign changes to a negative sign (-) to indicate the class information has expanded.



- 5 Double-click on an error to open an editor on the code.



When possible, the C++ Analyzer highlights the problem area in the code so you can immediately see the problem.

**Note:** You are not required to fix these errors at this time. The Rational Rose RealTime **Code Import** tool will continue to import the code containing these errors.

## Using CodeCycle to Add Tags to Code

The Rational Rose RealTime Code Import tool looks for specific tags in the source code to identify the begin and end of a section of code. For example, each operation body has a start and end tag to identify what code is included in the operation body. The **CodeCycle** operation of the C++ Analyzer adds these tags to the source code.

### To add tags to the code:

- 1 In the C++ Analyzer for the currently open project, click **Action > CodeCycle** to start executing the **CodeCycle** process.

**Note:** The C++ Analyzer prompts you to proceed as a precautionary step since it will be modifying your source code.

- 2 Click **Yes** to proceed.

**Note:** The C++ Analyzer will notify you if it encountered any errors.



cardgamecomponent.h ???

Has Errors: 1

Ct=rtsystem Ss=rtsystem

### Example

The following example shows the tags added to a member function of the class Engine.

```
int Engine::start()
{
//## begin Engine::start%F0556897FEED.body preserve=yes
    return 1;
//## end Engine::start%F0556897FEED.body
}
```

All tags will start with either **begin** or **end**. The rest of the tag depends on the type of code being imported. In the above example, we are importing the body of a member function so the next part of the tag is the name of the class followed by the name of the function. Following that is the Rational Rose GUID for the corresponding operation, followed by a unique tag (body in this case) that describes the type of information (the body of a member function). The Rational Rose RealTime **Code Import** tool uses these tags to determine where to place the code in the model.

The final part of the **begin** tag is **preserve=yes**. When this option is set to **no**, the Rational Rose RealTime **Code Import** tool will not import that code segment (from the **begin** to the **end** tag). For example, the **preserve** option is set to **no** when a section of code matches one of the export options that you specified not to have imported (see *Specifying Export Options and Selecting a Source File Location on page 388*).

## Importing the Code

After the CodeCycle process adds tags to the code, you want to open the model in Rational Rose RealTime and import the code.

### To import C++ code:

- 1 Start Rational Rose RealTime.
- 2 Open a Rational Rose model.

If errors are encountered as the model loads, a dialog appears stating that errors occurred. You can get more information about the errors from the Log tab in the Output window.

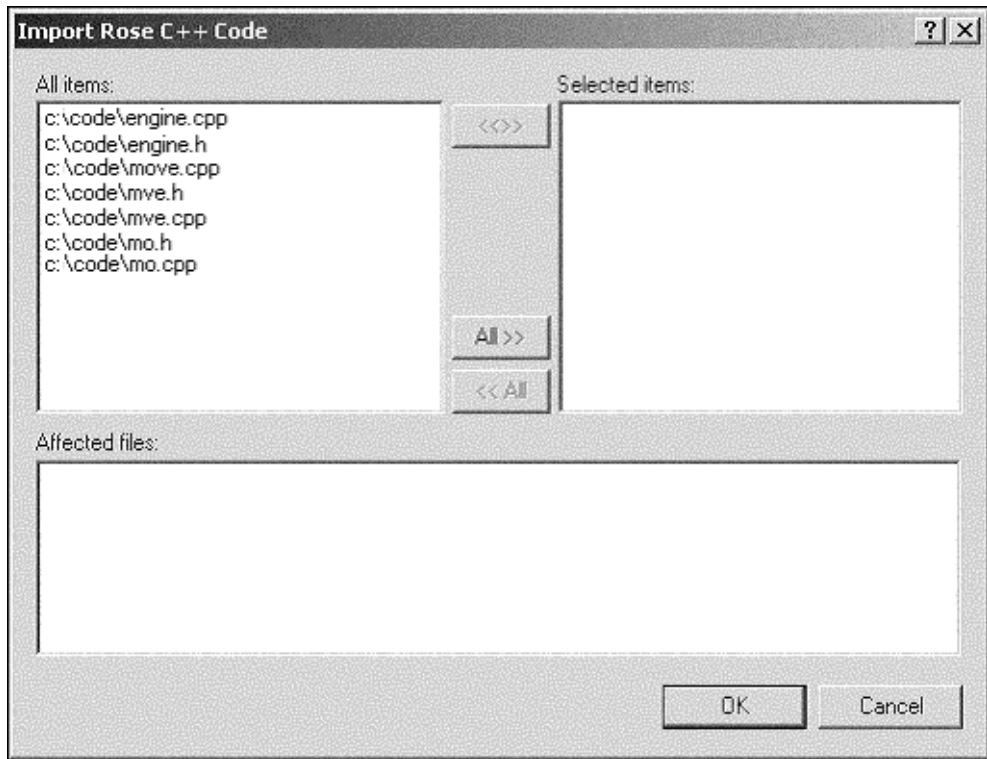
You are prompted with a dialog indicating that the model was saved in an old format. The purpose of this dialog is to inform you that the next time you save the model in Rational Rose RealTime, it is saved using the Rational Rose RealTime format.

- 3 Click **Tools > Convert From Rose Classic C++**.

The toolset runs the **AddCodeImportProperties** script.

**AddCodeImportProperties** is a SummitBasic script that adds properties to classes in the model. It can create these properties for models that use ANSI or Classic C++. The location of this script is in the `$ROSERT_HOME\bin\%ROSERT_HOST` directory. The source code for this script is in the `$ROSERT_HOME\Scripts\RoseImport` directory. This script is automatically executed by the toolset after selecting the Import Rose C++ Code. When the script completes, all the classes in the model will contain the required properties for the *Code Import* tool to find the source files.

The **Code Import** tool displays a list of source files that it gathered from properties of the classes in the model.



**Note:** The **Code Import** tool examines the properties of classes and components to determine the location of source code files. A component is examined only if its stereotype is set to **Package Body** or **Package Specification**. If the files associated with a component do not appear in the list of files to import, check the stereotype for the component to ensure it is set to **Package Body** or **Package Specification**.

- 4 Click **All >>** to select all the files, or select only a subset of the files.
- 5 Click **OK** to import the code into the model.

The toolset executes the **ConfigureFromRoseProperties** script. The **ConfigureFromRoseProperties** is a SummitBasic script that sets various Rational Rose RealTime model properties based on the corresponding Rational Rose properties. The location of this script is in the `$ROSE_HOME\bin\%ROSE_HOST` directory. The source code for this script is located in the `$ROSE_HOME\Scripts\RoseImport` directory.

**Note:** The **ConfigureFromRoseProperties** script checks the language of each class. If the language is not C++, it does not configure any properties. For example, if a class has Classic C++ properties, and its language is currently set to ANSI C++, the Rational Rose RealTime properties are not configured based on the Rational Rose properties.

Since the **ConfigureFromRoseProperties** script does not examine all of the Rational Rose properties, you can modify this script if there are additional properties you want to process.

**Note:** A Readme file, `ConfigureFromRoseProperties-README.txt`, is included in the `$ROSE_HOME\Scripts\RoseImport` directory. This file contains additional information about the properties that the **ConfigureFromRoseProperties** script currently processes.

## Referencing an External Library

---

In Rational Rose RealTime, you can make use of an external library when developing a model, including libraries created from Rational Rose models. This means that Rational Rose users can continue to develop a model using Rational Rose while referencing those libraries (built from a Rational Rose model), in a Rational Rose RealTime model.

To reference an external library from within Rational Rose RealTime, you can configure manually using the **Component Specification** dialog, or you can use the **Convert Rose Component** wizard. The **Convert Rose Component** wizard quickly configures a C++ External Library component, by retrieving information that Rational Rose adds to a component, to reduce the number of manual steps.

**Note:** The **Convert Rose Component** wizard is only available for the C++ language.

For additional information on the Convert Rose Component Wizard, see *Using the Convert Rose Component Wizard* on page 397.

## Using the Convert Rose Component Wizard

---

You can use the **Convert Rose Component** wizard to configure the **Inclusion Paths** and **Libraries** properties, and to automatically retrieve existing inclusion paths from a component. You can view the **Inclusion Paths** and **Libraries** information in Rational Rose by opening the **Specification** dialog for a component, and selecting either the **C++** or **ANSI C++** tab (depending on the type of component you have selected). For **C++** components, the inclusion paths are stored in the **AdditionalIncludes** property. For **ANSI C++** components, the **Inclusion Paths** are stored in either the **RevEngRootDirectory** or **NewHeaderFileDirectory** properties.

**Note:** The directory that contains the header file for the component that was generated by Rational Rose is also included in the **Inclusion Paths** list.

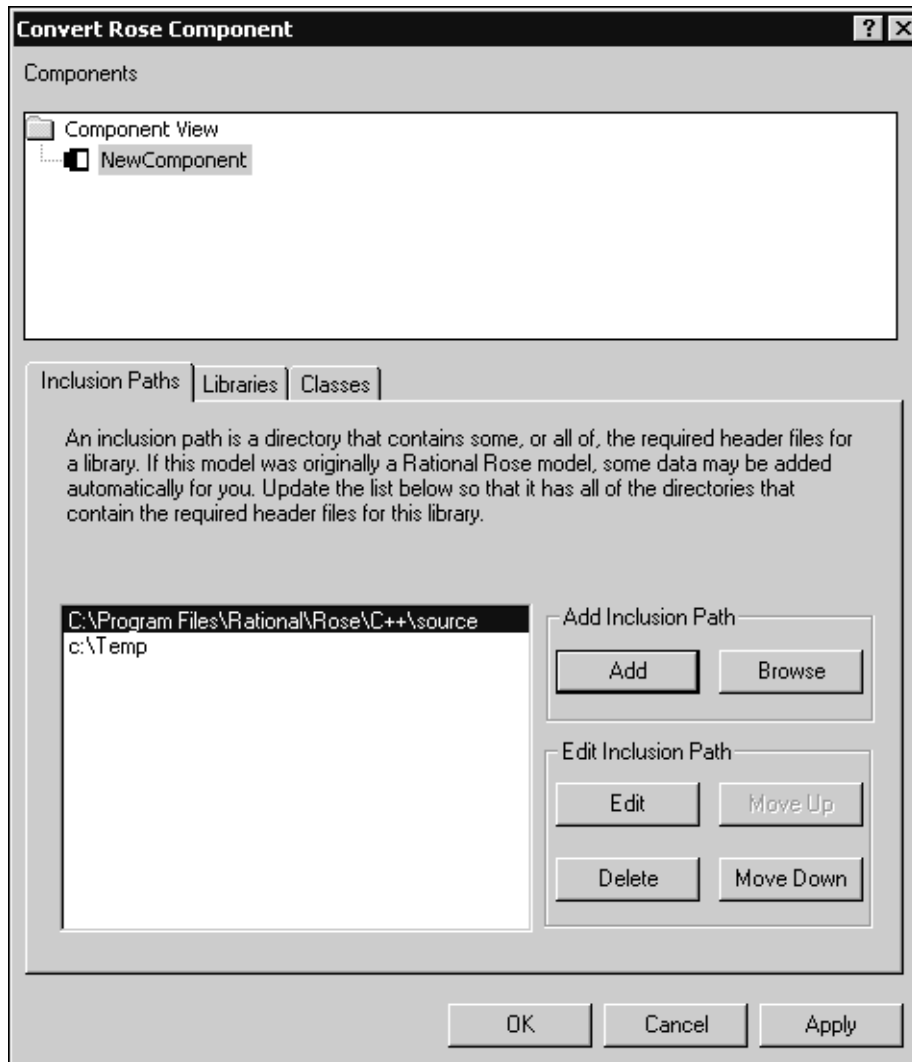
### To convert a component:

- 1 Start Rational Rose RealTime.
- 2 Open a Rational Rose model.
- 3 Select a component, right-click, and then select **Convert Rose Component**.

**Note:** The **Convert Rose Component** menu command is available only when the selected component contains Rational Rose data.



- 4 Click **Ok**.



- 5 Edit the data for the selected component.

The **Inclusion Paths** tab shows a list of inclusion paths including those currently part of the selected component, as well as those retrieved from the Rational Rose properties. You can change the inclusion paths by adding and deleting items, and by changing their order.

The **Libraries** tab shows a list of libraries currently included in the selected component. You can change the libraries by adding and deleting items, and by changing their order.

The **Classes** tab shows a list of assigned model elements to display. You cannot modify this list from within the wizard.

- 6 After modifying the data, click **Ok** to save the changes, or click **Apply** to select another component.

## Exporting a File

---

You can export packages, classes, components, and use cases into .rtptl files. Exporting these types of objects allows you to import these files into other models.

### To export an element in a model:

- 1 On the **Model View** tab in the browser, select the element export in the model browser.
- 2 Right-click and select **File > Export**.

**Note:** You cannot export diagrams by themselves because they belong to the package they are defined in. Also, do not export Services Library shared packages. For more information, see *Exporting Controlled Element From Model To File* in the *Guide to Team Development - Rational Rose RealTime*





## Contents

This chapter is organized as follows:

- *Fundamentals of Source Control in Rational Rose RealTime* on page 401
- *Using Source Control in Rational Rose RealTime* on page 402
- *Source Control Settings* on page 404
- *Adding Elements to Source Control* on page 414
- *Options for Obtaining Change Management Information When Loading a Model* on page 417
- *Checking Out Files When a Newer Version Exists* on page 422
- *Controlling a Unit with an Uncontrolled Parent* on page 425
- *Viewing the ClearCase Version Tree for a VOB* on page 426

## Fundamentals of Source Control in Rational Rose RealTime

---

Rational Rose RealTime provides source control facilities by integrating with existing source control systems, such as Rational ClearCase, to provide versioning and controlled access to model files. Source control systems are the repositories that store successive versions of files, usually with a comment attached to each version. Before a repository can begin keeping track of a file's versions, the file must be added to the repository.

**Note:** Prior to placing Rational Rose RealTime models under source control, there are some setup steps that must be followed to configure the source control system to allow proper integration with Rational Rose RealTime. Most of these tasks are performed outside of Rational Rose RealTime and require knowledge of the source control tools that you will be using. If you are unsure about the procedures, see your source control tools documentation.

Users of a source control system typically have their own local working area that stores a copy of the files from the repository that they access. Although a repository may contain thousands of files, each user's working area only needs to be populated with the files from the repository that they will be accessing.

If a file is checked out to a user's working area, it will be write-enabled. If the file is not checked out, it will be read-only. To prevent multiple users from attempting to make changes to the same file simultaneously, exclusive access is usually enforced. This is accomplished by allowing only one user at a time to check out a file version. In addition, some source control systems only allow the most recent version to be checked out.

## Using Source Control in Rational Rose RealTime

---

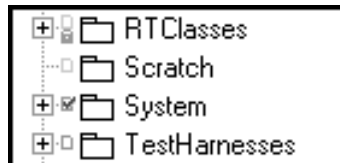
When source control is enabled, Rational Rose RealTime queries the active source control system for the status of each controlled unit. For each unit, the status indicates whether the corresponding file is present in the source control system, and if present, indicates whether the file is checked out to a specific user.

If a unit's file is checked out from source control, the element on the **Model View** tab in the browser shows a check mark next to the unit (Figure 92). The **Unit Information** tab in the **Model Specification** dialog for a unit shows whether the unit is under source control. This status is also visible in the browser. Units that are under source control are shown in the browser with a darkened controlled unit indicator, and units that are not under source control are shown with a faded controlled unit indicator.

Figure 92 shows the different source control status options displayed in the Rational Rose RealTime browsers:

- The light gray unit box opposite **RTClasses** and **Scratch** indicates that they are both controlled units, but are not currently under source control.
- The check mark in the **unit** box opposite **System** indicates that it is a controlled unit under source control and is checked out to the current user.
- The empty gray unit box opposite **TestHarnesses** indicates that it is a controlled unit under source control and is not checked out to the current user.

**Figure 92** Controlled Unit Icons with Source Control



For information on performing an unreserved checkout, see *Performing an Unreserved Checkout* on page 415.

For information on moving model elements that are currently under source control, see *Moving Model Elements* on page 372.

## Maintaining Integrity When a Model is Under Source Control

Changes to a model sometimes require that several model elements be modified to effect the change. Some edits, such as element name changes and hierarchy manipulations, may require modifications to every reference of the element.

Updating all of the cross-references is not necessary to maintain the model's integrity - only direct references must be updated (such as the reference from a derived class to its superclass). Although model integrity is maintained in this way, code generation may not work properly unless all references are also updated.

Due to the many cross-references in a model, it is often unfeasible to check out all of the units that are affected by an edit so that all references can be updated immediately. Rational Rose RealTime enforces the rule that only those elements required for an edit must be accessible to allow the edit to proceed. The changes affecting these required units is called a **primary** edit. If these units are not accessible and cannot be checked out, then Rational Rose RealTime will not allow the edit to proceed.

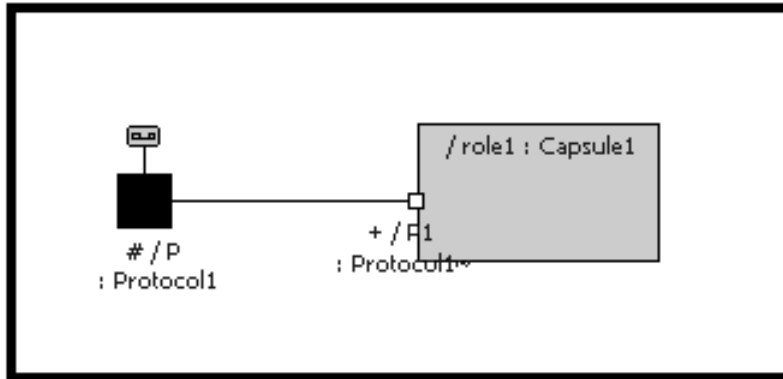
All other changes, such as references that will be modified as a result of the edit, are called **secondary** edits. Rational Rose RealTime prompts you to check out secondary edit units after the operation ends.

**Note:** It is important that secondary edits be updated as soon as possible. Otherwise model validation problems may arise.

### Example

Figure 93 shows the Structure diagram for a simple model.

**Figure 93 Model Validation Example**



In this example:

- **Capsule1** is a primary edit and it must be checked out to proceed with the edit.
- **Capsule2** is a secondary edit and it should be checked out but, if not, the edit can proceed.

If **Capsule2** is not checked out and the edited **Capsule1** is checked in to source control, users who open a model with those versions of **Capsule1** and **Capsule2** will encounter a model validation error that corresponds to the deletion of the connector in **Capsule2**.

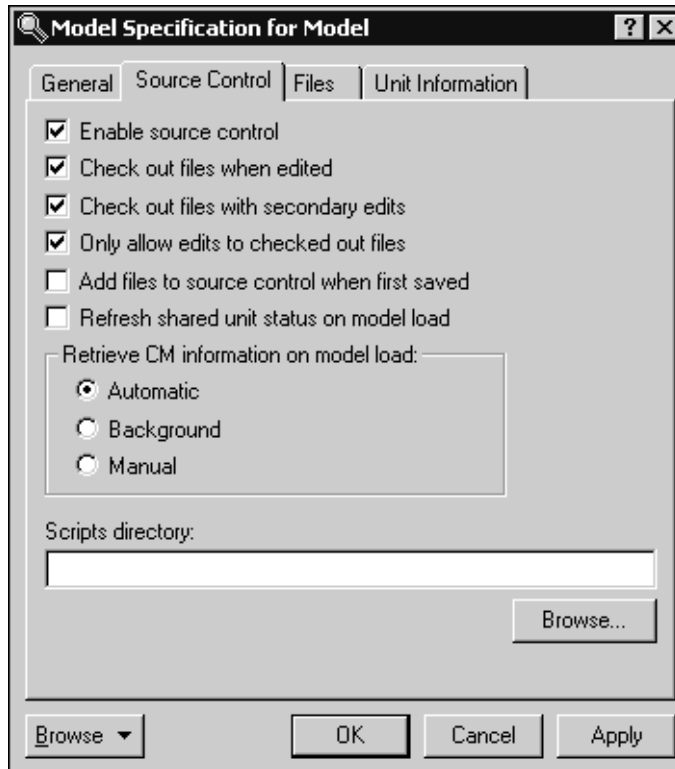
If you delete port **P1** from capsule class **Capsule1**, the port role **P1** on capsule role **role1** in **Capsule2** will also be deleted. This, in turn, would cause the connector to be deleted in **Capsule2**.

## Source Control Settings

---

All source control settings are stored in the workspace file. Source control settings are located in the **Source Control** tab in the **Model Specification** dialog. Alternatively, you can access the **Source Control** tab by clicking **Tools > Source Control > Configure**.

**Figure 94 Model Specification Dialog**



**Enable source control**

Allows for the checking in and checking out of model elements from a source control system.

**Check out files when edited**

Automatically checks out a model element from your source control system if you attempt to edit it.

**Note:** Select this option if the model is under source control. If this option is not selected, you may have difficulty saving the changes you made, which can also lead to problems when building your model.

### **Check out files with secondary edits**

Automatically checks out a model element from your source control system if an edit to another model element causes a change in the element.

**Note:** Select this option if the model is under source control. If this option is not selected, you may have difficulty saving the changes to the affected model elements, which can also lead to problems when building your model.

### **Only allow edits to checked out files**

Prevents edits to model elements unless the element is checked out.

It is recommended that this option be selected if the model is under source control. If this option is not selected, then you may have difficulty saving your changes, which can also lead to problems when building.

### **Add files to source control when first saved**

Causes all model elements to be placed in the source control when the model is saved.

This option is not usually selected. Instead, use the **Tools > Source Control > Submit All Changes to Source Control** command when submitting additions/changes.

### **Refresh shared unit status on model load**

Indicates whether the Toolset refreshes the source control status of shared controlled units when a model is first loaded.

Clearing this option can significantly improve the time it takes to open a model with source control selected. The status of a unit can always be refreshed later, if required.

### **Retrieve CM information on model load**

Specifies how you want to obtain CM information when loading a model.

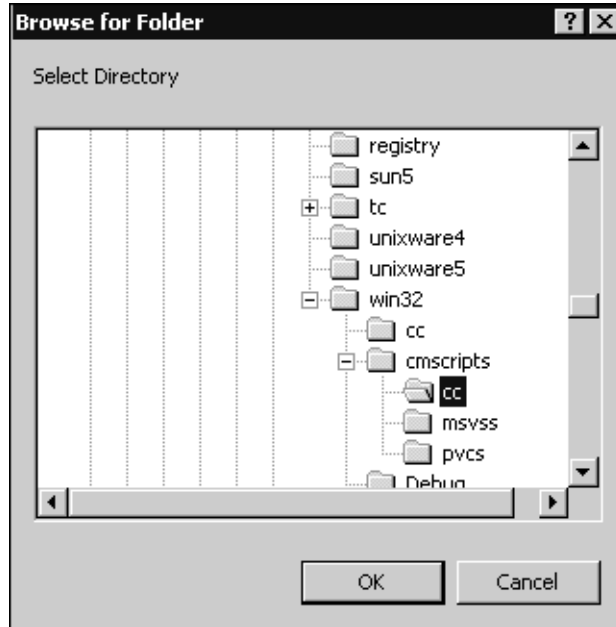
For additional information on CM options, see *Automatic Mode* on page 420, *Background Mode* on page 421, and *Manual Mode* on page 421.

## Scripts Directory

When working with source control, Rational Rose RealTime must know the location of the scripts that interface with your source control tool.

Click **Browse** to select the directory that contains the appropriate scripts.

**Figure 95 Browse for Folder Dialog**



The **Browse for Folder** dialog shows a subdirectory for each of the supported source control tools. Depending on your configuration (UNIX or Windows), the directory names corresponding to the source control systems directly supported by Rational Rose RealTime will appear.

**Note:** Source control interface scripts are located in `$ROSET_HOME/bin/<host platform>/cmscripts`.

The available directory names are:

- **cc** - Rational ClearCase (UNIX and Windows)
- **msvss** - Microsoft Visual SourceSafe (Windows only)
- **pvcs** - Professional Version Control System (Windows only)
- **rccs** - Revision Control System (UNIX only)
- **sccs** - Source Code Control System (UNIX only)

**Note:** You can use pathmap variables in the **Scripts directory** box. When saving the workspace, the pathmap variable **CM\_ScriptPath** will contain this directory.

```
[General]
ModelFile=t2.rtmdl
Version=Rational Rose RealTime 6.5
DefaultLanguage=Analysis
DefaultRTS=C++ TargetRTS
[CMConfiguration]
CM_EditOnlyCheckedOut=Yes
CM_CheckOutOnEdit=Yes
CM_CheckOutAfterTouch=Yes
CM_AddOnFirstSave=No
CM_RefreshSharedUnitStatusOnLoad=No
CM_ProviderType=CM_ProviderType_Script
CM_ScriptPath=$ROSE_HOME\bin\win32\cmscripts\fsrw
```

## Optimizing Performance

The **BatchSize** and **SupportsFileArgs** options within the **cm\_getcaps** script enable you to optimize performance.

### BatchSize

You can now increase the **BatchSize** option that was previously set at 20 files. The default is 30000, and the minimum is 1.

**Note:** If your CM scripts generate errors resulting in an inability to execute a command in Rational Rose RealTime, decrease the **BatchSize** value.

### SupportFileArgs

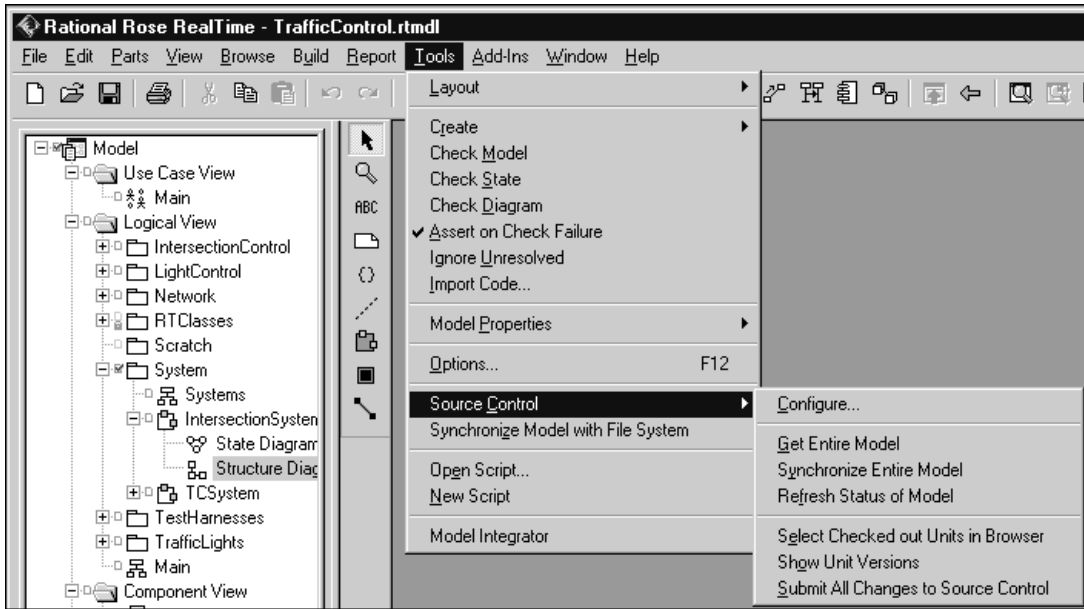
You can change the **SupportsFileArgs** option to **False** from the default setting of **True**. When working with a large number of files, setting this option to **False** allows you to bypass the command line by passing a temporary file which contains the command-line arguments.

## Accessing Source Control Operations

In Rational Rose RealTime, you can access source control operations by clicking **Tools > Source Control** (Figure 96). These operations generally apply to all controlled units in the entire model, and include several add-in helpers and convenience operations.

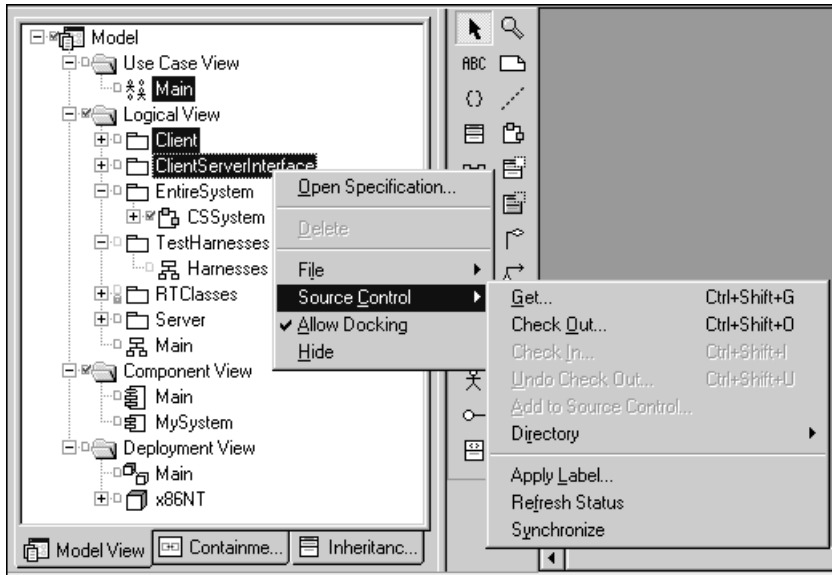


**Figure 96 Tools > Source Control Menu**



Alternatively, you can access source control operations using context menus in browsers. When you select a controlled unit from the browser, the context menu contains source control operations. To apply an operation to multiple units at the same time, select all the desired units, and access the source control operation through the context menu (Figure 97).

**Figure 97 Source Control in the Browser Context Menu**



## Source Control Operations

The following source control operations are available from within the Rational Rose RealTime toolset with all supported source control systems:

- Refresh Status
- Synchronize
- Get
- Check Out
- Uncheckout
- Add
- Check In
- Submit All Changes
- Apply Label
- Show Differences
- Show History

**Note:** Some operations are handled slightly differently for some source control systems. Unless otherwise indicated, these operations are all enabled for any selection of units.

## Refresh Status

Queries the active source control system for each unit selected and determines whether the unit's file is under source control. If the file is under source control, this operation determines whether the file is checked out. Refreshing the status does not retrieve new versions of files, nor does it reload files if they have been changed outside the toolset.

## Synchronize

Performs the same status updating that the **Refresh Status** operation performs. The Synchronize operation also determines if the file on disk has changed since the file was loaded into the toolset. If the underlying file has changed, it is reloaded into the toolset.

**Note:** For Rational ClearCase, if a dynamic view is used and the version of a file available in the view changes, **Synchronize** detects the changes and reloads the file. **Synchronize** is a safer operation than a **Get** because Synchronize will not lose any checked out changes, while **Get** may replace your checked out changes with the most recent version in the VOB.

## Get

Interfaces with the active source control system and requests the latest version of the files corresponding to the selected units. If a new version is retrieved, Rational Rose RealTime reloads the file.

**Note:** For Rational ClearCase, **Get** does not retrieve a specific version of a file to a view because the version being observed in a view can only be changed using the **config spec** for that view. However, if a file is checked out, you can use **Get** to replace the checked out file with a copy of a particular version of the file. If a file is not checked out, performing a **Get** on that file is the same as performing a **Synchronize** on the file.

## Check Out

Prompts your source control system to lock the specified files so that you can modify them, and then submit a new version using **Check In**. If the specified file is currently checked out to another user, the check out operation fails. **Check out** retrieves the latest version of the files being used.

**Note:** For Rational ClearCase, when working with a snapshot view, ClearCase marks elements in the VOB as being checked out. When checking out an element, you are not warned if a more recent version exists in VOB.

## Uncheckout

Removes the lock that the user holds on the file in the source control system, and replaces their local file with the most recent file from the repository. **Uncheckout** is available for any file that is currently checked out to the current user.

## Add

Attempts to place the selected units under source control. After a unit is added to source control, it can be versioned using **Check Out** and **Check In**. Unless a file must be added to source control without submitting other changes at the same time, use **Submit All Changes** rather than explicitly clicking **Add**.

**Note:** For Rational ClearCase, when adding files to source control, the ClearCase integration assumes that the containing directory is under source control and is not currently checked out. If the containing directory is already checked out, the **Add** operation will fail.

## Check In

Submits a checked out file to the repository so that a new version is stored. Unless a file needs to be checked in without submitting other changes at the same time, use **Submit All Changes** to submit changes to the repository.

**Note:** For Rational ClearCase, when checking in files, ClearCase copies the new version to the VOB, as long as there is no successor version already in VOB. If there is a successor, an error is returned from the scripts and will appear in the **Log** tab in the **Output** window. To check in your changes, you must first merge the most recent version from the VOB into your local copy.

To update your snapshot, click **Tools > Source Control > Update Snapshot View**. The **Update Snapshot View** command helps you merge any changes. This is the preferred method since your snapshot view will also get any new elements that appear in the VOB.

**Note:** If you know that only one element has changed in the VOB, select that element from the browser and use the context-menu **Source Control > Get** command to retrieve the most recent version. Then perform the merge.

## Submit All Changes

Performs the following actions:

- Determines which units are not under source control, and prompts the user to add them.
- Determines which units are checked out from source control, and prompts the user to check in the units.

After **Submit All Changes** is successfully completed, the repository is updated with all changes made by the user.

**Note:** This command is only available from **Tools > Source Control**.

## Apply Label

Instructs the source control system to apply a specified label to the selected units. Directories may also be labelled with the option of working recursively on the directory contents.

For Rational ClearCase, labelling a directory only applies the label to the directory element itself. To apply the label to the files contained within a directory, use the **Recursive** option.

## Show Differences

Compares the local version of a unit with the latest version stored in the source control repository.

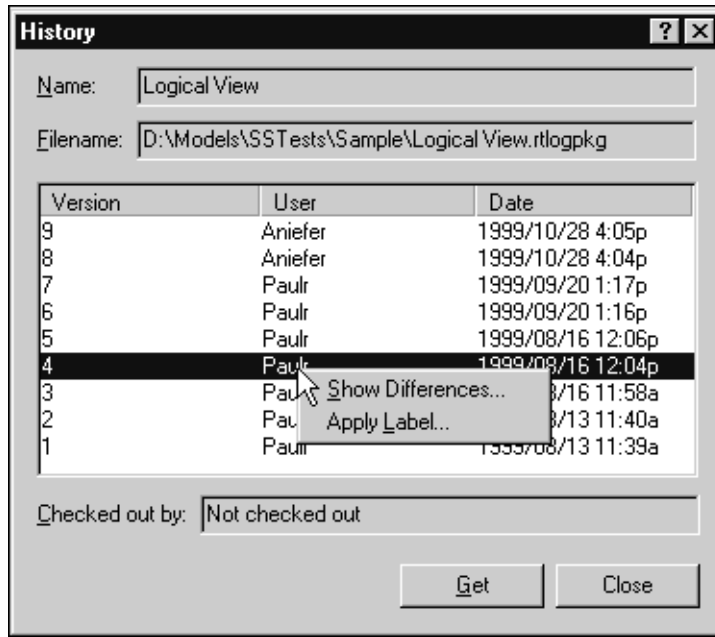
For details on using the Merge Differencing tool, see the *Rational Rose RealTime Model Integrator* documentation.

**Note:** **Show Differences** is only enabled when a single unit is selected.

## Show History

Displays the version history of a unit based on the revisions of the file that are in the source control repository.

**Figure 98 History Dialog**



Most source control systems support the retrieval of a specific version of a file. In these systems, the **Get** button is enabled when a version is selected in the list.

To compare the local version of the unit with a specific version, right-click on the version to compare, and click **Show Differences**.

For source control systems that support applying a label to arbitrary versions of elements, the context menu will also include the **Apply Label** command.

**Show History** is only enabled when a single unit is selected.

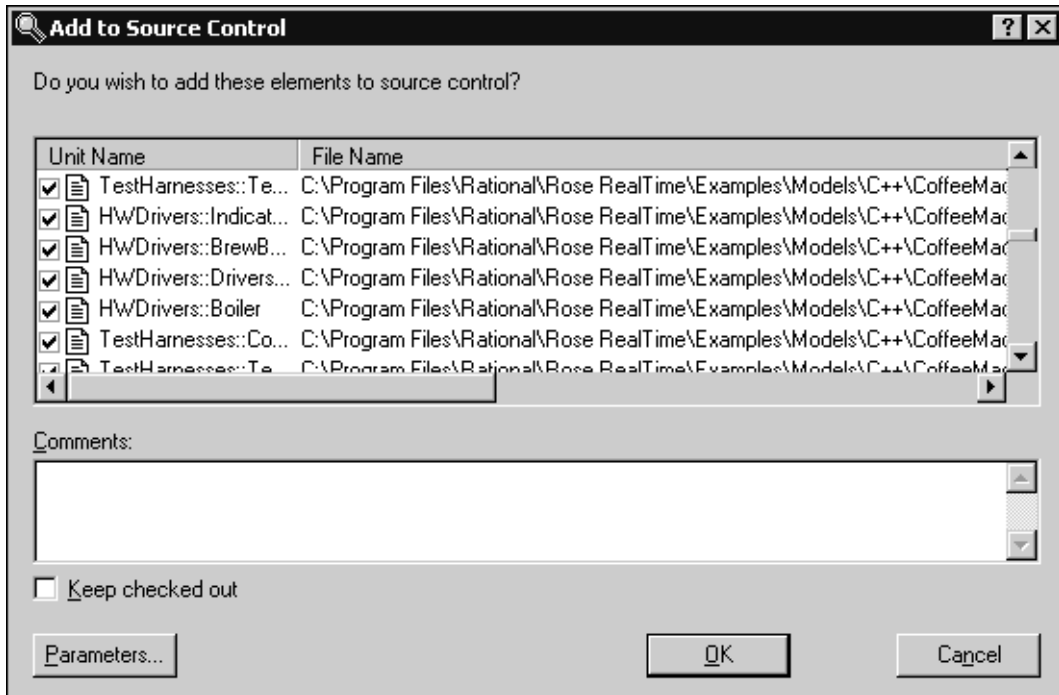
## Adding Elements to Source Control

---

You want to ensure that you add all appropriate units to your source control system. Forgetting to add new units can result in model validation errors when other users obtain the new version of the units.

Figure 99 shows the **Add to Source Control** dialog where Rational Rose RealTime prompts you to add any new units to source control that it has detected.

**Figure 99 Add to Source Control Dialog**



By default, all new and checked out units are submitted. You can use the check boxes on the left side of each unit to filter items from the list.

### **Keep checked out**

Automatically checks out the units from the list after they have been added to source control.

### **Parameters**

Displays the **Parameters** dialog where you can specify parameters for the items.

## **Performing an Unreserved Checkout**

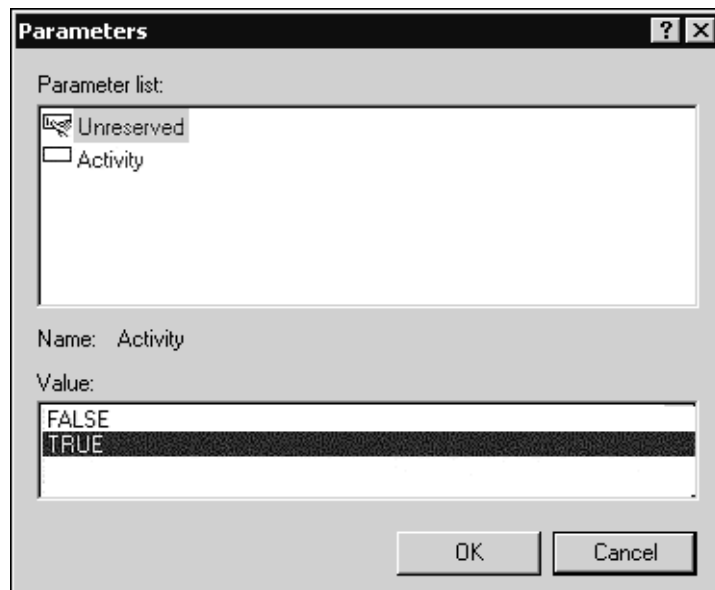
You can perform unreserved checkouts in Rational Rose RealTime when using Rational ClearCase as your source control system. Unreserved checkouts are useful when the changes you need to make are only temporary. For example, if a designer wants to modify an element but does not intend to submit the changes, an unreserved checkout allows you to complete the required work without stopping other designers from checking out that element. Another scenario for using an unreserved checkout occurs when two or more designers want to work on the same element at the same

time. The first designer would have a reserved check out, and all other designers would specify that they want an unreserved checkout. Those designers with an unreserved checkout will not be able to check in their changes until the designer with the reserved checkout has submitted changes, or performed an **Uncheckout**.

**Note:** After the designer with the reserved check out submits their changes, those designers that have an unreserved checkout will be required to perform a merge. Rational Rose RealTime will notify you and walk you through the procedure for performing a merge.

**To perform an unreserved check out:**

- 1 Right-click on an element, and click **Checkout**.
- 2 Click **Parameters**.



- 3 Change the value of the parameter to **True**.
- 4 Click **OK**.



## Options for Obtaining Change Management Information When Loading a Model

---

A model is stored in unit(s) that may be under Change Management (CM) control (also known as Source Control Integration). When loading a model, the Rational Rose RealTime toolset attempts to obtain the CM status of its units. CM information indicates whether a unit is under CM control, if it is currently checked out, and, depending on your CM capabilities, identifies the version of the unit. Because it takes time to obtain this CM information when loading a model, you may want to specify when this CM refresh occurs.

If CM integration is enabled for a model, you can specify additional options when the retrieval of CM information occurs: **Automatic Mode** (default), **Background Mode**, and **Manual Mode**.

Figure 100 shows the **Model View** tab when the units have no CM information specified.

**Figure 100 Model View with no CM Information**

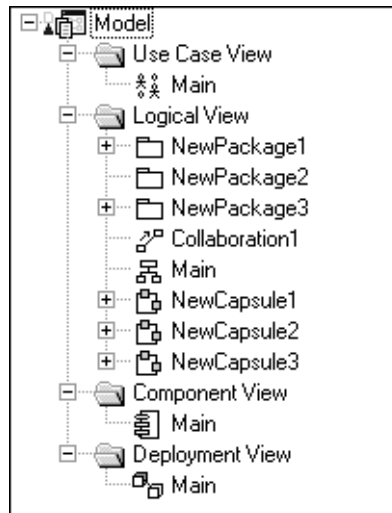
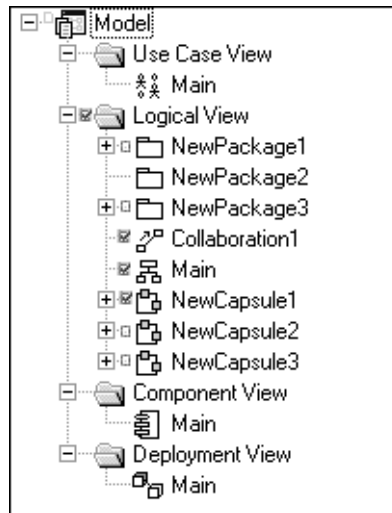


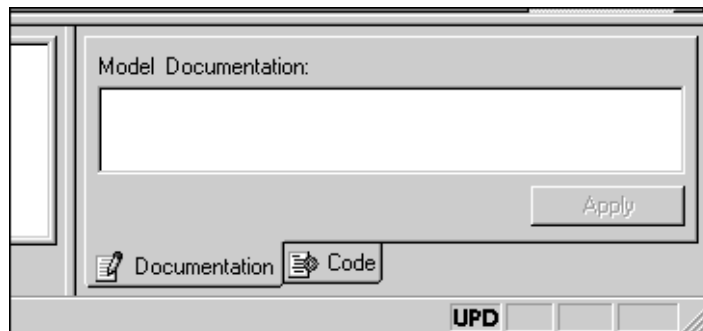
Figure 101 shows the **Model View** tab when the units have CM information.

**Figure 101 Model View with CM Information**



A new Status Bar indicator, **UPD**, identifies when a **Background** refresh occurs; for **Automatic** and **Manual** modes, the Status bar indicator is blank.

**Figure 102 UPD Status Bar Indicator**



## Updating the Log

When a **Background** refresh starts, a message is added to the Rational Rose RealTime log indicating that the retrieval of CM information has started:

```
10:31:39| Starting background refresh of CM information.
```

When a **Background** refresh is completed, a message is added to the Rational Rose RealTime log indicating that the retrieval of CM information has been completed:

```
11:42:39| Finished background refresh of CM information.
```

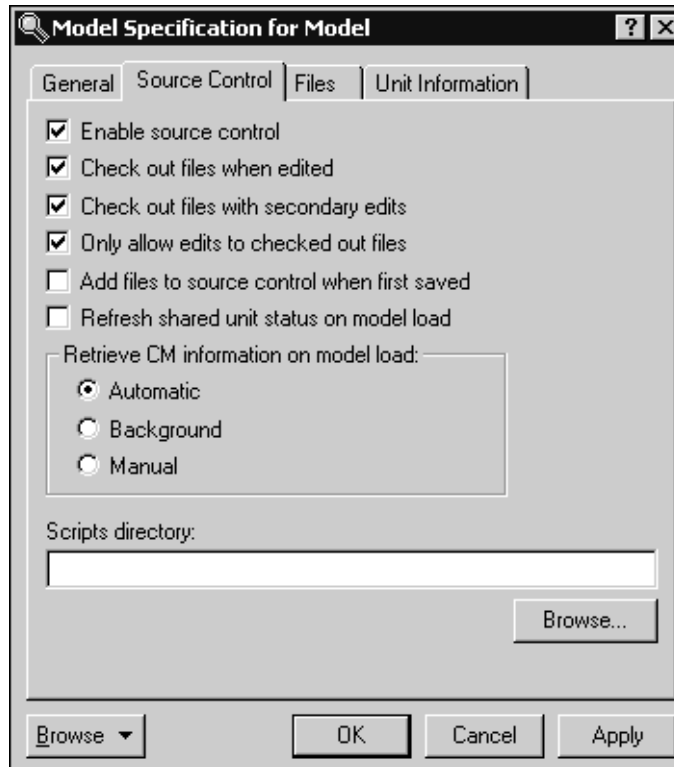
**Note:** If you select **Automatic** or **Manual** mode, the Rational Rose RealTime log is not updated.

## Changing the CM Retrieval Option

If a model has CM integration enabled, you can specify when the retrieval of CM information occurs: **Automatic Mode** (default), **Background Mode**, and **Manual Mode**. Because it takes time to obtain CM information when loading a model, you may want to specify when this CM refresh occurs.

To specify how to retrieve CM information when loading a model:

- 1 In the **Model View** tab in the browser, right-click on **Model**.
- 2 Click **Open Specification**.



**Note:** To enable the CM options, the **Enable source control** option must be selected. By default, **Automatic** is selected. The CM option specified in this dialog is stored in the workspace file and is used during the next model load.

## CM Retrieval Options

In the **Retrieve CM information on model load** area, you have three options for obtaining CM information when loading a model.

### Automatic Mode

When selected, a non-interruptible CM refresh occurs immediately after the model loads in the toolset.

**Note:** The toolset remains locked until the CM refresh activities complete.

## Background Mode

When selected, all CM activities are performed when there is no user activity (during the application's idle mode) and you will be unaware of the retrieval of CM information.

**Note:** To modify an element in the model before the **Background** mode obtains its corresponding CM status for that specific element, right-click on **Model** in the browser, then click **Source Control > Refresh Status** before you modify the element.

The **Source Control** submenu also contains a **Refresh Status (with Child Units)** option. When selected, this option performs a recursive refresh of the child units for the selected units in the model. This option provides you with the granularity to refresh the entire package, rather than refreshing each individual item.

**Note:** Both the **Refresh Status of Model** and **Refresh Status (with Child Units)** options support multi-select.

## Manual Mode

When selected, no CM information is obtained when loading a model. This means that after the model loads, there is no additional delay because there is no CM information being retrieved at this time. When using **Manual** mode, before performing modifications to elements in the model, you must right-click on **Model** in the **Model View** tab, then click **Source Control > Refresh Status**, or **Source Control > Refresh Status (with Child Units)**.

## Limitations

Workspace files are not backward compatible because the new options are stored in the workspace file.

If the Change Management operations are slow, using **Background** mode may reduce the responsiveness of the Rational Rose RealTime application (during the **Background** refresh). For this situation, we recommend that you use **Automatic** or **Manual** mode.

A **Background** refresh may not obtain CM information for a unit if its parent was edited before the CM information was obtained for the child unit. In this case, you must manually refresh the CM information by right-clicking on **Model** in the **Model View** tab, and then clicking **Source Control > Refresh Status**.

## Checking Out Files When a Newer Version Exists

---

If you make a change that requires the Rational Rose RealTime toolset to automatically check out one or more files, the toolset performs a "get" of the latest version in your CM tool before checking out the file and then the toolset compares it with the current version. If the version from your CM tool is newer, the toolset prompts you with the following warning:

```
The units in source control are more recent than the current
version. If your checkout retrieves the latest version it
will need to be imported, and your current operation will be
lost.
```

```
Do you wish to continue with the checkout?
```

If you click **No**, the checkout is canceled and your changes are not applied.

If you click **Yes**, you may lose your changes. To ensure that you do not lose any changes, set the **Don't get a local Copy** option to **True** on the **Parameters** dialog by clicking **Parameters** on the **Checkout** dialog.

**Note:** If you explicitly requested a checkout, this type of version checking is not performed.

## Get Dialog

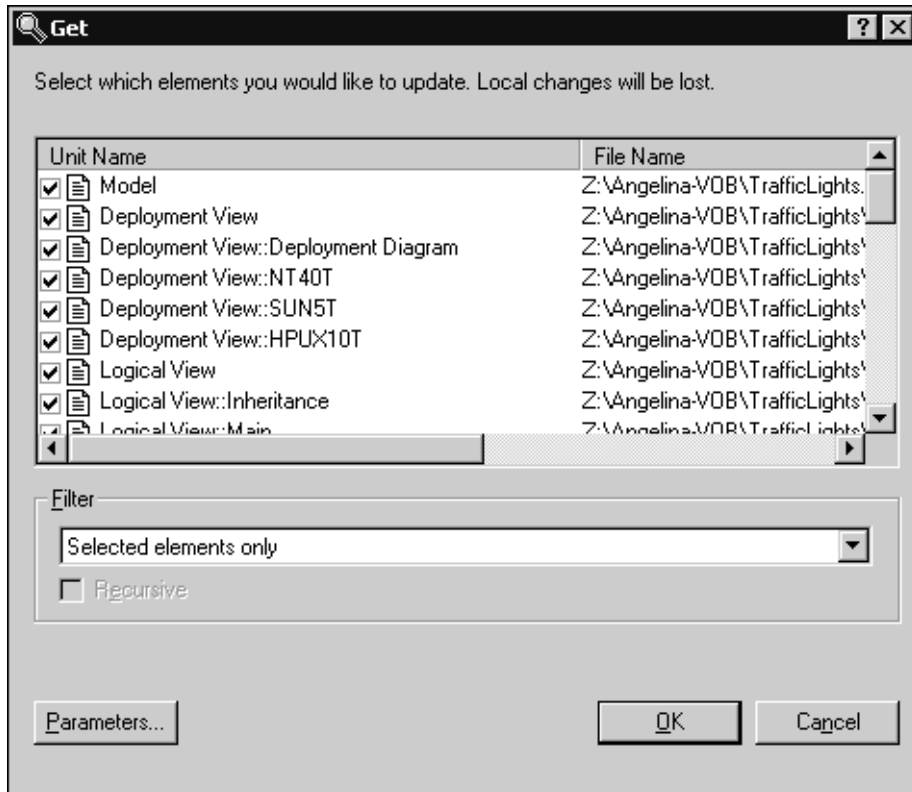
---

A "get" interfaces with your active source control system and requests the latest version of the files corresponding to the selected units. If a new version is retrieved, Rational Rose RealTime reloads the file.

Use the **Get** dialog to request files from your CM tool (Figure 103). To access the **Get Entire Model** dialog, click **Tools > Source Control > Get Entire Model**.

**Note:** The **Get Entire Model** option is not available for the current selection if it does not contain units that are under source control.

**Figure 103 Get Dialog and Get Entire Model Dialog**



**Unit Name**

Shows all the elements in the current model that you can update. Items that are not under source control do not appear in this list.

**File Name**

Specifies the location of an element.

**Filter**

Specifies the level of refinement for getting elements from your source control tool.

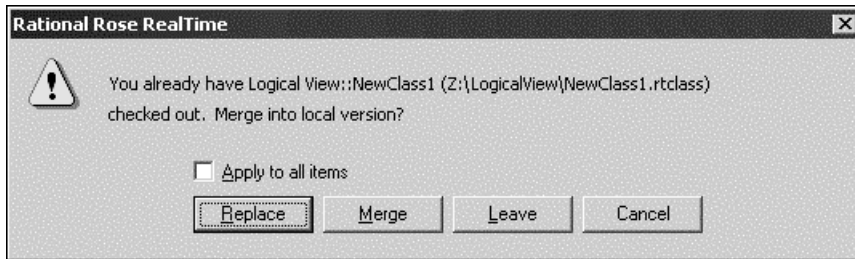
**Recursive**

When selected, it indicates that the "get" command applies to current or specified elements, and to any child controlled units.

## Parameters

Displays the **Parameters** dialog where you can specify parameters for the items.

If you perform a get on a checked out item for which a more recent version exists in source control, the following dialog appears:



### Apply to all items

Applies the current selection (**Replace**, **Merge**, or **Leave**) to all subsequent files that would have prompted this dialog.

**Note:** Selecting **Apply to all items** has no affect if you click **Cancel** because the "get" operation will abort.

### Replace

Replaces your existing file with the version in source control.

### Merge

Merges the version from source control with your current version.

### Leave

Leaves this file alone and does nothing.

### Cancel

Cancels the get operation. Any gets or merges done prior to clicking **Cancel** will remain intact.



## Controlling a Unit with an Uncontrolled Parent

---

In Rational Rose RealTime, you can change a shared unit to the unit owned by the model (and vice versa), and allow a unit to be controlled when its parent is not a controlled unit. Controlling a unit in an uncontrolled parent allows you to perform some temporary action without having to clean up additional directories created when directories are controlled units.

### Changing Unit Ownership

You can easily identify the ownership of a unit by observing the following:

- If a tiny lock icon appears to the left of the unit name on the **Model View** tab in the browser, the unit is owned by the model.
- If the **Owned by model** option in the **Unit Information** tab of the selected unit's Specification is selected, the unit is owned by the model.

**Note:** The **Unit Information** tab displays only on the **Specification** dialogs for controlled units.

**Note:** Changing ownership has non-trivial consequences to the underlying file structure; and you should do this only when absolutely necessary (such as when changing the ownership to move a shared package to a different location). Do not attempt to change an owned unit to non-owned when this unit has unsaved modifications. Rational Rose RealTime will not save a non-owned unit and your modifications will be lost, possibly resulting in model inconsistencies.

#### To change the ownership:

- When units are not owned by a model, you can right-click in the **Model View** tab in the browser, and click **File > Share External Package**.
- Select the **Owned by model** option in the **Unit Information** tab of the selected unit's Specification.

Changing the ownership alters the containing unit of the unit being changed, while the unit that you changed the ownership for remains unchanged. As a result, the containing unit is marked as modified (a blue delta, ▲, in the **Model View** tab in the browser).

## Limitations

- Changing a non-owned unit to an owned unit:  
When saving an owned unit, you must save it to a new location based on the file location of the containing unit.
- Changing a non-modified owned unit to a non-owned unit:  
You cannot save units that are not owned by the model. Also, source control operations are not supported for units that are not owned by a model.
- Changing a modified owned unit to a non-owned unit:  
If the model does not own the unit, you cannot save modifications to the unit (including modifications already made and not yet saved).
- A non-owned unit cannot contain an owned unit. Consequently, if you have a tree hierarchy structure of units and you change the top unit in the hierarchy to non-owned, all the units under the top unit also change their status to non-owned. This means that any change in ownership (non-owned to owned, and owned to non-owned) is applied recursively to all containing units in a hierarchy.

## Viewing the ClearCase Version Tree for a VOB

---

ClearCase includes a browser to view the version tree of a VOB element. In Rational Rose RealTime, you can view the ClearCase version tree browser to view the CM history of a specific model element. You can either view a textual or graphical representation of the ClearCase version tree.

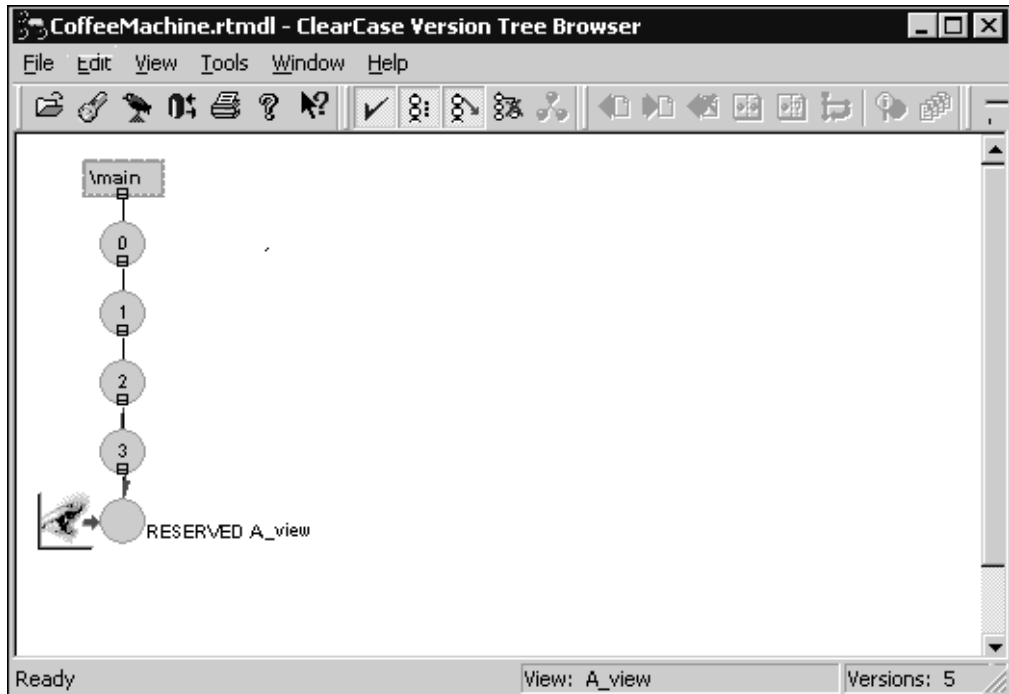
### To view a graphical representation of the ClearCase version tree:

- In Rational Rose RealTime, right-click on a model element, and then click **Source Control > Version Tree**.

**Note:** Because this version tree browser is specific to ClearCase, the **Version Tree** option is not available for other CM systems.

Figure 104 shows a graphical view of the ClearCase version tree on Windows.

Figure 104 ClearCase Version Tree Browser



**Note:** For UNIX, the **ClearCase Version Tree** Browser diagram looks slightly different.

- From the command-line, type the following:  
`cleartool lsvtree -graphical <element_name>`
- Open ClearCase Explorer, and click **Tools > Version Tree**.

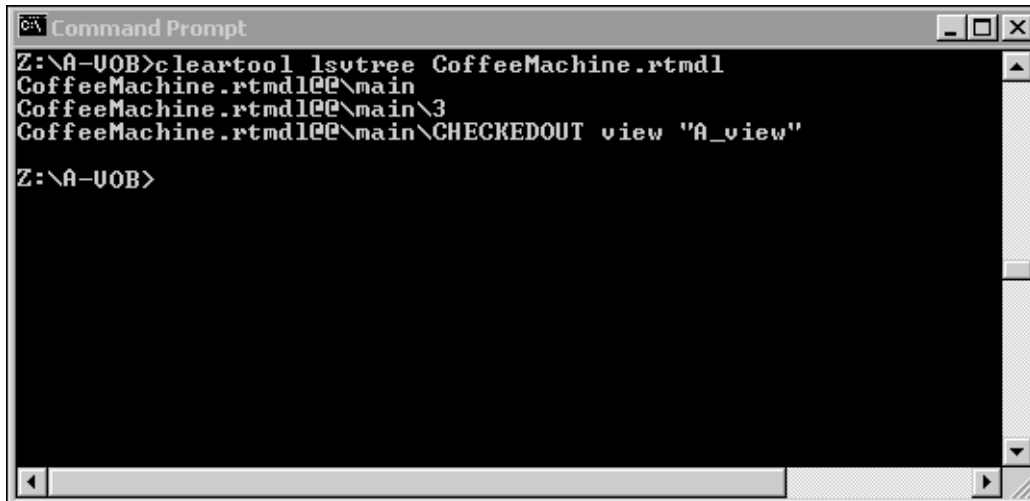
**To obtain a textual view of the version tree:**

From the command-line, type the following:

```
cleartool lsvtree <element_name>
```

Figure 105 shows the text representation of a ClearCase version tree from the command-line.

Figure 105 Text Representation of a ClearCase Version Tree



```
Command Prompt
Z:\A-UOB>cleartool lsvtree CoffeeMachine.rtm1
CoffeeMachine.rtm1@@\main
CoffeeMachine.rtm1@@\main\3
CoffeeMachine.rtm1@@\main\CHECKEDOUT view "A_view"

Z:\A-UOB>
```

## Contents

This chapter is organized as follows:

- *Introduction to Naming Guidelines* on page 429
- *Assigning Names* on page 429
- *Special Case Notes* on page 430
- *Using Logical Names for Model Elements* on page 430

## Introduction to Naming Guidelines

---

Rose RealTime does not support name spaces, so there are a number of names that will be part of the global name space. Be careful not to use the same names for any elements that may conflict. Also, make sure you avoid using reserved names, such as any names from the Rose RealTime Services Library, Language-reserved words (for example, C++), names of common operating system functions or data structures. Spaces in names should also be avoided because some targets do not handle them.

## Assigning Names

---

Each unique model element must have a unique name, and each relationship can be labeled with a word or phrase that denotes the semantics or purpose of the relationship. You can type the name in the diagram or in the Name field in the specification.

- If you type the name in the diagram, your entry is displayed in the Name field.
- If you type the name in the specification, the software displays the new name in the element icon and updates the information in the model.

You can rename an element using one of the following methods:

- Change its name in the diagram.
- Change its name in the specification.
- Change its name in the browser.

For more information about renaming, see the topic *Renaming a Model Element*.

## Special Case Notes

---

Special considerations for naming include the following:

- **class attribute** - Each attribute must be unique within a class.
- **operation** - Omit the function parenthesis when typing the operation name. The software automatically displays the parenthesis when you display the operation in the class compartment.
- **connection** - The name field is optional.
- **state** - State icons that have the same name are assumed to represent the same state if they appear in the same context; otherwise, each state icon is assumed to represent a distinct state. State icons that appear in different state diagrams represent distinct states, even if they have identical names.
- **use case actors, classes, capsules, protocols** - These elements must all have names that are unique. Class names are part of the generated code name space, and must not conflict with each other or with other items in the global name space, for example, global functions, signal names.

## Using Logical Names for Model Elements

---

Users of international character sets have a convenient mechanism for working with multi-byte characters. You can specify an alias shortcut for model elements that display on diagrams. This alternate name is called a **Logical Name**, and it displays on diagrams.

You cannot create the logical name using inline editing directly on the Class diagram. Inline editing only effects the logical name of the class if a logical name (alias) was previously created. If an element does not have a logical name defined, inline editing on the **Class** diagram only updates the class name, and not the logical name. If you

right-click on a **Class** diagram, then click **Filter > Show Logical Name if Defined**, the logical name is not updated. To create a logical name, select the **Alias** tab on the **Class Specifications** dialog, and then create a logical name for the element.

**Note:** Model elements will continue to require an ASCII physical name for code generation purposes.

You can assign a **Logical Name** to the following model elements:

### **For Class Diagrams and Use Case Diagrams**

- Class
- Capsule
- Attribute
- Operation
- Protocol
- Signal
- Package

### **For Collaboration Diagrams and Structure Diagrams**

- Port
- Port Role
- Capsule Role
- Connector
- Classifier role (available on Collaboration Diagrams)

### **For State Diagrams**

- Transition
- Initial point
- Choice point
- State
- Final state (available on State Diagrams for classes)

**Note:** Diagrams will show logical names only for the first level of inclusion. For example, consider a capsule image on a **Class Diagram**: the protocol name on the capsule's port will always display the physical name.

Automation interfaces were extended for **Logical Name** to allow for modification and retrieval of an object's **Logical Name**.

## Logical Name Example

Figure 106 shows the **Class Diagram** for an example model.

**Figure 106 Class Diagram - Example Model**

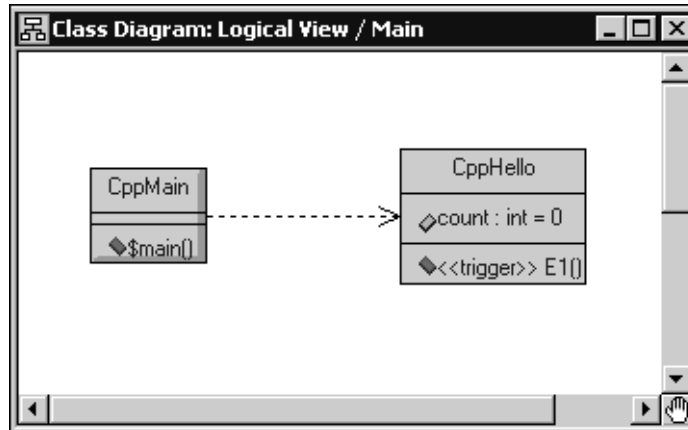


Figure 107 shows the **Alias** tab for the **Class Specification** dialog box for the **CppHello** class.

**Figure 107 Class Specification Dialog Box for CppMain - Alias Tab**





The **Logical Name** box is a text field that allows you to enter non-ASCII characters, spaces, and punctuation marks. There are no restrictions associated with the element's name. Logical names are not used for code generation.

If a logical name is not specified for a model element, the physical name is displayed on the diagrams.

Figure 108 shows the logical name assigned to the **CppMain** class.

**Figure 108 Class Specification - Alias Tab - Logical Name Change**

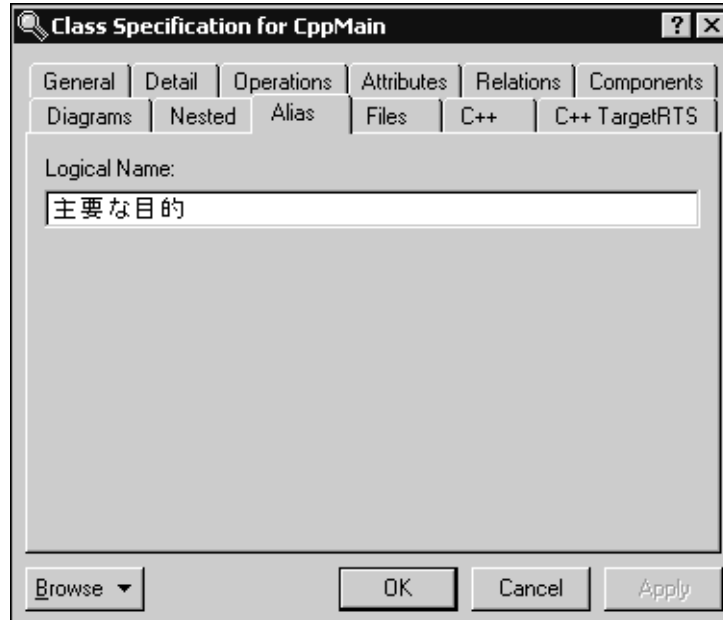
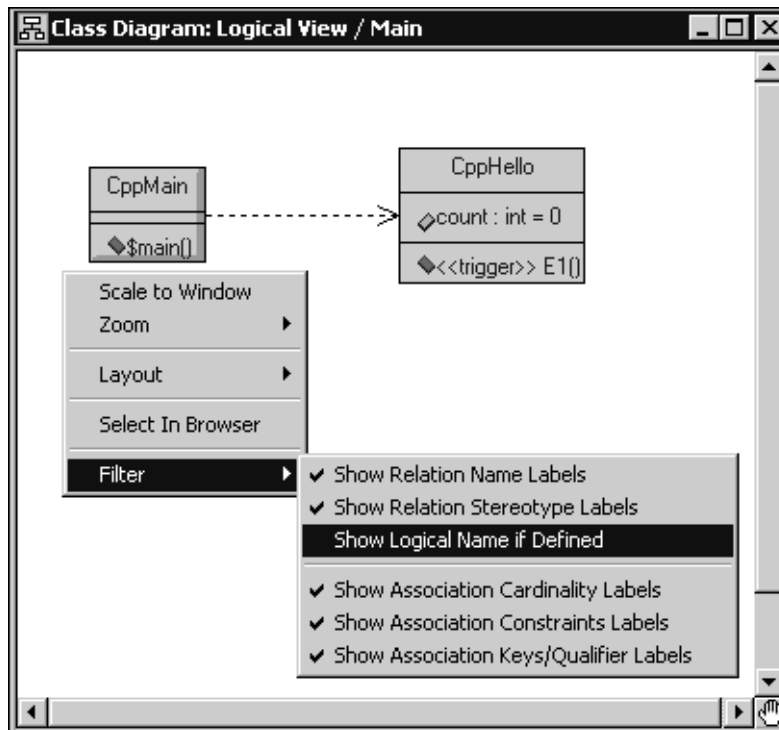


Figure 109 shows the context menu for a **Class Diagram**. To change the display to use logical or physical names in your model, use the **Show Logical Name if Defined** option from the **Filter** submenu.

**Note:** Since the filter option **Show Logical Name if Defined** is selected, you can edit logical names on diagrams in the same way physical names are edited (that is, using inline edit).

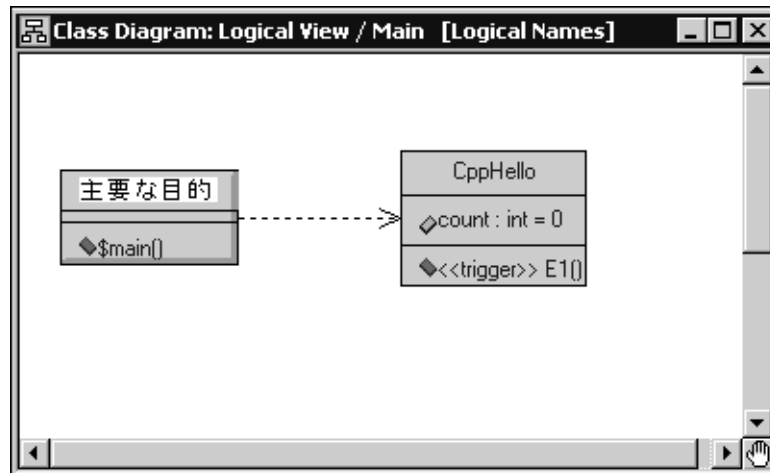
Figure 109 Class Diagram - Show Logical Name if Defined



If the **Show Logical Names if Defined** option is selected, the title for the corresponding diagram will have the string [Logical Names] appended to the end of the name to provide a visual reminder that logical names (not physical names) are being used.

Figure 110 shows the use of the Logical Name specified for this class (Figure 108).

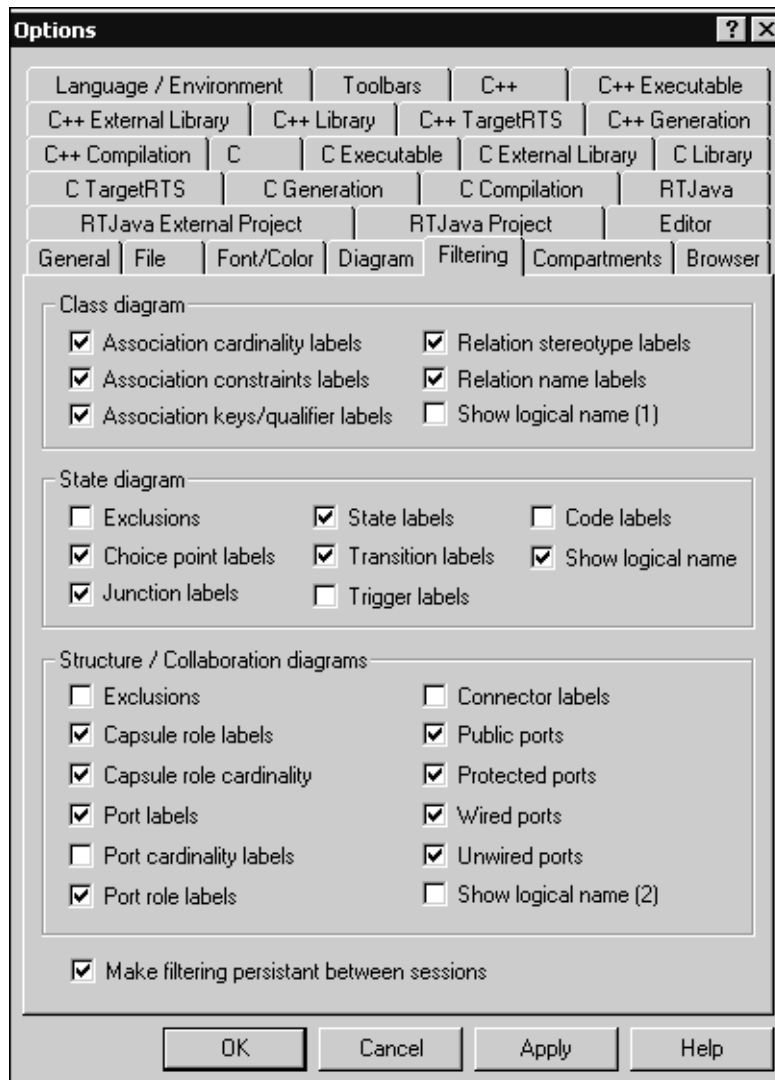
**Figure 110 Class Diagram - Using Logical Names**



The **Filtering** tab in the **Options** dialog box (see Figure 111) shows three **Logical Name** options:

- Class Diagram - **Show logical names (1)**
- State Diagram - **Show logical name**
- Structure and Collaboration diagrams - **Show logical names (2)**

Figure 111 Options Dialog Box- Filtering Tab



## Contents

This chapter is organized as follows:

- *Building and Running Models* on page 437
- *Before You Start* on page 438
- *Building Basics* on page 439
- *Assigning an Active Component* on page 440
- *Creating a Component* on page 441
- *Starting a Build* on page 441
- *Generate Dialog* on page 442
- *Reviewing Build Results* on page 444
- *Opening Code Generated for Model Elements* on page 445
- *Build Menu* on page 447
- *Build Settings Dialog* on page 450
- *Build Log Tab* on page 450
- *Build Errors Tab* on page 451
- *Component Specification* on page 451
- *Generating Documentation Fields* on page 453
- *Component Dependencies* on page 461

## Building and Running Models

---

The mapping from design - that is, classes and capsules - to source code and executables is not an easy task. It is during this phase of the software development process that the majority of errors are introduced into a system, especially when it is done manually. There is always a risk that the implementation will diverge from the original design, and in most cases that is exactly what happens. However, since the UML has well-defined semantics, Rose RealTime can automatically generate a model or design into a lower-level language and then compile it into an executable. With automatic total source code generation of your design, the model becomes the system.

## Is Rational Rose RealTime a Compiler?

The answer is yes and no. Rational Rose RealTime compiles models into a high-level language representation. It generates source code, or complete implementations, of models while the generated source code is compiled and linked into machine language using an external compiler and linker. The result is an executable that can be run and observed via the Rose RealTime toolset.

## Real-Time Services (Services Library)

Behavior in a model is specified using a State machine, and communication patterns are specified with capsule structure. When a model is built, these abstractions must be converted to implementation. Normally, you would have to implement your own state machine, inter-process communication, concurrency control, thread management, timing, and debugging capabilities. However, Rose RealTime provides a set of pre-compiled Services Libraries for different platforms, which provides this functionality for you. In summary the facilities provided by the RealTime Services Library are:

- The mechanisms that support the implementation of concurrent communicating state machines
- Thread management and concurrency control
- Timing
- Inter-thread and inter-process communication
- Observability and debugging of a running model

## Before You Start

---

To allow models to be built and executed on a variety of platforms with different tools (for example, on Windows NT with Visual C++ 6.0 or on Solaris with gcc 2.8.1), Rose RealTime allows build settings to be fully configurable. However, even though complicated build and execution configurations can be setup, there are also default settings that can be used to build and execute less complicated models.

To learn more about building and executing models select a basic or more advanced topic from the list below:

## Building

- **Building Basics** - helps you build your first model
- **Creating a Component** - using component aggregation

## Executing

- **Execution Basics** - Helps you run and observe your first model.
- **Loading and Running Component Instances on Embedded Targets** - Target loading, restarting, and resetting.
- **Overview of Observability Options** - Watches, traces, sequence diagrams, behavior breakpoints, logging output, source code break points.
- **Running from Outside the Toolset** - Running a model without immediate observation, attaching to a running model.

## Building Basics

---

Before trying to build a model, it is important to understand the role of components for modeling the physical aspects of a system. The physical elements of a model refer specifically to source code and executables.

A component is always created with a default configuration for your host machine. This includes a default compiler, compiler flags, linker, and so forth. In many cases these settings are sufficient for building simple sets of classes and capsules that do not require integration with external source files, or libraries.

This section leads you through the steps of building a simple model that does not require integration with external files (everything is defined with the toolset). This will help you understand the build workflow without getting into specialized configuration options. After you understand the basic build workflow refer to the Component Wizard for more information on configuring components with advanced build setting.

## Top-level Capsule

Basically any capsule can be built and run. The capsule that you choose to build is called the top-level capsule. It represents the highest scope of the executable that you want to create. All classes and capsules referenced (contained or in a dependency relationship) with the selected top-level capsule, directly or indirectly, will also be compiled.

**Note:** Since any set of capsule and classes can be compiled, you are not required to compile the entire model all the time. The capsule you decide to build may form only a subset of the whole system. This allows for easier unit testing.

- Create a component.
- Build the component.
- Review the build results.

## Assigning an Active Component

---

If you find yourself building and running the same component and component instances often you should configure an active component. When a component is configured as being active the toolbar build icons and menu items become available for easy access to common build and run commands. In addition you can configure which component instances (executables) should be automatically run when the run button is pressed.

In the browser, select **Set As Active** from the context menu

or

- 1 From the **Build** menu select the **Settings** item to open the Build Settings dialog.
- 2 From the **Active Component** combo box, choose a component that will become the active component.
- 3 For information on the other configurable build options shown, see Build Settings Dialog.
- 4 Click **OK**.

**Note:** The build toolbar icons are now enabled and so are items under the **Build Menu**.



## Creating a Component

---

To build an executable of a model, you must first create a component that will be used to manage the build configuration parameters. There are a couple of different ways of creating a component and assigning a top-level capsule.

You can create the component first, then assign the top-level capsule to it later.

### To create a component:

- 1 Select the Component View folder, right-click and from the popup menu choose **New > Component**.

A new component with the default settings for your platform is created.

- 2 Double-click on the default Component diagram, usually called Main, to open it.
- 3 Drag and drop the new component you just created onto the Component diagram.
- 4 Then drag and drop the top-level capsule onto the new component that was added to the component diagram.

**Note:** You can also assign a capsule or class to a component by dragging and dropping the capsule, class, or protocol from the model browser onto the component in the model browser.

- 5 Open the components specification, switch to the References tab and set the top-level capsule.

Alternatively, you can use the Component Wizard to help configure a component. To run the Component Wizard, select **Build > Component Wizard**.

## Starting a Build

---

When a component is built, there are actually quite a number of things that happen. First the capsules referenced by the component are verified, then the model files are written to disk, an external program is called to generate the source code from the model files, the external compiler is invoked to compile, and lastly the linker is invoked to create the final executable version of the component.

Each phase of the build process produces output that is used by the next phase, with the final result being an executable.

### To build a component from the browser:

- 1 Select the component from the model browser.
- 2 Right-click and select **Build** from the popup menu.  
**Note:** If you are working on a UNIX-based platform, and are planning to run the component with Purify, select a component, then click **Build > Build** and select the **Link with purify** option. For information on running a component with Purify, see *Running a Component Instance with Purify* on page 477.
- 3 After the elements have been saved to disk the build dialog appears and shows the build progress.

The build results will be shown. You should review to see if there are any errors or warnings.

### Building a Component from the Build Menu or Toolbar

Instead of directly building a component from the browser, you can build the active component directly from the **Build** menu or by selecting one of the active component toolbar buttons to verify, generate, or build the active component.

## Generate Dialog

---

Use the **Generate** dialog to specify the build options for the selected component.

### Build level

In this area, you can specify the build level for the selected component. The **Build level** options are:

- **Generate** - The capsules and classes referenced by the component are verified, then the model files are written to disk.
- **Generate and compile** - The capsules and classes referenced by the component are verified, then the model files are written to disk, an external program is called to generate the source code from the model files, the external compiler is invoked to compile, and then the linker is invoked to create the final executable version of the component.

## Show warnings

In this area, you want to specify which messages appear in the **Log** tab on the **Output** window as Rational Rose RealTime builds the selected component. The **Show warning** options are:

- **Warning** - All warning messages appear in the **Log** tab in the **Output** window.
- **Notification** - All notification messages appear in the **Log** tab in the **Output** window.
- **Information** - All information messages appear in the **Log** tab in the **Output** window.

**Note:** If you are working on a UNIX-based platform, and are planning to run the component with Purify, select a component, then click **Build > Build** and select the **Link with purify** option. For information on running a component with Purify, see *Running a Component Instance with Purify* on page 477.

## Unable to Compile a Component?

---

When compiling a component in Rational Rose RealTime on a Windows NT or Windows 2000 configuration, the code generation stage may complete without a problem; however, during the compilation stage, you may receive various error messages, such as "Cannot find C:\Program: file does not exist." This error means that your specific compiler does not understand the space between **Program** and **Files** for the location C:\Program Files.

### Cause

Some compilers do not understand spaces in paths to files on Windows configurations.

### Resolution

If your compile does not understand spaces in the paths to files, you can:

- Install Rational Rose RealTime to a location where the path does not contain spaces.
- Use the DOS name.

On Windows NT, to obtain the DOS name of a file, select the file in the Windows NT explorer, right-click and select **Properties**. Now you have to use this DOS name in your **ROSERT\_HOME** environment variable.

On Windows 2000, open a Command Prompt window (cmd.exe) and type the command **DIR /X**. This command displays the short names generated for non-8.3 file names on both Windows NT and 2000. It will look similar to the following:

```
ROBERT_HOME = "c:\Program~1\Rational\Rose~1"
```

After modifying the environment variable, modify your PathMap in the Rational Rose RealTime toolset by clicking **File > Edit PathMap**, and then specify the same name as indicated above.

Now, your model will build after you perform a full rebuild of your components.

- Use the NT **subst** command

You can use the **subst** command to substitute a drive letter to the directory containing spaces. You can then set your **ROBERT\_HOME** environment variable to this drive letter. For example:

```
subst K: "%ROBERT_HOME%"  
set ROBERT_HOME=K:\
```

We recommend that you place the commands in a batch file and ensure this batch file runs every time the workstation is started. Additionally, you must modify your **PathMap** in the Rational Rose RealTime toolset by clicking **File > Edit PathMap**, and rebuild the components.

- Use Windows drive sharing

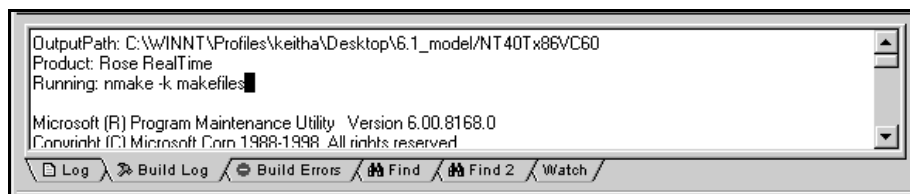
You can enable drive sharing on the directory that Rational Rose RealTime is installed on, and map that directory to a drive letter. Additionally, you must modify your **PathMap** in the Rational Rose RealTime toolset by clicking **File > Edit PathMap**, and rebuild the components.

## Reviewing Build Results

---

You can view the results of your build by selecting **View > Output** and clicking the **Build Log** tab.

**Figure 112 Build Log tab**



Review any errors shown in the **Build Errors** tab, and correct before trying to rebuild. You can jump to the error location in your model by double-clicking on any error shown in the bottom part of the results window. As well, you should be familiar with some of the most common build errors (Understanding Build Errors). They are described briefly and should be used in conjunction with your compiler and linker documentation.

The **Build Log Tab** contains **stdout** and **stderr** of all phases of generation, compilation, and link. The Build Errors Tab contains a parsed version of the output stream.

For information on saving the Build Log output to a file, see *Saving Build Output to a Log File* on page 113.

## Opening Code Generated for Model Elements

---

You can open the code generated after building a component for class, capsule, and component elements within the toolset. Opening the code lets you view or modify the header files (.h for C and C++), body files (.c for C and .cpp for C++), or source files (.java for Java) generated by the toolset.

### Selecting Elements

You can select one or more classes and capsules, or components from the **Model View** tab in the browser, or directly from a **Class** diagram. However, all selected elements must be of the same type and language. This means that your selection must contain only classifiers (classes and capsules) or only components that are all the same language (either all C, all C++, or all Java). Otherwise, the **Browse Header** (C and C++) and **Browse Body** (C and C++), or the **Browse Source** (Java) options on the context menu are not available.

**Note:** Before selecting a class, capsule, or component to view its generated code, you must first build the component for that element. In addition, because you can assign a capsule or class to one, or more components, different code can be generated for every component for the selected model element.

### Selecting a Single Element

In your model, you can select an class or capsule where:

- *The Element is not Assigned to a Component* on page 446
- *The Element is Assigned to a Single Component* on page 446
- *The Element is Assigned to Multiple Components* on page 446

### The Element is not Assigned to a Component

If a class or capsule is not assigned to any component, the **Browse Header** and **Browse Body**, or the **Browse Source** options are not available on the context menu for the selected class, capsule, or component.

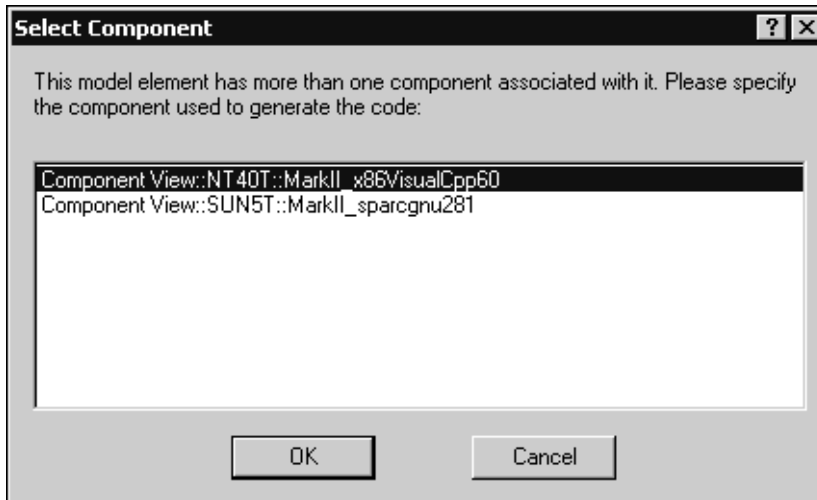
### The Element is Assigned to a Single Component

If a class or capsule is assigned to a single component, Rational Rose RealTime uses that component to find the generated code. The **Browse Header** and **Browse Body**, or the **Browse Source** options are available on the context menu for the selected element.

### The Element is Assigned to Multiple Components

If the selected class or capsule is assigned to more than one component, the **Select Component** dialog (Figure 113) shows all of the components for that element.

Figure 113 Select Component Dialog



If a model has an active component set that appears in the list of selected components, it is automatically selected when **Select Component** dialog opens. For additional information on setting a component as the active component for a model, see *Assigning an Active Component* on page 440.

**Note:** For large models, you may encounter a delay before the **Select Component** dialog appears.

## Selecting Multiple Elements

If you select multiple elements, the **Select Component** dialog (Figure 113) shows only the components that are in common for all of the selected elements.

If the selected elements do not have any common components, you will receive an error message.

## Using an Editor

When viewing the code generated for selected model elements, you have two editor options: using the Internal Editor or specifying an External Editor.

### Internal Editor

Included with Rational Rose RealTime is an editor that you can use to open code generated for classes, capsules, and components.

**Note:** If you do not specify an external editor, Rational Rose RealTime automatically uses its internal editor. The Internal editor can open only one file at a time. If you have multiple elements selected, only the header, body, or source file for the elements selected first appears in the internal editor. We recommend that you specify an external editor if you want to view the code generated for multiple files.

For information on using the shortcut keys in the internal editor, see *Rational Rose RealTime Keyboard Shortcut Summary* on page 571.

### External Editor

You can specify an external editor to view files containing generated code. Specifying an external editor may provide you with additional capabilities, as well as allowing the toolset to open multiple files at the same time. You can specify an external editor by clicking **Tools > Options**, and selecting the **Editor** tab.

For information on specifying an external editor, see *Editor Tab* on page 559.

## Build Menu

---

### Build

Opens the **Build** dialog from which you can choose the **Build Level**.

### Quick Build

Builds the component incrementally.

## **Rebuild**

Forces a complete build of a component. All classes references by the component will be verified, regenerated, compiled, and linked.

## **Clean**

Removes all files from the output directory.

## **Code Sync**

Invokes the mechanism to capture external changes made to the generated code back into the model. For more information, see “Using Code Sync to Change Generated Code” on page 531.

## **Stop Build**

Stops the build (or the Code Sync) in progress.

## **Run**

Loads the component instances specified in the Build Settings Dialog. The component must be successfully built before it can run.

If the Attach Target observability flag was set on the Component Instance Specification dialog, and a Target observability Port number filled in, then the execution interface is displayed allowing you to control the execution of the model.

## **Start (F5)**

Starts the execution of the component instances. If the component instances are in the reset state, then execution begins with all fixed capsules being initialized (initial transitions fired). If the component instances are in the stop state, then execution resumes.

## **Stop (Shift+F5)**

Stops the execution of the component instances at the current point of execution and remembers the state of all capsules. Execution is stopped as soon as each currently running transition is finished. The stop button does not halt execution in the middle of a transition action.



### **Step (F10)**

Steps through the next deliverable message. Pressing the step button while in the stopped state causes the next message of the highest available priority to be delivered, and any associated transitions are executed. Execution stops again as soon as the last transition segment for that message has finished executing.

### **Restart (Ctrl+Shift+F5)**

Resets the component instances, resetting all fixed and destroying all dynamic capsule instances. The running component instance is terminated and a new one is run.

### **Load**

Loads the components instances specified in the Build Settings dialog. The component must be successfully built before it can run. The Load command spawns an external process in which the model executable runs. You will likely see an external command window appear.

The Attach Target observability flag must be set on the Component Instance Specification dialog, and a Target Observability Port number filled in for the model to be loaded within the tool.

The execution interface is displayed allowing you to control the execution of the model. See Execution basics for more information on the execution tools.

### **Reload**

Kills the existing model process and runs the model again. The execution interface stays open.

### **Shutdown**

Kills the existing model process and closes the execution interface.

### **Settings...**

Displays the Build Settings Dialog. You must use this dialog to specify the active component before you can build the component.

### **Add Class Dependencies...**

Runs a script that checks for any missing dependencies between model elements and add them. The script checks dependencies found in attributes or operations. It does not check for code-level dependencies.

## Component Wizard...

Activates the Component Wizard to help you through the steps of creating and deploying a component.

## Build Settings Dialog

---

The Build Settings dialog is used to select an active component for building and component instances for running. The build settings are not saved as part of a model. They are saved with the workspace.

### Active Component

Used to select an active component. The combo box contains all components in your model.

### Active Component Instances List

This list is populated with all the component instances in the model. Component instances that are selected in this list are automatically run when the active component is run. You can select and de-select component instances by clicking in the checkbox on the left-hand side of each component instance name. The order in which the component instances are run is determined by the load order setting in the Component instance specification.

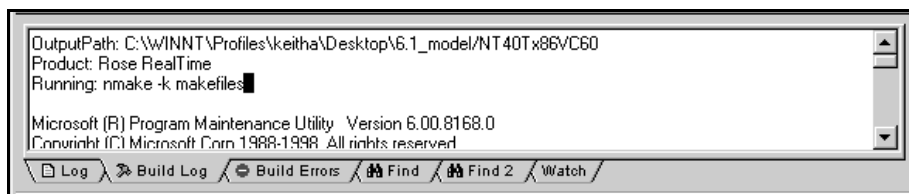
## Build Log Tab

---

The Build Log tab stores the contents of the compilation and code generation log. Select **View > Output** and click the Build Log tab to open it. Compilation or code generation messages are posted to the Build Log tab regardless of whether it is visible.

You can save the contents of the Build log tab to a file. You can also choose to automatically save messages to a file as they are posted.

**Figure 114 Build Log Tab**



The **Build Log** tab contains the output stream from the build. Examine the contents of this window to get see any error message displayed in the build messages list.

## Build Errors Tab

---

The **Build Errors** tab contains a parsed version of the output stream. It is important to review the **Build Log** tab because some errors cannot be parsed by the error parser.

The **Build Errors** tab contains a **Location** column that provides the class/code segment name pair. The **Context** column provides the context of the problem. The **Message** column describes the problem. These messages come directly from the compiler error stream and reflect the accuracy of the compiler that you use. Further, errors within your code segments may lead to errors being reported in system-generated files.

Double-clicking on an error or warning on the **Build Errors** tab brings you to the location in the model where the problem occurred. See *Common Build Errors (Understanding Build Errors on page 463)* for a short summary of common build errors.

## Unknown Compiler Message Stream

It is possible that the compiler being used reports errors in ways that are not understood by Rational Rose RealTime. There are no standards for error reporting by compilers and linkers. Hence, the error parser is often targeted for a particular compiler and linker. If you use an unsupported compiler, Rational Rose RealTime will probably not be able to understand the error output from the parser, and may inaccurately report errors. You have to rely on the raw output stream to see the direct output of the compiler, rather than going by the errors reported by the **Build Errors** tab. See the book *Adapting for Target Environments, Rational Rose RealTime*.

## Component Specification

---

A **Component Specification** displays and modifies the properties and relationships of each component in the current model, and is used for all component kinds.

### Specification Content

The Component Specification consists of the following tabs:

- Component Specification - General Tab
- Component Specification - References Tab
- Component Specification - Relations Tab
- Component Specification - Files Tab

## Component Specification - General Tab

### Name

The component name is referenced during the build process.

### Parent

Specifies the parent component package.

### Environment

Specifies the run-time system and code generator used in the build.

### Type

Specifies what is being built, for example, an executable or a library.

**C++ Executable** - Allows you to build a C++ executable based on a **main** program. This type of component cannot contain capsules and has no dependencies on the TargetRTS.

**C Executable** - Allows you to build a C executable based on a **main** program. This type of component cannot contain capsules and has no dependencies on the TargetRTS.

### Stereotype

A component stereotype represents the subclassification of an element. The most common type of components are already predefined as stereotypes, including Main Program, Package Body, Package Specification, Subprogram Body, Subprogram Specification, Task Body and Task Specification. You can also define and add your own kinds of stereotypes.

### Documentation

Provides a description about the selected component.

## Component Specification - References Tab

### References List

The references list displays the list of packages (includes all elements in the package), classes, capsules and protocols to be compiled with this component.

If during the dependency check elements that are not in this list are found to be needed for the build, a dialog appears asking you to add them.

## Component Specification - Relations Tab

### Relations List

The relations list displays aggregation relations between the component and other components in component diagrams.

## Component Specification - Files Tab

Provides a list of referenced files that you can link external files to model elements for documentation purposes. You can insert and delete references to files or URLs.

## Generating Documentation Fields

---

While debugging on target environments, you can modify source code directly. Consequently, you may need to understand the code that you are working on and may need to provide some kind of modification history independent of the configuration management system.

Rational Rose RealTime includes a feature that enables:

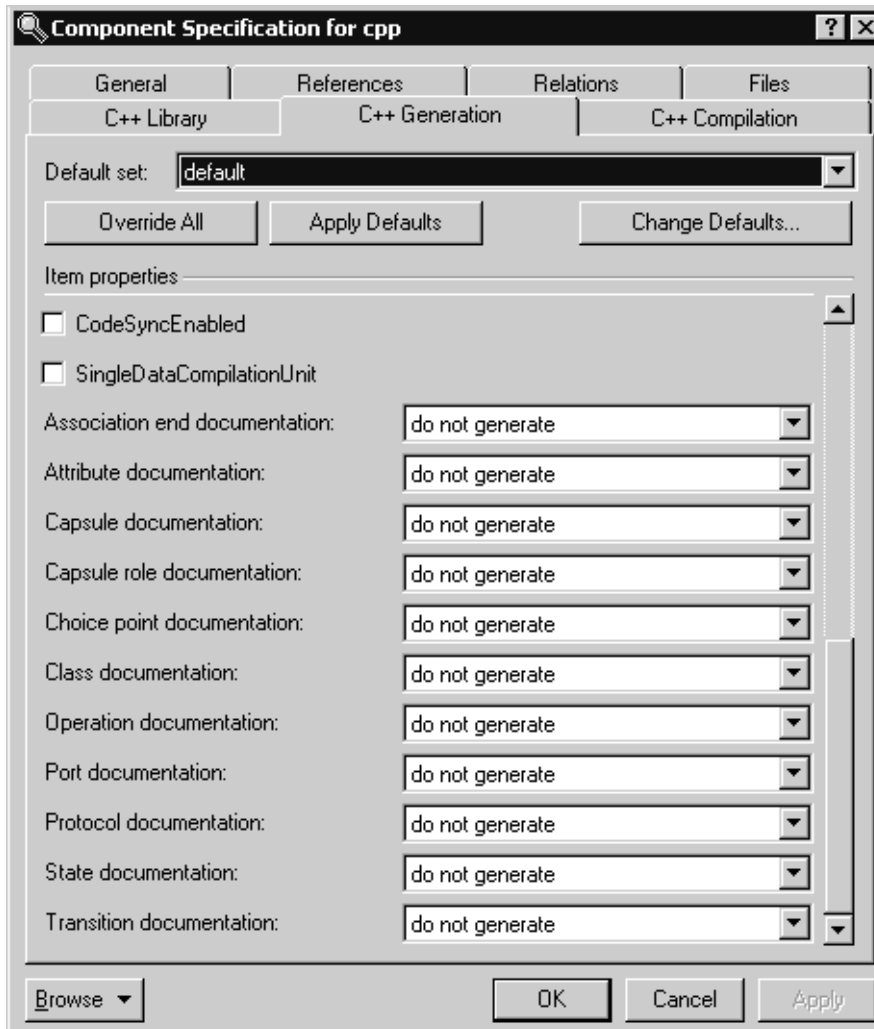
- the generation of documentation fields for different model elements as comments into the generated source files
- the use of code sync to integrate the modified comments back into the model

This feature is particularly useful for multi-byte character and international users.

**Note:** The feature is available for C and C++ models.

For a component, the **C++ Generation** tab for C++ (**C Generation** tab for C) contains all of the fields for which you can generate documentation (see Figure 115).

Figure 115 Component Specification Dialog Box - C++ Generation Tab



You can control the generation of documentation for the following model elements:

- Association end
- Attribute
- Capsule
- Capsule role
- Choice point
- Class
- Operation
- Port
- Protocol
- State
- Transition

The drop-down lists for each box contains the following options:

Option	Description
<b>do not generate</b>	The default option is <b>do not generate</b> . Documentation is not generated when this option is selected.
<b>prefer in header</b>	When selected, the documentation appears with the generated source for the model element in the header file. If the model element does not appear in the header file but appears in the implementation file, then the documentation is generated in the implementation file.
<b>prefer in implementation</b>	When selected, the documentation appears with the generated source for the model element in the implementation file. If the model element does not appear in the implementation file but appears in the header file, then the documentation is generated in the header file. Comments are generated in the implementation file whenever possible, and in the header file only if necessary.
<b>only in header</b>	The code generator outputs documentation only if the source for the affected model element appears in the header file.
<b>only in implementation</b>	The code generator outputs documentation only if the source for the affected model element appears in the implementation file.

Documentation is generated as a comment with its own RME tag (for general documentation), and if code sync is enabled, the documentation is enclosed in `USR` tags (see Figure 116). When Code sync is enabled, empty documentation fields continue to be generated in the source file allowing you to add new comments between the `USR` tags in the generated code. To ensure the unambiguous use of code sync, each generated documentation field in the source files (header or implementation) are generated only once in the source code.

**Figure 116 Generated Documentation with Code Sync Not Enabled**

```
// {{RME classifier 'Logical View::Model::TopCapsule::SystemTest'
// {{RME general 'documentation'
/*
SystemTest is the top capsule.
This is its documentation.
*/
// }}RME

#ifdef SystemTest_H
#define SystemTest_H

#ifdef PRAGMA
#pragma interface "SystemTest.h"
#endif

#include <RTSystem/NT_UC6.h>
#include <Go.h>
#include <Prot.h>
#include <tWait2Send.h>
extern const RTActorClass Dummy;
extern const RTActorClass DummySubClass;
extern const RTActorClass ReplyHiSub;

extern const RTActorClass SystemTest;
""\work\NT_UC6\src\SystemTest.h" [unix] 245L, 7035C 5,8 Top
```

Figure 117 shows the generated documentation with code sync enabled.

**Figure 117 Generated Documentation with Code Sync Enabled**

```
// {{RME classifier 'Logical View::Model::TopCapsule::SystemTest'
// {{RME general 'documentation'
/* {{USR
SystemTest is the top capsule.
This is its documentation.
}}USR */
// }}RME

#ifdef SystemTest_H
#define SystemTest_H

#ifdef PRAGMA
#pragma interface "SystemTest.h"
#endif

#include <RTSystem/NT_UC6.h>
#include <Go.h>
#include <Prot.h>
#include <tWait2Send.h>
extern const RTActorClass Dummy;
extern const RTActorClass DummySubClass;
extern const RTActorClass ReplyHiSub;

// {{RME tool 'OT::Cpp' property 'HeaderPreFace'
""\work\NT_UC6\src\SystemTest.h" [unix] 279L, 7656C 5,8 Top
```

**Note:** When code sync is disabled, comments are generated in the source files without USR tags (see Figure 122). Also, empty documentation fields are not generated.



If code is generated with code sync enabled and some documentation comments are modified or added in the generated source, those changes can be imported back into the model with the Code sync feature.

## Using Generated Documentation Fields

Typically, you will generate the source code with code sync enabled and with documentation for some (or all available) model elements enabled. During a review of the generated code, a user may modify the source code and the documentation. Code sync will import these changes back into the model.

For example, given the following class diagram:

**Figure 118 Class Diagram - Example Model**

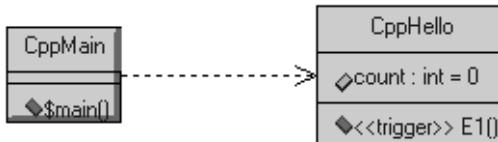


Figure 119 shows the results after building the model (the code that is generated in the header file for CppMain.h).

**Figure 119 No Documentation Generated and Code Sync is Disabled**

```
CppMain.h
//{{RME classifier 'Logical View::CppMain'
#ifdef CppMain_H
#define CppMain_H

#ifdef PRAGMA
#pragma interface "CppMain.h"
#endif

#include <RTSystem/cpp.h>

//{{RME operation 'main(int, const char * const *)'
int main( int argc, const char * const * argv );
//{{RME

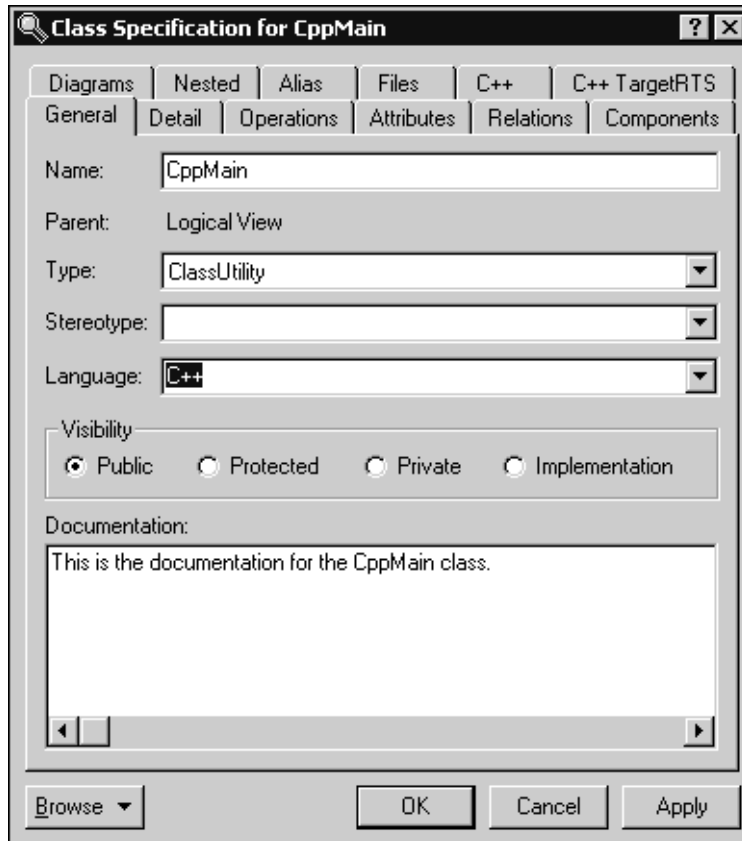
#endif /* CppMain_H */

//{{RME
```

The screenshot shows a code editor window titled 'CppMain.h'. The code is a C++ header file generated from a model. It includes preprocessor directives for conditional compilation, a pragma interface directive, and an include directive for 'RTSystem/cpp.h'. The main function signature is also present, along with RME (Runtime Model Element) markers.

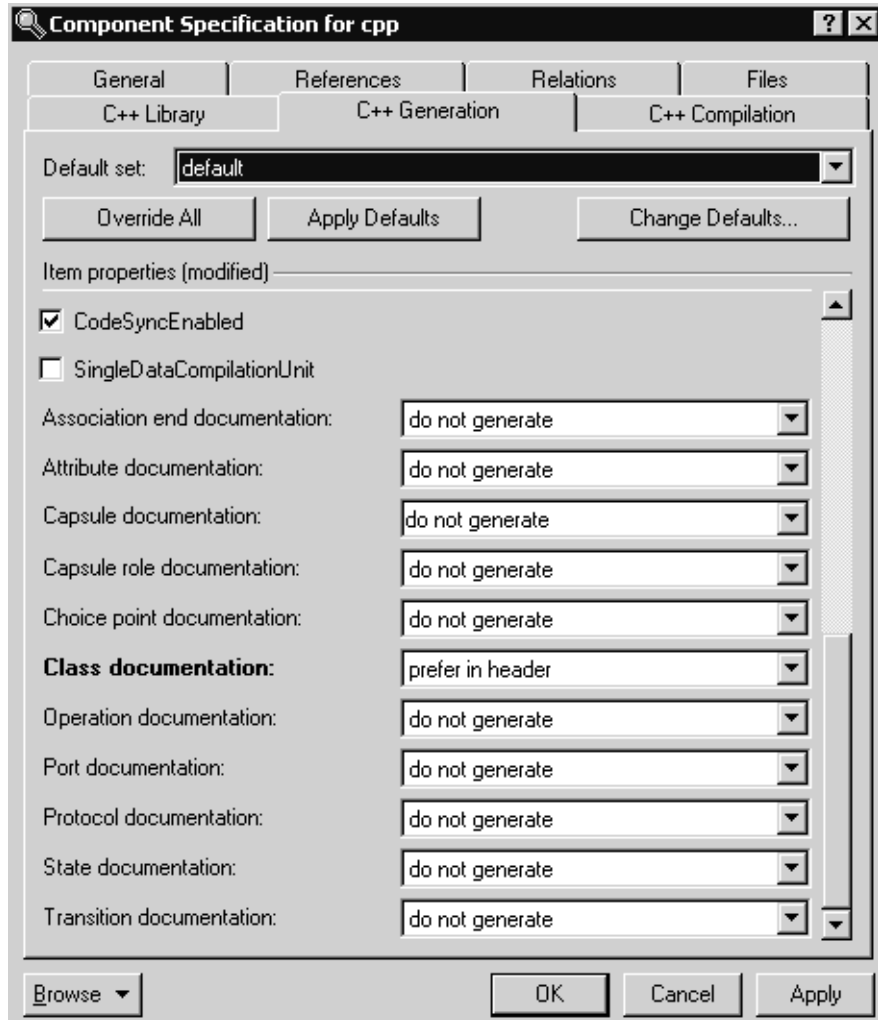
Next, open the **Class Specification** dialog box for the **CppMain** class and add text to the **Documentation** box in the **General** tab (see Figure 120).

**Figure 120 Class Specification Dialog Box for the Class CppMain**



To see the documentation fields, select the desired component from the **Model View** tab in the browser, right click and select **Open Specification**, then click the **C++ Generation** tab. At the bottom of this tab, select **prefer in header** for the **Class documentation** box (see Figure 121). For this example, we will also select the **CodeSyncEnabled** option.

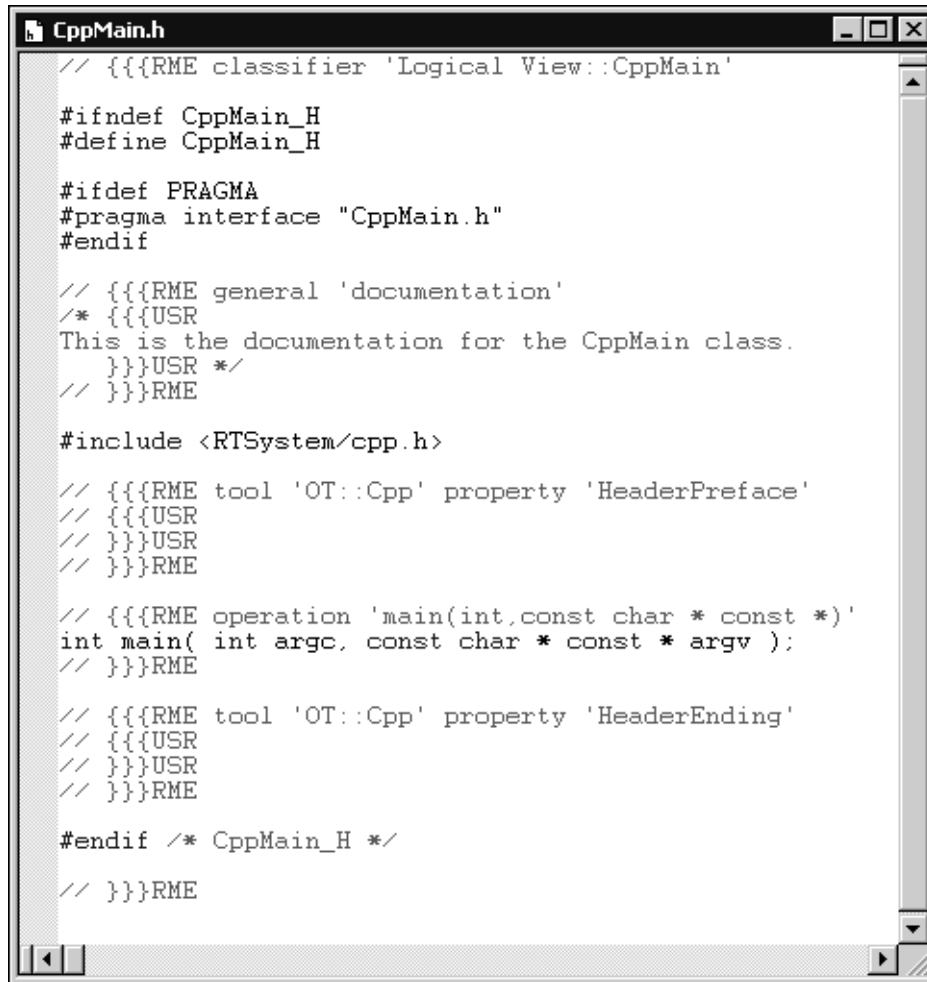
Figure 121 Component Specification Dialog Box for the Cpp Component



**Note:** When the label for a box is bold, it means that this property is overridden.

After you build the model, if you open the header file (CppMain.h), the file is updated to include the text in the **Documentation** box for all classes in the model.

Figure 122 Generated Documentation with Code Sync Enabled



```
CppMain.h
// {{{RME classifier 'Logical View::CppMain'
#ifdef CppMain_H
#define CppMain_H

#ifdef PRAGMA
#pragma interface "CppMain.h"
#endif

// {{{RME general 'documentation'
/* {{{USR
This is the documentation for the CppMain class.
}}}USR */
// }}}RME

#include <RTSystem/cpp.h>

// {{{RME tool 'OT::Cpp' property 'HeaderPreface'
// {{{USR
// }}}USR
// }}}RME

// {{{RME operation 'main(int, const char * const *)'
int main( int argc, const char * const * argv );
// }}}RME

// {{{RME tool 'OT::Cpp' property 'HeaderEnding'
// {{{USR
// }}}USR
// }}}RME

#endif /* CppMain_H */

// }}}RME
```

**Note:** Some types of model elements have source code generated only in the header file or only in the implementation file. For example, capsule role and port RME tags appear only in the header file so the documentation cannot be generated in the implementation file. Similarly, RME tags for states are generated only in the implementation file.

**Note:** RME tags for transitions that do not have any code are not generated in the source files. Consequently, their documentation will not appear in the source files.

## Component Dependencies

---

You can break up the system you are building into multiple components. Model the build dependencies using the component dependencies.

See the *Guide to Team Development - Rational Rose RealTime* and language-specific guides for more information.



## Contents

This chapter is organized as follows:

- *Understanding Build Errors* on page 463
- *Model Management - Importing Model Compilation Results* on page 468

## Understanding Build Errors

---

Often, the compilation details returned in the Build Results window gives you a clearer picture of what the error is, but you must understand what the compilation details are reporting. The compilation details are the direct results returned by the compiler. They contain the names and line numbers from the actual generated code, so you have to analyze the error message to determine where the error occurred. It is often very useful to refer to the compilers documentation to understand the meaning of certain reported errors or warnings.

Below is a small list of generic and common build symptoms with possible causes:

### **Unknown command, command not found, the name specified is not recognized**

- Is your compiler installed correctly?
- Is your make program configured and installed correctly?
- Are you linking with the correct Services Libraries?
- Are the Rose RealTime environment variables set?

### **Redefinition of basic types or multiple declarations for X**

- Do you have any name conflicts?

### **Unresolved symbol or undeclared identifier**

- Have you configured the necessary inclusions, libraries, or object files?
- Are you missing dependencies between classes in your model?
- Does the Capsule role have the same name as the Capsule?

## Missing Class Dependencies

Missing dependencies are a common source of compilation errors. You need to identify which capsules and classes depend on other classes in your model. That way when you compile a capsule or class, it will find the definition of the class you depend upon. Also if that class's interface changes, the build process will automatically rebuild all the capsules and classes that depend upon it.

To resolve these types of errors add the correct dependencies between classes using the **Build > Add Class Dependencies** Wizard or by manually creating a dependency relationship between classes.

## Capsule Role Name Same as Capsule Name

An error of this type is generated when a capsule role instance has the same name as a capsule.

To resolve the problem, give the capsule role a different name than the capsule class. A good rule in situations like these is to always start capsule class name with an uppercase letter, and capsule roles with lowercase letters.

## Linking Wrong Services Library Set

If you find that the output stream has many undefined messages, you may be accessing an inappropriate Services Library set.

Code generated with this release of Rose RealTime does not work with the Service Libraries of previous releases (and vice versa).

The compiler must match the library set being used since most compilers do name-mangling on variables. For example, if your compiler target is NT and your compiler is MSVC++ 6.0, use the target NT40 and the x86-VisualC++-6.0 library entry.

## Compiler Not Installed Correctly

If the CC environment variable is either undefined or the default compiler and linker defined in **libset.mk** cannot be found, or CC is defined to something that either cannot be found or is not a compiler, you will sometimes see the following in the raw output field of the Build Results window if your make command is not found:

```
The name specified is not recognized as an internal or external
command, operable program or batch file.
```



## Compile a Simple Hello World Program

To ensure that your compiler and linker are installed correctly, write and build a small test program from outside of Rose RealTime. Ensure that it compiles and runs successfully.

## Check Environment Variables

You should be able to invoke your compiler and linker from outside of Rose RealTime on the command line. If you cannot you should verify your PATH environment variable and ensure that the directory that contains the tools for your platform is in the path.

## Review Your Compiler Flag Settings

You should review your compiler settings. Have you overridden the default compiler, or have you added flags to the component specification compiler tab?

## System Does Not Understand the Make Command

Your OS does not understand make or the make is being used is in some interesting way different from what Rose RealTime expects. You will sometimes see the following in the raw output field of the Build Results window if your make command is not found:

```
The name specified is not recognized as an internal or external
command, operable program or batch file.
```

## Check Environment Variables

You should be able to invoke **make**, **gmake**, or **nmake** from outside of Rose RealTime, for example, on the command line. If you cannot you should verify your PATH environment variable and ensure that the directory that contains the make utility for your platform is in the path.

## Ensure that Component has Correct Make Types Configured

Also, you should ensure that the make name and types defined in the component specification compilation make and generation make tabs represent the correct type of make installed on your system.

## Name Conflicts

Odd compile errors can easily be caused by name conflicts, such as naming a capsule role the same as a signal name. You must be aware of the name scoping of various entities in your programming language to ensure that no conflicts occur.

In Rose RealTime, most named entities have capsule-level name scope. For example, within a capsule class, the following are named entities, and any duplication among the names of these entities may cause problems:

- capsule roles
- attributes
- ports
- operations

As well, symbols declared as extern, as in included .h files, are generally part of the global name space, and must not conflict with any names of entities in your model. There are several names that are reserved for the OS/Compiler, such as 'return' and 'exit'.

Some name conflicts are more insidious in that the conflicting names are actually 'compile-time' compatible, and slip by the compiler, resulting in a run-time error that may be difficult to track down. Typically, this means that the elements actually have a common superclass, or, if the error occurs in a function which takes a void \* parameter, it is because the entity that was passed as a parameter was not the expected one.

## Missing Header Files, Object Files, and Libraries

Most models make calls to external code libraries, even if it is just the basic system calls (such as printf, scanf, cin, cout, and so forth). The include files that define these calls must be specified prior to compilation, so that the compiler can resolve these references. Likewise, the libraries or object modules that contain the actual compiled definitions of these external classes and functions must be specified so that the linker can resolve the symbol references.

You will likely see the following type of error message if you have not included the correct header files:

```
'print_this' : undeclared identifier
```

You will likely see the following type of error message if you have not specified a library or object files that should be linked into your model:

```
unresolved external symbol "int __cdecl print_this(void)"  
fatal error XXXXXX: 1 unresolved externals
```

To resolve these types of errors add the correct files or search directories to the component specification dialog under the inclusions or libraries tabs.

## **Compile Fails on Valid C++ Models with VC++ 5.0 or VC++ 6.0**

The \$INCLUDE and \$LIB environment variables may not be properly set. Ensure that your compiler binaries are on the path and that the \$INCLUDE and \$LIB environment variables are set (for example, they could be set for the user who installed VC++, but not set for another user). Set the environment variables. Refer to the VC++ documentation for further details.

Error loading Capsule ("could not spawn process")

If the executable (capsule1.exe) is stored on an NFS server then the NFS client must be configured to have execute permission set.

## **Error Linking Capsule - Error From nmake**

If the executable (capsule1.exe) is stored on an NFS server then the NFS client must be configured to have execute permission set.

## **Windows NT Compilation Command Line Limits**

If you encounter a compilation error message that complains about the command line being too long, the cause may be that the length of your compile or linker has exceeded a limit.

Windows NT compilation has command line limits in two areas: source compilation and linking. Both limits have been explored for the Visual C++ 5.0, VRTX PPC Microtec 1.4 and Tornado 1.0.1 PPC Cygnus 2.7.2 compilers.

## **Source File Compilation**

The variables in source compilation are the update name, the \$ROSERT\_HOME path, compilation options, the local working directory and include directories. The only compiler that has a measurable limit is VRTX. The command line limit is 768 characters.

A workaround for the problem is to reduce the number of include directories by combining include files. Other solutions are to shorten paths and names for the variables listed in the previous paragraph.

## Linking

The variables in linking are the update name, the \$ROSERT\_HOME path, the link options, the number and name length of libraries, the library search paths and the local working directory. The link limits are shown below:

- Visual C++ 5.0: more than 20875 characters
- VRTX PPC Microtec 1.4: 4147 characters
- Tornado 1.0.1 PPC Cygnus 2.7.2: 4150 characters

A workaround for the problem is to shorten paths and names for the variables listed in the previous paragraph.

## Model Management - Importing Model Compilation Results

---

You can now import your model compilation results in to the **Build Log** and **Build Errors** tabs in the **Output** window. Importing compilation results allows you to address build issues and errors within your models at a later time, instead of addressing them immediately after the model builds.

### Build Log Tab - Saving and Importing Compilation Results

The **Build Log** tab contains the raw output stream from the build. You can examine the contents of this window on any error message displayed in the build messages list (**Build Errors** tab). To capture the results in the **Build Log** tab, you will first have to capture the results to a file. You can capture the output by:

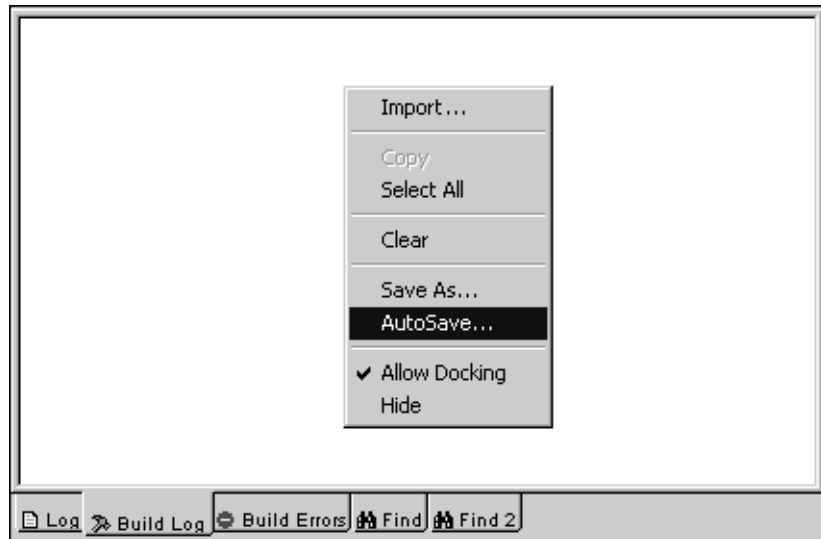
- building the model from the command-line and capturing the output to a file
- *Saving the Build Output to a File Directly from the Build Log Tab* on page 468

### Saving the Build Output to a File Directly from the Build Log Tab

You can automatically or manually capture all of the build output to a log file to process the output later.

**To automatically save build results to a file:**

- 1 In the **Build Log** tab in the **Output** window, right-click and click **AutoSave**.

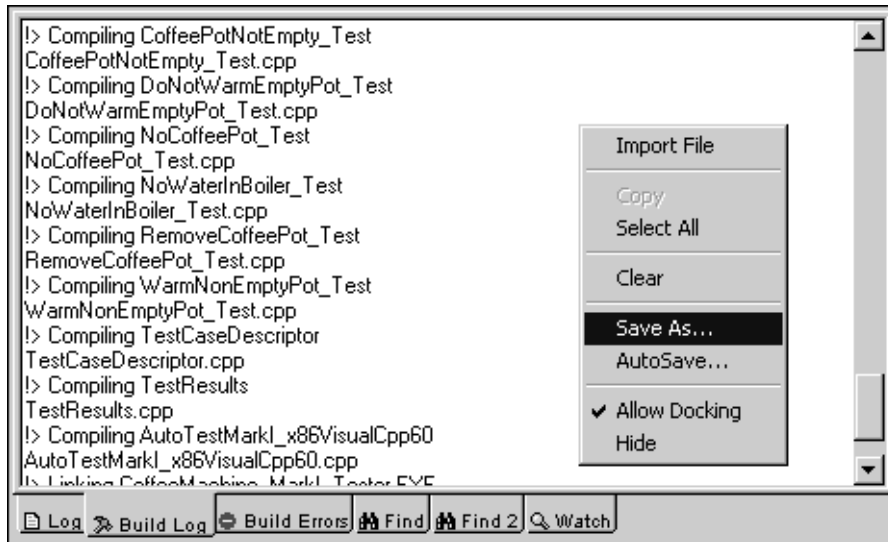


- 2 In the **AutoSave Log** dialog, specify a name for the log file and select a location.
- 3 Click **OK**.
- 4 Build your model.

**Note:** If you attempt to open the **Build Log** file, you may encounter a Sharing Violation message. To view the contents of the **Build Log** output file, right-click in the **Build Log** tab, click **AutoSave**, and then open the log file.

## To save the Build Log results after compilation:

- 1 In the **Build Log** tab in the **Output** window, right-click **Save As**.



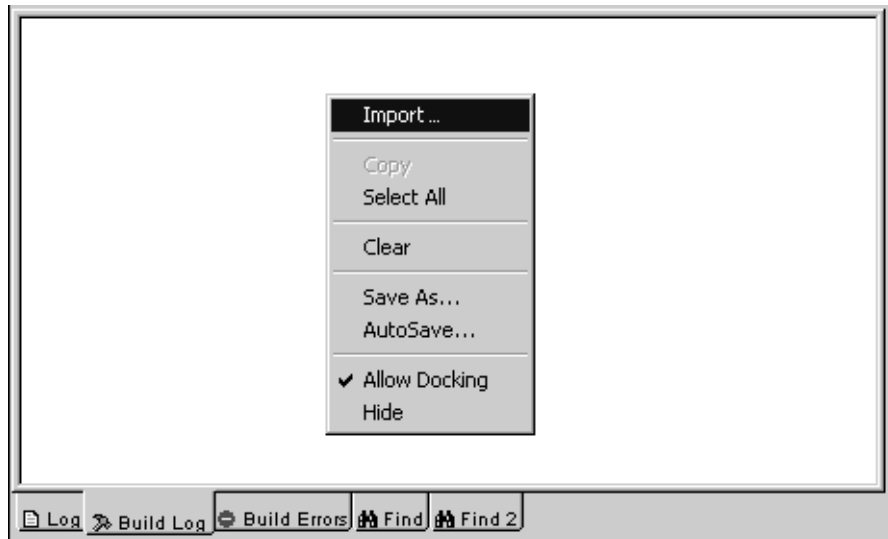
- 2 In the **Save Log** dialog, specify a name for the log file and select a location.
- 3 Click **OK**.

## Importing from the Build Log Tab

If you saved the **Build Log** results (for example, by capturing the results from running a build from the command-line, or by selecting **Save As** or **AutoSave** from the context menu in the **Build Log** tab), you can import results at a later time for further examination.

## To import Build Log results previously captured to an output file:

- 1 In the **Build Log** tab in the **Output** window, right-click **Import**.



- 2 In the **Import Build Output** dialog, specify the name for the .log or .txt file containing the build compilation results.
- 3 Click **Open**.

The **Build Log** tab now contains the results from a previous build.

**Note:** When importing the build log file (after saving the log file in Rational Rose RealTime by clicking **Save As** from the context menu on the **Build Log** tab), only the **Build Log** results are imported. No build errors are imported because clicking **Save As** only saves the **Build Log** results. However, if you create a log file from the command line, importing the results will populate the **Build Log** and the **Build Errors** tab.

## Build Errors Tab - Importing Compilation Results

The **Build Errors** tab contains a parsed version of the output stream. You can examine the contents of this window for any error message displayed in the build messages list. To capture the results from the **Build Errors** tab, you will first have to capture the results to a file. You can capture the compilation results by building the model from the command-line and capturing the output to a file.

### To capture the Build Errors compilation results to a file:

- 1 Open an existing model and select a component.
- 2 Right-click on the component, and click **Build > Build**.
- 3 In the **Build** dialog for the selected component, click **Generate**.
- 4 From the command-line, change the directory to the location of the output for your model (build).
- 5 Use the following instructions to redirect the results to an output file.

**Note:** Typically, the syntax should be similar to the following:

```
make_command [/nologo] RTcompile redirection_command OutputFile.log
```

where:

**make\_command** - Specifies the **make** command (the command being used to control the code generation for your specific platform) in the build directory. This command appears in the build directory after you click **Generate** in Step 3 (such as **make** on UNIX, and **nmake** on Windows).

**/nologo** - Suppresses the logo screen on startup. Use this option only when using the **nmake** command.

**RTcompile** - Causes the compilation of the code after successful generation.

**redirection\_command** - Consult your documentation for the appropriate redirection commands (specify redirection for both standard out and standard error).

**OutputFile.log** - Specifies the name of the file that will contain the compilation output results.

- For example, for Windows, type the following command:

```
nmake /nologo RTcompile > BuildFileLog.log 2>&1
```

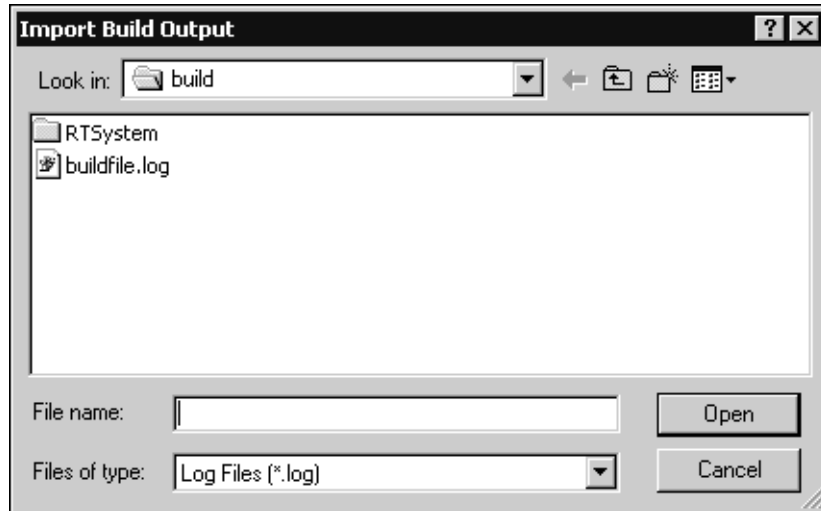
- For example, for UNIX (csh), type the following command:

```
make /nologo RTcompile >& BuildFileLog.log
```

**Note:** This command might work better from the output directory because `rtsetup.pl` may be used.



- 6 In the **Build Errors** tab in the **Output** window, right-click and select **Import**.



- 7 In the **Import Build Output** dialog, specify the name for the .log or .txt file containing the build compilation results.
- 8 Click **Open**.

The **Build Errors** tab now contains the results captured from the command-line.

**Note:** When importing the build file (the output file created from the command-line), if you specified redirection for both standard out and standard error, the compilation results for both the **Build Log** and the **Build Errors** tabs are imported.



## Contents

This chapter is organized as follows:

- *Execution Basics* on page 476
- *Creating a Component Instance* on page 476
- *Running a Component Instance with Purify* on page 477
- *Running a Component Instance without Purify* on page 479
- *Observing a Running Component Instance* on page 481
- *Rational Rose RealTime Execution Interface* on page 482
- *Overview of Observability Options* on page 483
- *Component Instance Menu* on page 484
- *RTS Browser* on page 485
- *Monitors* on page 488
- *Navigating to Model Elements from Debug Monitors* on page 490
- *Trace Windows* on page 490
- *Probes* on page 493
- *Inject Window* on page 494
- *Capsule Instance Trace* on page 494
- *Message Trace Configuration Dialog* on page 496
- *Execution Watch Tab* on page 496
- *Run-time Exception While Running a Component Instance* on page 497
- *Instance Browser* on page 498
- *Source Code Debugging* on page 498
- *Source Debugger Integration without Target Observability* on page 500
- *Setting Breakpoints* on page 500
- *Customizing Rational Rose RealTime for Target Control and Observability* on page 507
- *Running from Outside the Toolset* on page 508
- *Using the Command Line* on page 509
- *Loading and Running Component Instances on Embedded Targets* on page 510
- *Component Instance Specification* on page 511
- *Processor Specification Dialog* on page 516
- *Device Specification* on page 525
- *Connection Specification* on page 526
- *Probe Specification* on page 527

## Execution Basics

---

After a component has been built successfully, you can run the resulting executable. If you have Purify installed, you can run the executable with Purify, to customize error detection for each component in your program. After the component has been built, see *Running a Component Instance with Purify* on page 477. If you do not have Purify installed, see *Running a Component Instance without Purify* on page 479.

Rational Rose RealTime provides an execution environment that can be used to execute and observe component instances on a processor (a type of node).

While a component instance runs, you can control and observe its execution. This functionality is very powerful: it allows a component instance to be observed at the modeling language level, rather than at the source code level.

### Tasks

- 1 Creating a component instance
- 2 Running a component instance with Purify
- 3 Running a component instance without Purify
- 4 Observing a running component instance

## Creating a Component Instance

---

Before running a component that has been built, you must first assign an instance of the component to a processor.

### Tasks

- 1 Select the **Deployment View** folder, right-click and from the popup menu click **New > Processor**.
- 2 A new processor with the default settings for your platform is created.
- 3 Double-click on the Deployment diagram to open it.
- 4 Drag and drop the processor onto the Deployment diagram.
- 5 Then drag and drop a component from the **Component View** model browser on to the processor that was just added to the Deployment diagram.

**Note:** You can also create a component instance by dragging and dropping a component from the model browser onto a processor in the model browser or to the **Processor Specification - Detail** tab.

- 6 Open the processor's specification dialog, and change to the Details tab. Under the Component Instances list you should see the new component instance that was created.
- 7 From the **Model View** browser you can also see the list of component instances associated with their respective processor.

## Running a Component Instance with Purify

---

If you do not have Purify installed on your system, see *Running a Component Instance without Purify* on page 479.

After the component is built and a component instance has been created, the instance can then be run and observed. Purify detects errors in your own code as well as the components your software uses.

The **Run with Purify** item is only visible if you have Purify installed.

The processor must have the same operating system as the toolset, otherwise the **Run with Purify** item will be grayed out. For example, a component instance with a Unix processor must be running on a Unix operating system.

If you are using a UNIX operating system, ensure that you linked with Purify during the build.

### Tasks

If you have configured an active component, then once the build is complete, you can use the execute icon from the toolbar (press F5), or select **Build > Run with Purify** from the main menu to automatically run all the component instances selected in the Build Settings dialog.

You can also run any component instance by selecting the component instance from the model browser, right-clicking and selecting **Run with Purify** from the popup menu.

After you select **Run with Purify**, you will be prompted to select **Yes** if you haven't got a build. After you answer the prompt, it may take a minute or so before the toolset finishes running the executable, especially for a large model.

While a component instance runs with Purify, follow these steps to set up execution control from the toolset:

- 1 A console window appears and you must ensure that the following is displayed (for Windows NT users).

**Note:** A console window only appears on host-based targets. Other tools are required to see console windows on targets.

If the observability line below is not shown in the console, ensure that the observability check box and observability port have been configured in the Component Instance specification.

```
Purify for Windows NT,  
Copyright (C) 1993-2002 Rational Software  
All rights reserved.  
Version 2002.05.00 Early Access; Build: 3142;  
WinNT 4.0 1381 Service Pack 6A Uniprocessor free  
Instrumenting:  
    Compile.EXE 241726 bytes
```

```
Purify: while processing file  
z:\versions\models\myfiles\build\Compile.EXE:
```

Note: Instrumentation repeating with 6 additional entry points.

```
Rational Rose RealTime C/C++ Target Run Time System  
Release 6.20.B.03 (+c)  
Copyright (c) 1993-2002 Rational Software  
rosert: observability listening at tcp port 8978
```

- 2 Bring control back to the toolset by clicking on any part of the toolset. You will notice a new tab called RTS has been added on you model browser. The browser contained in this new window is called the RTS Browser. It is used to control the execution of a running component instance. You can run and control multiple component instances from within Rational Rose RealTime, for each running instance there is a separate RTS Browser tab.
- 3 Click on the new tab to show the RTS Browser.
- 4 The execution control buttons are at the top of the RTS browser. Press the Start button to start the execution of the loaded component instance. Everything printed from your model to stdout and stderr will be shown in the console window that appeared when the component instance was loaded.

- 5 After you exit the RTS browser, the Purify window appears with the Purify results. For information on how to interpret the results, see “Interpreting the Purify Log Reports” on page 297. For information on how to save the Purify results to a file, see “Running from outside the toolset” on page 318.
- 6 When you are finished running the component instance with Purify, press the shutdown button. The component instance is killed and control is returned to Rational Rose RealTime.

**Note:** You can also control the execution of a component instance by using the entries in the Build section of the main menu, or in the popup menu of a component instance.

## Interpreting the Purify Log Reports

The Purify output is displayed in a tree control listing all exceptions in order of occurrence. When running on a UNIX platform, each exception report consists of a message. When running on a Windows platform, each exception report consists of a message preceded by an icon, to indicate the severity.

- Messages preceded by a blue circle containing the letter **i** are for information only.
- Messages preceded by a red circle containing the letter **i** indicate that there is a user error.
- Messages preceded by a yellow triangle containing an exclamation mark (!) are warning messages. They usually indicate memory leaks.

If the message text is bold, it indicates that there is something in the model you can see; usually a user error, such as a memory leak.

If several levels of message text are bold, you can scope down to the actual message which points to the line of code changed by the user. You can double click on the bold messages to see the section in the code that caused the message.

## Running a Component Instance without Purify

---

After the component is built and a component instance has been created, the instance can then be run and observed. There are two basic ways of running component instances. They are both described below.

### Tasks

If you have configured an active component, then once the build is complete you can use the execute icon from the toolbar (press F5), or select **Build > Run** from the main menu to automatically run all the component instances selected in the Build Settings dialog.

You can also run any component instance by selecting the component instance from the model browser, right-clicking and selecting **Run** from the popup menu. If the **Run** item is grayed out, it is probably because the target control scripts configuration is pointing to the wrong directory in the Processor specification.

While a component instance runs, follow these steps to setup execution control from the toolset:

1 A console window appears and you must ensure that the following is displayed.

**Note:** A console window only appears on host-based targets. Other tools are required to see console windows on targets.

If the observability line highlighted below is not shown in the console, ensure that the observability check box and observability port have been configured in the Component Instance specification.

```
Rational Rose RealTime C/C++ Target Run Time System
Release 6.20.C.00 (+c)
Copyright (c) 1993-2001 Rational Software
rosert: observability listening not enabled
```

2 Bring control back to the toolset by clicking on any part of the toolset. You will notice a new tab called RTS has been added on you model browser. The browser contained in this new window is called the RTS Browser. It is used to control the execution of a running component instance. You can run and control multiple component instances from within Rational Rose RealTime, for each running instance there is a separate RTS Browser tab.

3 Click on the new tab to show the RTS Browser.

4 The execution control buttons are at the top of the RTS Browser. Press the Start button to start the execution of the loaded component instance. Everything printed from your model to stdout and stderr will be shown in the console window that appeared when the component instance was loaded.

5 When you are finished running the component instance, press the shutdown button. The component instance is killed and control is returned to Rational Rose RealTime.

**Note:** You can also control the execution of a component instance by using the entries in the Build section of the main menu, or in the popup menu of a component instance.



## Observing a Running Component Instance

---

A very powerful feature of Rational Rose RealTime is the ability to observe a running component instance at the model level. This kind of high-level debugging is not what most developers are used to. More conventionally, developers converted design models to source code. When it was compiled and run, the only way to trace the execution was at the source code level. The design model representation was of no use.

In Rational Rose RealTime you can see the triggered transitions, active states in the state monitors, and watch the dynamic structure animated in the structure monitor. In addition, you can use probes to trace the messages being passed in the system.

### Tasks

Observe a running capsule instance by opening monitors and message traces:

- 1 Once you have followed the steps to run your component instance, change to the RTS browser tab and press the Start button.
- 2 Expand the top-level capsule folder and select a leaf capsule instance. Non-leaf capsules instances represent the class of the instances.
- 3 Right-click on a capsule instance, and from the popup menu select **Open State Monitor**.

A monitor window appears, and you should be able to see the state machine of this capsule instance. The current state is highlighted in black. In addition the last transition fired is drawn in black.

- 4 Select the Probes tool from the monitor toolbox. Place a probe onto a state by moving the probe cursor over the state then clicking the left mouse button to apply the probe to the state.  
Select the probe that you have just applied to a state, and from the popup menu choose **Open Trace Window**.
- 5 The opened trace window shows all messages that occur in this state. Follow similar steps for adding probes to ports, and junction points.
- 6 Notice that any new probe that is added to a monitor is also added to the Probes folder in the RTS Browser. You can perform common operations on probes by using the popup menu from the Probes folder.

# Rational Rose RealTime Execution Interface

---

The execution control of component instances is separated into two main functions: the target control of the component instance and the observability of a component instance. The target control interface provides an interface for automating the tasks related to running, loading, and terminating component instances. The observability interface provides the ability for the Rational Rose RealTime toolset to connect to a running component instance and provides a visual view of the running instance.

## Target Control Programs

In order to allow control of component instances on different platforms, easy customization, and support for other targets, the target control utilities are implemented as a set of external executables and scripts that are invoked from the toolset to perform the various target control tasks.

These scripts and executables for target control are located in the following directory:

```
$Target_scripts = $ROSEXT_HOME/bin/tc/"host"
```

Below the tc directory (tc for target control) is a list of the hosts on which Rational Rose RealTime can run. And within each of these directories is a list of platforms for which there are control utilities. For example, in the `$ROSEXT_HOME/tc/win32` directory there are other directories, for example, win32, tornado, and tornada2. This shows that for a toolset running on a Windows platform the toolset can control component instances for Windows and Tornado platforms, meaning that they can be run, loaded, terminated automatically by Rational Rose RealTime.

The Processor specification dialog must be told in which directory to look for the control utilities for the platform that the processor represents. The control options on the component instance menu (run, load ...) are enabled or disabled depending on the control utilities that are found in the directory specified for that processor. For each utility program that the toolset finds in the target control directories, the appropriate menu item is enabled, indicating that the toolset supports the control function for that platform.

**Note:** You can always manually run, load, etc., a component instance from outside the toolset.

## Overriding Target Control

The Operation mode field on a Component Instance specification dialog specifies whether the controls utilities should be used or the component instance will be loaded manually. In the latter case, most of the component instance control menu items are disabled.

## Observability Interface

Once a component instance is running (it must be listening to a specified tcp/ip port using the -obslisten command line parameter) the observability interface can connect to the running component instance, and control and animate its execution.

You can observe (connect to) any component instance that was started with observability enabled (listening to the tcp/ip port specified in the component instance spec dialog) even though it was not started with the target control utilities. Use the **Attach Target** option in the component instance to observe a running component instance.

## Overview of Observability Options

---

After you become familiar with building and running within the toolset, you can start debugging your models. There are several options that are available from within execution environment. You should read the details about each option to become more familiar with the following options.

Observability option	Explanation
Watches	Use watches to inspect and modify the values of capsule attributes.
Traces	Use traces to see the messages that are being sent within the system.
Injecting Messages	Inject test messages into a model to unit test capsules.
Probe Break Points	Use probe breakpoints to stop a running model when a specified event is received.
Sequence Diagrams	Create and save sequence diagrams of message traces between capsule instances.
Command Line Debugger	Debug a model without the use of the toolset.
Source Code Debugging	Debug detail code problems using a source code debugger.

# Component Instance Menu

---

The component instance menu provides commands that are used to control the execution of a component instance.

## **Load**

Loads or downloads a component instance to a target platform. The load does not start the execution of the loaded component instance. Use **Run** once it is loaded. This is only used with target platforms that require loading of modules before they are run. For platforms that do not require loading of modules, this menu item is disabled.

## **Unload**

Use only with target platforms that require loading of modules before they are run. For platforms that do not require loading of modules, this menu item is disabled.

## **Run**

Starts the execution of the component instance. If observability is configured to attach at start-up the RTS browser appears. When observability is attached at start-up the component instance is paused, or does not start processing messages, until the start button is pressed on the RTS Browser.

## **Run with Purify**

Starts the execution of the component instance and tests for different aspects, including memory leaks. This menu item appears only if Purify is installed.

## **Shutdown**

Kills the running component instance, closing the RTS Browser if necessary.

## **Restart**

Kills the running component instance and runs another instance. If the instance is running on an target board, the component is reloaded before a new instance is run.

## **Reload**

Used only with target platforms that require loading of modules before they are run. For platforms that do not require loading of modules, this menu item is disabled. This unloads then loads the component without resetting the target board.

### **Attach Target**

Enabled only if a component instance has been run without observability at startup. You can attach observability to a running process at any time, if that the process was started with observability enabled. This menu item can only be used with **Detach Target**.

### **Detach Target**

Detaches observability, meaning that the toolset no longer communicates with the running component instance. This menu item can only be used with **Attach Target**.

### **Attach Console**

Attaches a console window to the executing target model to interact with the command line model debugger, and so forth.

### **Open Breakpoint Diagram**

Opens the **Breakpoint Diagram** dialog from which you can set various breakpoints.

### **View BreakPoints**

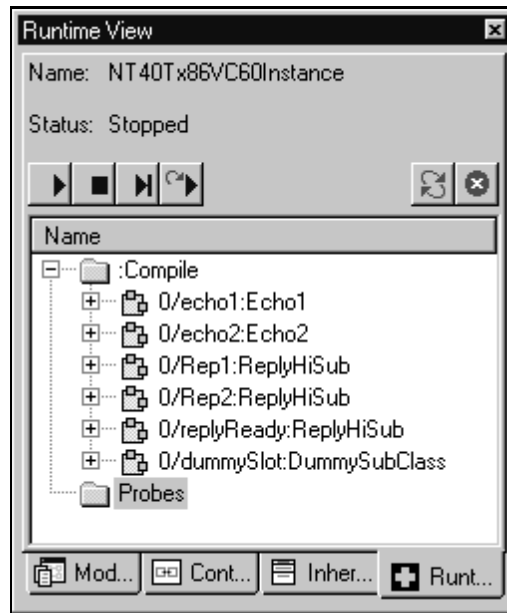
Displays a list of all current breakpoints for the selected component.

## **RTS Browser**

---

The RTS Browser appears as an additional tab on the model browser. It provides an execution interface for controlling the running instance. There is always one RTS Browser for each component instance that is running with observability. The browser is composed of three main parts: an execution control and information pane, a capsule instance browser, and a probes browser.

Figure 123 RTS Browser



## Execution Control and Information Pane

This area shows the name of the component instance, the execution status, and the execution buttons.

- **Start** - Runs the component instance, allowing all messages to be delivered. The button can be pressed after the component instance has been initialized or is stopped.
- **Stop** - Pauses the execution of the component instance. Execution is only paused after the currently executing transition is finished. The button does not halt execution in the middle of a transition. The stop button can be pressed when the model is running.
- **Step** - Allows one message to be delivered in the component instance. Pressing step while the component instance is running allows the current executing transition to finish, then delivers the next message, then pauses or stops. The step button can also be pressed when a component instance is stopped or paused.
- **Restart** - Causes the current component instance to terminate and starts a new component instance. This button can be pressed at any time; however, it is disabled when you are in manual mode. (Reloads when target is loadable.)

- **Refresh** - Updates the status of the capsule instances and probes shown in the browser tree.
- **Shutdown** - Terminates the current component instance and effectively stops the execution environment. All execution monitor windows, watches, traces, and any other execution environment windows are closed.

## Capsule Instance Folder

This list shows all the capsule instances. All the capsule instances are located in the folder named after the top-level capsule. Instances that have not been created yet, for example, optional or plug-in instances, are shown in the browser but with a red 'X' in front of the capsule instance name.

By default only the capsule instances are shown in the folder. The name shown contains the replication index of the instance within the capsule role, the capsule role name into which the instance was created, and the capsule class name of the instance. For example:

```
0/echo1:Echo1
```

You can also view the capsule roles - the roles into which the capsule instances are created - by right-clicking in the **RTS Browser** main window and clicking **Filter > Show Roles**. The syntax is as follows:

```
<replicationFactor>/<portName>/<capsuleInstanceName>:<replicationIndex>_Probe
```

For example:

```
0/log/echo2:3_Probe
*/prot2/echo2:3_Probe
```

## Probes Folder

The **Probes** folder lists the current probes. Right-click on a probe in the list to gain quick access to the common operations performed with probes.

**Note:** The default name for probes includes the capsule instance name and its corresponding replication index.

## Monitors

---

Monitors are read-only views of capsule instances state machine and structure (capsule collaboration) during the execution of the instance. None of the structure or state elements can be modified or moved from the monitor view. The monitor view shows the state and structure components displayed in a lighter shade to emphasize the read-only nature of their parts. Multiple monitor diagrams can exist in the same model.

You can right-click on an element and select **Open Specification...** to open the element's specification. As well, on Composites states, transitions, and choice points, you can select **Show Source Code Location**, which opens a dialog that indicates the location of the code for the action associated with the element.

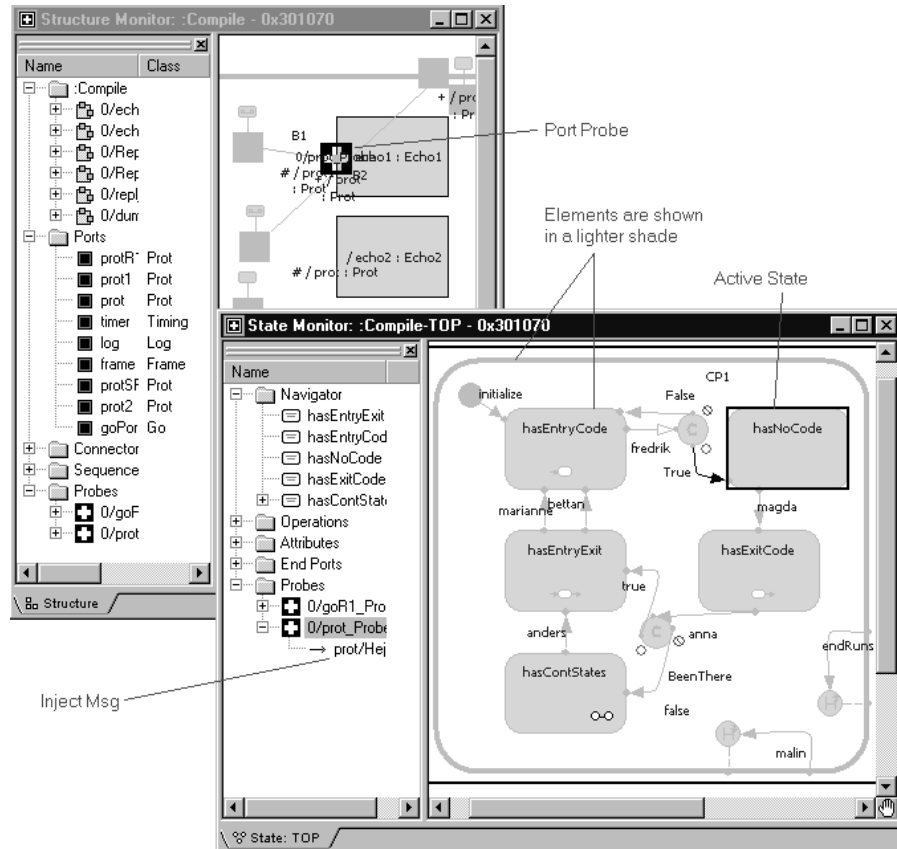
### Animation

To allow observation of state and structure, monitors provide visual clues during execution. The structure monitor shows changes in the dynamic structure by showing optional capsule roles that have not been created with the traditional shading. Once they are created, they are shown as fixed capsule roles. Also, current cardinalities of capsule instances are shown. In order to find specific instances of a replicated capsule role as shown in the structure monitor, you can use the cardinality browser tool.

In the State, or state diagram monitor, the current state is highlighted. In addition, if the state diagram shows hierarchical states, the last active state remains highlighted. When a transition is taken, the state monitor highlights the transition.



**Figure 124 Capsule State and Structure Monitors and Browsers**



## Opening a Monitor

Select a capsule instance from the RTS browser, and from its popup menu, select either **Open Structure Monitor** or **Open State Monitor**.

## Probes

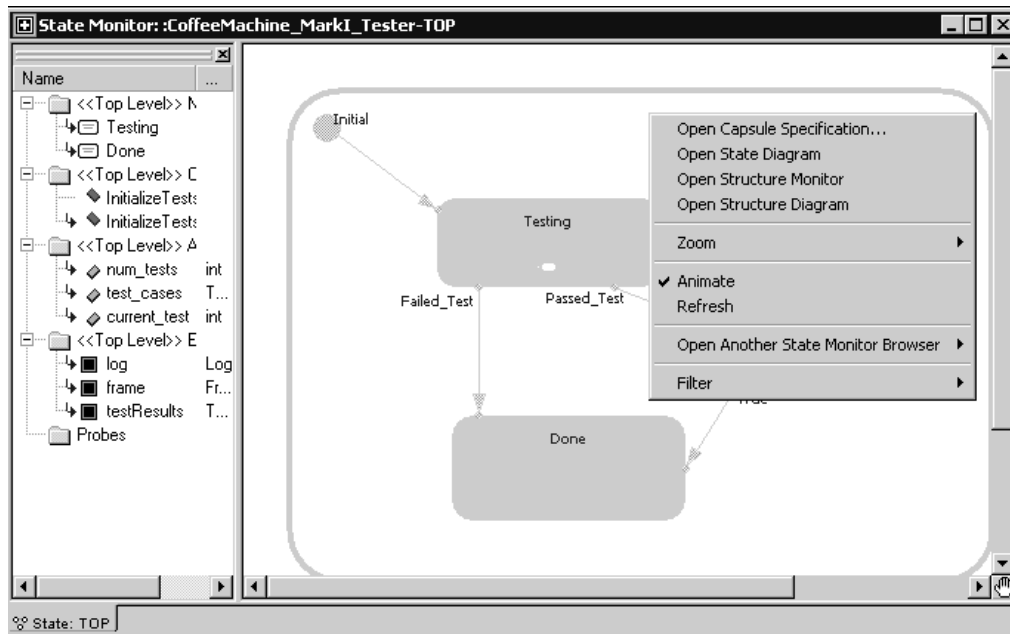
From within a monitor, you can place probes on ports, junction points, and states by using the Probes tool.

## Navigating to Model Elements from Debug Monitors

A very powerful feature of Rational Rose RealTime is the ability to observe a running component instance at the model level. In Rational Rose RealTime, you can see the triggered transitions, the active states in the state diagram monitors, and you can watch the dynamic structure animate in the Structure Monitor.

The context menus for the **State Monitor** (see Figure 125) and **Structure Monitor** include two new options: **Open State Diagram** and **Open Structure Diagram**. These options facilitate navigation to the design elements during a debug session.

**Figure 125 State Monitor - Context Menu**



## Trace Windows

Trace windows are used to log messages sent or triggering events in a running system. Trace windows show lists of messages, including local state, incarnate and destroy, and import and deport messages. Each row in the list corresponds to one message. Rows can be divided into multiple columns, where each column is used to display different details regarding the message in that row.

You can trace without having the Trace window open. Tracing is turned on by opening a Trace window. However, after tracing is started, closing the window does not stop the tracing. Messages are buffered internally in the probe. Tracing is only stopped by deleting the probe.

There are three different types of trace windows. Each type of trace window shows sets of messages captured at different scopes in the system.

Type of trace	Scope of message capture	Default columns shown	Opened by...
Capsule instance trace	Shows messages exchanged between capsule instances	Time, capsule instances, message signal, optional data	Selecting capsule instances in the RTS browser and choosing <b>Open Trace Window</b> from the popup menu.
Port trace	Shows messages coming in or out of a specific port	Time, direction (I/O), priority, signal, data	Creating a probe on a port, then selecting <b>Open Trace</b> from the Probes popup menu.
State trace	Shows messages that trigger and event in the state machine	Time, port, priority, signal, data	Creating a probe on a junction point or state, then selecting <b>Open Trace</b> from the Probes popup menu.

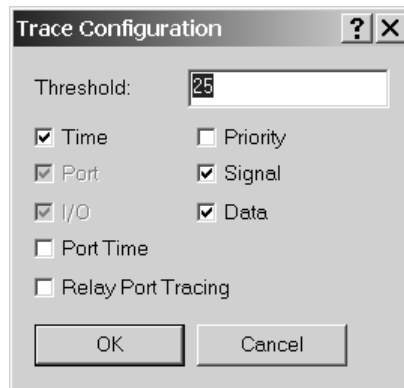
## Deleting Messages

Any message can be deleted from the trace by right-clicking on a message in the list then selecting **Delete** from the popup menu.

## Trace Configuration

You can configure the information displayed in the trace window by right-clicking in the trace window and selecting **Configure...** from the popup menu. The Trace Configuration dialog appears.

**Figure 126 Trace Configuration Dialog**



You can enable Relay Port Tracing by selecting the **Relay Port Tracing** check box. If this check box is selected, all messages between capsule instances, including instances being received through relay ports, appear in a trace. For messages relayed to sub-capsules within a capsule, the trace shows through which ports these messages were passed.

To observe the message passing, convert the trace to a sequence diagram.

## Using Different Types of Traces

Typically when a message fails to flow through a set of capsules as expected, it is important to see where the message flow was first in error. To debug these kinds of errors, first use Capsule instance traces to look at the messages originating and terminating from the capsules in the message flow. If the messages are incorrect and the fault origination cannot be identified, place Probes on specific ports in a composite capsule. Based on whether the messages are still faulty, you can narrow down the cause of the error by further subdivision. Once the faulty capsule has been identified, it is valuable to place traces and message breakpoints on the state machine.

## Opening a Sequence Diagram

Selecting this popup menu item opens a dialog that lets you choose the capsule where the generated sequence diagram is saved.

## Creating a Sequence Diagram From a Trace

In the **Trace** window, after you start a trace, you can right-click on an item to create a **Sequence Diagram** from the trace.

When selected, the **Generate local state information** option generates all the local states into the **Sequence Diagram**. When this option is not selected, the local states will not appear in the **Sequence Diagram**.

## Probes

---

Probes are used to monitor messages passing through ports and events that trigger transitions in a running capsule instance. They are attached to states, junction points, or ports by using the probe tool, which is available when viewing a state diagram or structure monitor of a capsule instance.

Probes can be placed on instances that have not yet been created, for example, even before the component instance is running. Probes are associated with component instances and are stored with them, such that they do not have to be redefined each time the component instance is run. A component instance's probes are listed in the RTS browser, inside the probes folder.

Probe type	Can be created on...	Description
port probe	ports, replicated ports	Port probes allow tracing messages passing through the port, or in the case of replicated ports, messages passing through all instances of the port. They also allow you to inject messages to a port.
state probe	junction points, states	State probes placed on junction points allow tracing of events that trigger the associated transition. Probes on states trace all messages that occur in that state. State probes do not allow message injection. They can, however, be used as state break points to stop the execution of the system when a particular probe has been reached.

Use the **Probe Specification** dialog to configure a probe. You can also use a probe's context menu to quickly open the probe **Trace** window, the **Inject** window, and activate or deactivate a probe.

## Placing Probes on Replicated Ports

If a port is replicated you can place a probe on all instances of the ports by closing the '\*' from the port instance browser. This results in a probe that monitors and injects on all instances of the port. You can also place a probe on a particular instance of a port by selecting a particular instance number from the port Instance browser, then placing the probe on the port.

## Inject Window

---

On a port probe specification sheet the Probe Specification—Detail tab allows you to define messages and send them in or out of the port on which the probe is attached.

Inject messages also appear under the owning port probe in both state and structure monitor diagram browsers. The inject message context menu lets you inject, modify, or delete an inject message. Double-clicking the inject message injects the message.

## Capsule Instance Trace

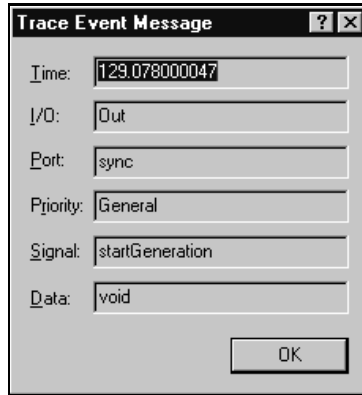
---

A capsule trace window is a type of message trace that shows capsule instances with messages listed in separate columns for recording message flow between instances. The left column displays the time at which the event occurred, the subsequent columns display the source and destination ports, the signal name, optional data, and the capsule instances.

### Trace Event Message Dialog

You can right-click on a capsule trace window and select **Open Specification** to open the Trace Event Message dialog (see Figure 78), which contains information about an event message.

**Figure 127 Trace Event Message Dialog**



## Creating a Sequence Diagram From a Message Trace

You can take a snapshot of a message trace at any time and create a Sequence Diagram. Each interaction in the resulting Sequence Diagram is labeled with the signal name. Message lines can cross one another indicating message overtaking. Since the Sequence Diagram is a snapshot of the trace, it is not updated dynamically.

### To create a run-time sequence diagram:

- 1 Open a capsule instance trace.
- 2 Right mouse click in the message trace window and select **Open Sequence Diagram**.

A sequence diagram is created from the message trace.

**Note:** Only the messages shown in the trace window will appear in the sequence diagram; therefore, if you want to create a sequence diagram with less messages you can pause the running component instance, delete messages from the trace, then create the sequence diagram.

- 3 You can select a saved sequence diagram and select **Open Trace** to reopen another capsule instance trace.

## Dragging Capsule Instances into a Trace

Additional capsule instances can be added to a trace window by dragging and dropping them from the RTS Browser to the trace window. This is useful for configuring the order of instances already in the window, as well as for adding optional instances that were not created at the time the trace was started.

## Message Trace Configuration Dialog

---

This dialog configures a Trace window.

### Threshold Field

An integer value used to specify the maximum number of events displayed in the trace window before discarding on a first-in first-out basis. The default threshold is 25.

**Note:** Messages are also buffered in the running component instance. The larger the threshold, the more memory is allocated in the running component instance. This can be set in the Probe specification threshold.

### Column Check Boxes

These correspond to the list columns in the trace window. You can specify which columns are displayed. Each type of trace has its own default columns that are shown. See the trace window help for details on the default columns for the different types of traces.

## Execution Watch Tab

---

Capsule instance attributes can be inspected at run-time and modified from the Watch tab of the Output window. The watch tab has two columns: the name of the attribute and its value.

To add an attribute instance or variable to the watch window, open a state monitor and drag-and-drop the attribute from the Attributes folder into the watch window.

You can also edit the value of a variable by selecting the Value field then entering another value for the variable.



## Refreshing the Watch Values

The watch values are refreshed when a message is received by the state of the capsule instance. If the state monitor from where the watch was created is closed, the watch value stops being updated. If the state monitor is closed, you can manually force an update of a watch value by right-clicking on the watch item and selecting **Refresh** from the popup menu.

## Run-time Exception While Running a Component Instance

---

A running component instance can crash suddenly with a run-time exception that could be due to either design errors (sending an inappropriate signal through a port, for example) or coding errors (illegal memory references, for example). Rational Rose RealTime will detect that the process is no longer running and display an information dialog warning that the RTS system will be shutdown.

If Purify is installed on your system, and if a component instance running with Purify crashes, the results appear on the Purify output window.



Rational Rose RealTime can help you resolve design errors. For example, problems with state machine logic can be found with a state monitor and message sequencing problems can be found with traces. However, when your model contains detail level coding errors that cause exceptions, the best tool for resolving these problems is your source level debugger. You can add source breakpoints from within a state monitor to automatically launch a source code debugger to help you resolve detail level coding errors.

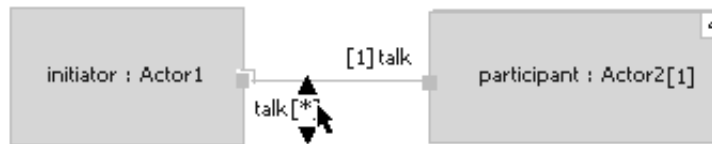
## Instance Browser

---

The instance browser tool is useful for selecting and examining a particular instance of a replicated port or replicated capsule role in a structure monitor.

### To use the instance browser:

- 1 Open a structure monitor that shows either replicated ports or replicated capsule roles.
- 2 Move the cursor over the cardinality field. The cardinality field is shown at the end of a capsule role name or port name in square brackets. Notice that two black arrows appear, one above the cardinality field and another below.



- 3 Select the top or bottom arrow to select a particular instance. For ports you can select the '\*' entry in the instances list to select all port instances.

**Note:** As you change the cardinality you may notice the capsules border and shading change to reflect the state of the instance you are viewing.

## Source Code Debugging

---

In addition to the observability debugging tools, you can also use the native debugging facilities. Occasionally, you have to step through your code to find out what is happening. Rational Rose RealTime can be configured to automatically start up an external source code debugger when a breakpoint is reached.

Using the breakpoint tool from the state monitor toolbox, you can place source code break points on any element in your state monitor that contains detail level code, including

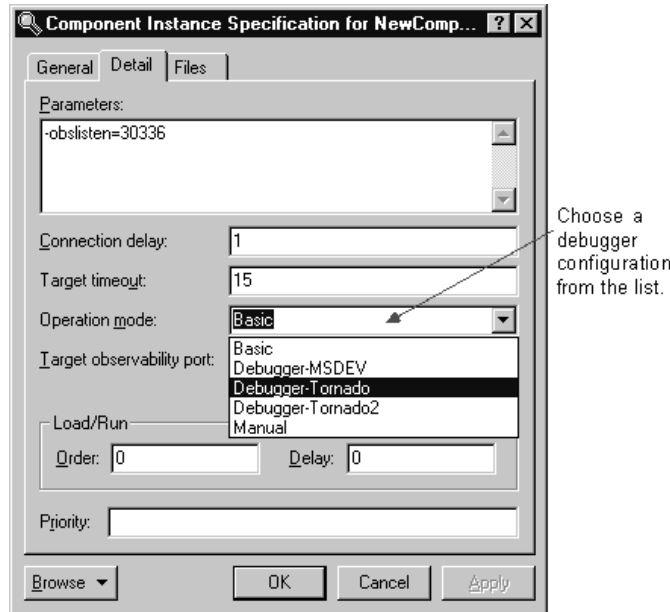
- Transitions
- State entry/exit actions
- Branches

**Note:** Actions map directly to an operation in the source code so that when the external source debugger hits a breakpoint the breakpoint will always be at the beginning of the operation.

To set source code breakpoints when running a component instance, follow these steps:

- 1 Rational Rose RealTime must know that a component instance is to be loaded with the source debugger. The component instances specification (Details tab) controls the selection of the source debugger to use, which is one of the options that is available for target control.

The appropriate debugger must be chosen from the **Operation mode** field in the **Component Instance Specification** dialog.



- 2 Load the source debugger and component instance by choosing **Load** from the component instance right-click menu or the load button from the toolbar.

At this point the source debugger should be loaded and initialized. The component instance has not run yet, hence the RTS Browser is not visible.

**Note:** Do not forget to configure the component to generate debugging information when compiled. Refer to your compiler and linker documentation for the specific flags that should be used to include debug info into an executable. If the component instance is loaded into the source debugger without debug symbols the source debugger will usually inform you of this.

- 3 Run the component instance (see *Running a Component Instance with Purify* on page 477 or *Running a Component Instance without Purify* on page 479).

- 4 Start the component instance, and use the breakpoint tool to add breakpoints on transitions, states (you are prompted for entry or exit breakpoint), and choice points.
- 5 When the breakpoint is hit the debugger pops to the front and displays the source code corresponding to the breakpoint. You can now use the debugger and Rational Rose RealTime to debug your running component instance. Remember, however, that once a breakpoint is hit, you must use the debugger to continue execution of the component instance.

After the source debugger has been loaded, it remains loaded until the Unload command on the component instance is chosen. This means that the source debugger can remain open while the component instance is run and restarted multiple times.

## Source Debugger Integration without Target Observability

---

Rational Rose RealTime includes support for source debugger integration with and without Target Observability (TO). Without TO, you **can** set breakpoints, on state machines and operations.

For information on setting breakpoints on state machines, see *Setting Breakpoints on State Machines* on page 501.

For information on setting breakpoints on operations, see *Setting Breakpoints for Operations* on page 507.

## Setting Breakpoints

---

You can set breakpoints on a state machine or for operations for your C and C++ models. This means that, for example, for non-dynamic models such as C models, there is support for source debugger integration at the state machine and operation level, when there is no Target Observability (TO).

## Setting Breakpoints on State Machines

If you have targets for which sockets are not available, you cannot use Target Observability to debug your models. However, you can use an external debugger and set breakpoints on the **Breakpoint Diagram** for a state machine.

**Note:** On the **Breakpoint Diagram** dialog, you can set breakpoints on a state machine with or without TO; however, the **Open Breakpoint Diagram** context menu command is active only when a model is in the loaded state, or if it is currently running without TO.

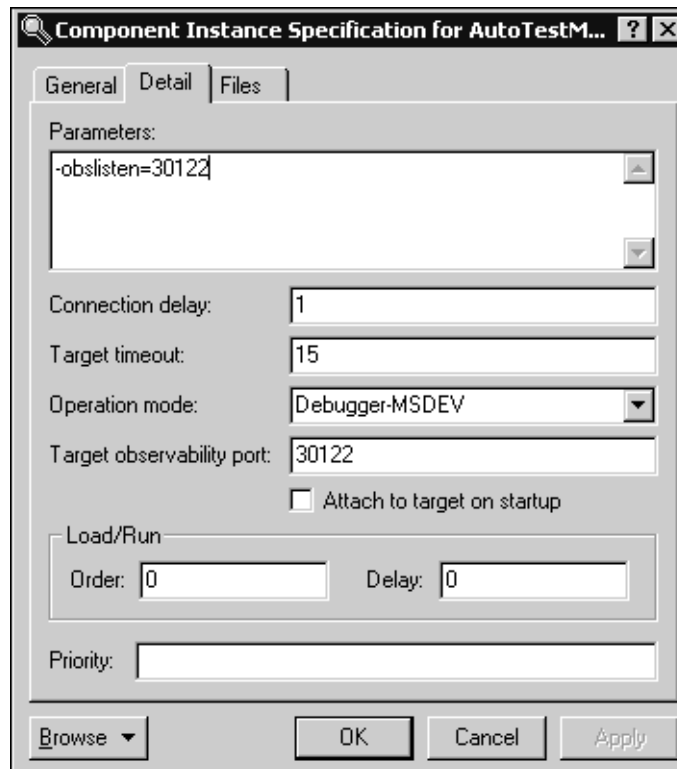
### To set breakpoints on a State Machine without TO:

- 1 In Rational Rose RealTime, open a model.
- 2 Right-click on a processor and select **Open Specification**.
- 3 In the **Operation mode** box on the **Detail** tab, specify a debugger.

For more information on debuggers, see *Using Debugger Modes* on page 521.

- 4 To debug your model without TO, clear **Attach to target on startup**.

**Note:** If you want the toolset to automatically observe a component instance when it is run by the target control scripts, ensure that **Attach to target on startup** is selected.



5 Click **OK**.

**Note:** You must create a debug build of your model, otherwise it will not contain any debug information, and your breakpoints cannot be set and will be disabled. For additional information, see *Starting a Build* on page 441.

6 Build the components for the selected processor.

For more information on building a component, see *Starting a Build* on page 441.

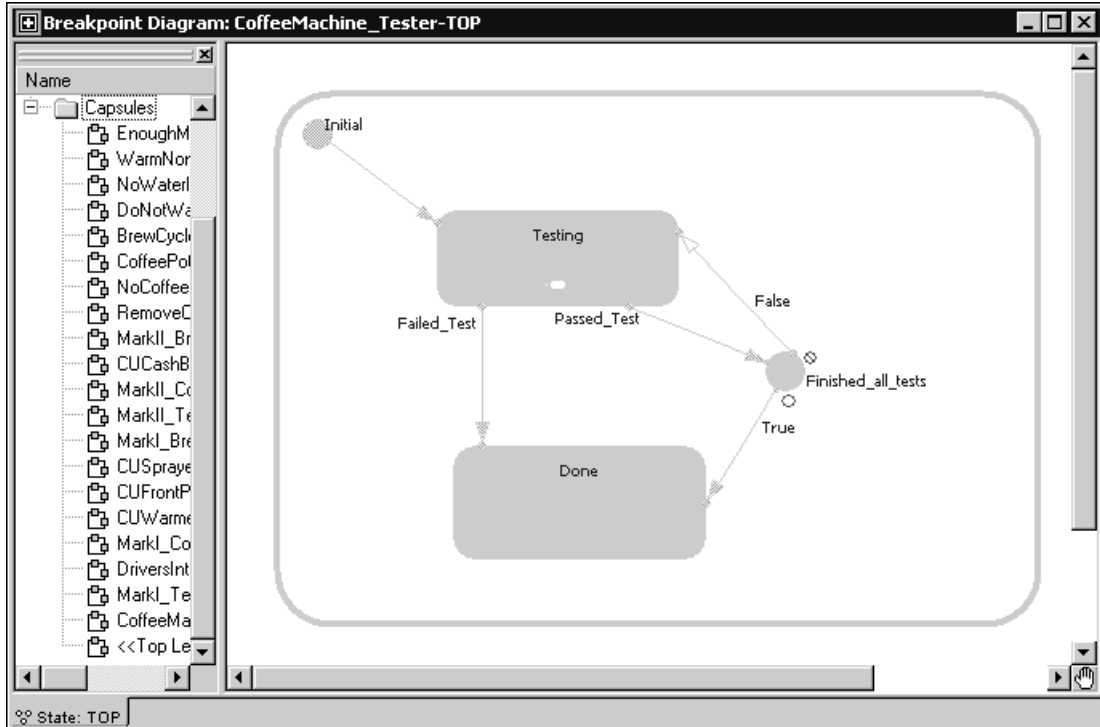
7 In the **Deployment View**, right-click on the same component instance and click **Load**.

The debugger you selected opens your debugging application.

- 8 In the **Deployment View**, right-click on the component instance and click **Open Breakpoint Diagram**.

Opening the **Breakpoint Diagram** dialog opens the state machine diagram for the top state of the top capsule associated with that component instance.

**Note:** Although, the **Breakpoint Diagram** dialog is very similar to a **State Monitor** dialog, you cannot modify the state diagram that appears on the **Breakpoint Diagram** dialog; you can only set breakpoints.

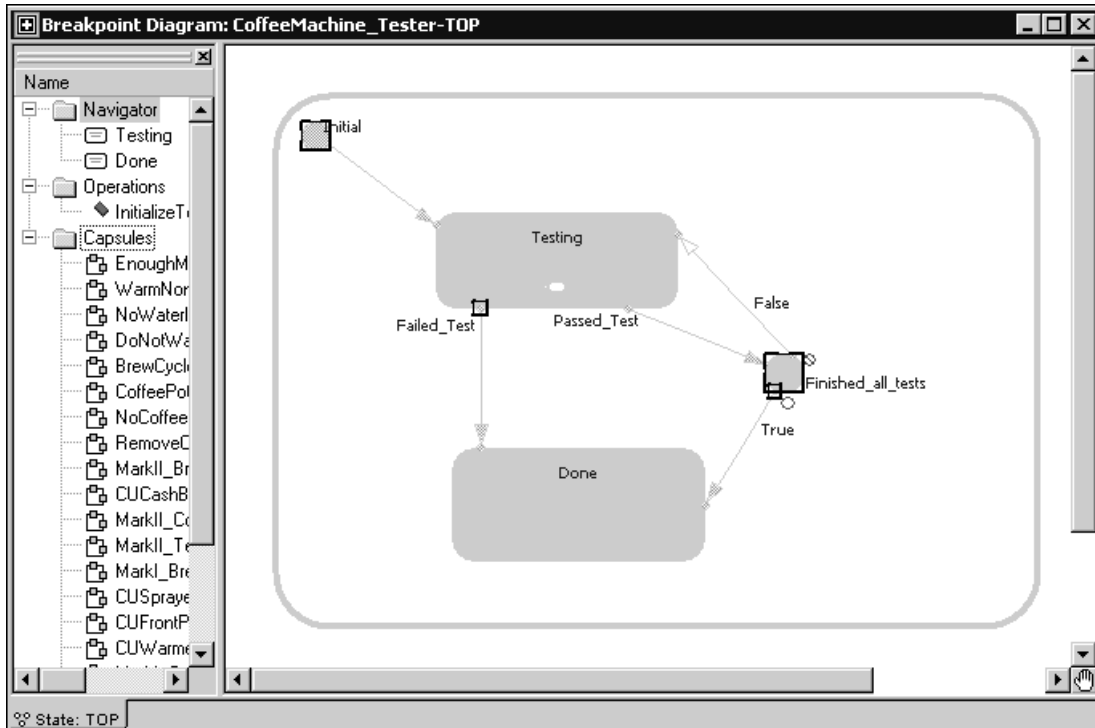


The browser for the **Breakpoint Diagram** dialog shows states similar to that of a typical state monitor browser. However, instead of displaying the elements (such as attributes, end ports, and probes), it shows all the capsules associated with the component for the Component Instance opened in the initial state monitor.

**Note:** To open a **Breakpoint Diagram** dialog for a capsule that is not the top capsule, right-click on that capsule on the **BreakPoint Diagram** dialog, then click **Open Breakpoint Diagram**.

- 9 On the **Breakpoint Diagram** Toolbar, click the **Breakpoint** icon, .

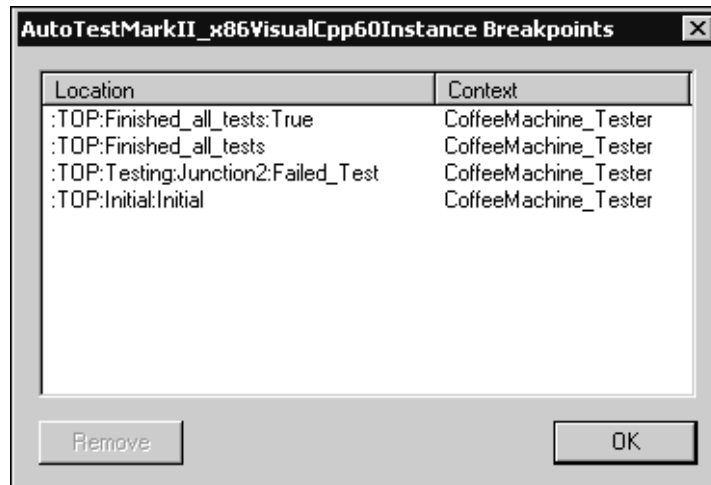
10 Set any desired breakpoints on the diagram.



11 To view all of the breakpoints you specified (those associated with the selected component instance), in the **Deployment View**, right-click on the component instance and click **View Breakpoints**.



**Note:** The **View Breakpoints** menu command is available only when the component instance specifies a debugger in the **Operation mode** box, and the debugger is currently loaded.



**Note:** Unlike the **Breakpoint Diagram** dialog, the **Breakpoints** dialog is available during TO.

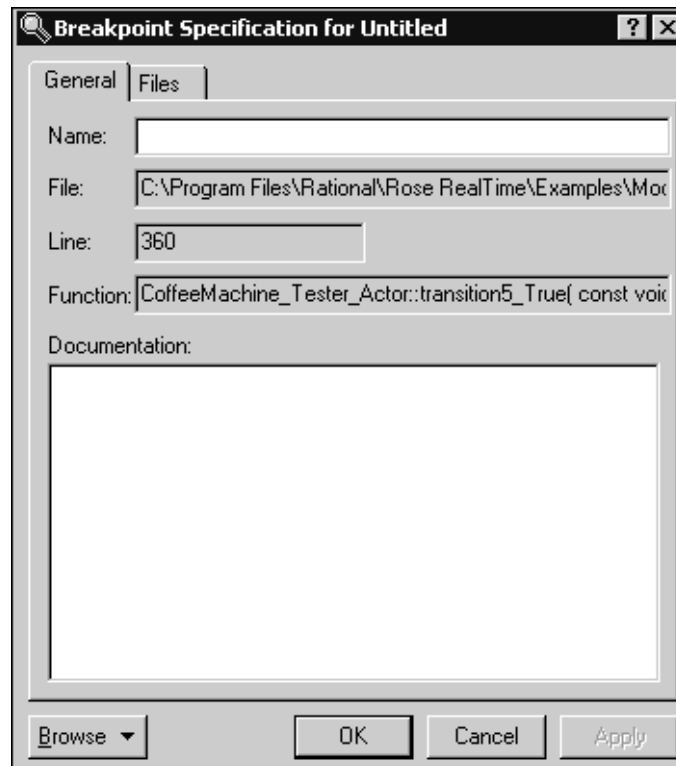
The **Location** and **Context** for each breakpoint displays in the list.

Click **Remove** to delete the currently selected breakpoints.

In the **Breakpoints** dialog, you can right-click and select **Show in diagram** to navigate to the breakpoint in either the **Breakpoint Diagram** dialog or the state monitor, depending on whether TO is used.

12 In the **Breakpoints** dialog, right-click and select **Open Specification**.

The **Breakpoint Specification** dialog shows the file, line number, and function on which the selected breakpoint is set.



13 Click **OK** in the **Breakpoint Specification** dialog and in the **Breakpoints** dialog.

Now, in your debugger, you can look at the breakpoints that you specified.

14 In the **Deployment View** in the **Model View** tab in the browser, right-click on the component instance, and then click **Run**.

15 Maximize your debugging application to view information regarding the breakpoints you set earlier.

16 When finished viewing the breakpoints in your debugger application, return to Rational Rose RealTime, right-click on the component instance, and click **Unload**.

## Setting Breakpoints for Operations

You can set breakpoints on operations, with or without TO.

### To set a breakpoint for an operation:

- 1 Before you can set a breakpoint on an operation, ensure that you:
  - Specify a debugger in the **Operation mode** box on the **General** tab of a **Component Instance Specification** dialog.
  - Create a debug build of your model, otherwise, it will not contain any debug information, and you cannot set breakpoints. For additional information, see *Starting a Build* on page 441.
  - Load the appropriate component instances for the selected processor.
- 2 On the **Model View** tab in the browser, right-click on an operation.

**Note:** The **Add Breakpoint** command exists on the context menu for an operation only when there is at least one component instance that has a debugger specified in the Operation mode box. The **Add Breakpoint** menu item is enabled if at least one component instance is loaded. If more than one component instance is loaded, you must choose the component instance to set the breakpoint.

- 3 Right-click and click **Add Breakpoint**.

The **Remove Breakpoint** command is enabled only when there is at least one breakpoint currently set for the selected operation. If there is more than one breakpoint set on the operation, you can choose the component instance from which to remove the breakpoint.

The **Add Breakpoint** and **Remove Breakpoint** commands are also available for operations on **Breakpoint Diagram** dialogs.

## Customizing Rational Rose RealTime for Target Control and Observability

---

**Note:** Rational Rose RealTime can integrate with other source code debuggers. To add support for target control and observability, and to learn how to integrate Rational Rose RealTime with source code debuggers, see the chapter Customizing for Target Control and Observability in the book *Adapting for Target Environments, Rational Rose RealTime*. The pdf version of this document (rosert\_adapting\_targets.pdf) can be found in \$ROSERT\_HOME/Help.

## Running from Outside the Toolset

---

Binary files or executables built from Rational Rose RealTime do not necessarily have to be run from within the toolset. In some cases it is necessary to run, or even download to another machine (usually a RTOS), the executable manually. For example, this is useful if you are using a target for which the target control scripts and programs are not available.

### Purify

You can run Purify from outside the toolset and import the Purify results into the Rational Rose RealTime.

- 1 When using Purify outside the toolset, run the results.
- 2 Save the results as plain text.
- 3 Import the results into Rational Rose RealTime by going to the Purify pane and selecting **Import** from the context menu.

If the purify output matches a line of code in the model, then the corresponding line of code in Purify appears bold.

### Observability Command Line Parameter

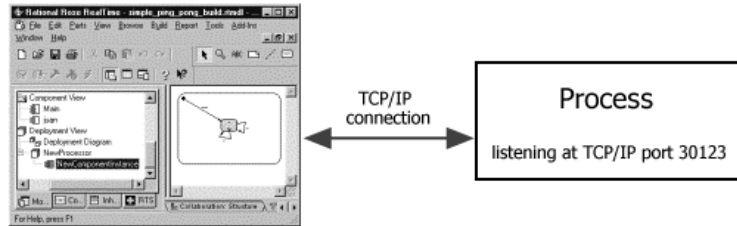
Although the executable is run from outside the toolset, it can still be observed using the observability interface provided by the toolset. If you ensure that the observability command line parameter is passed to the executable before it is run, the toolset can connect to the running model at any time, as well as disconnect. For example, this is how you would start your component instance from outside the toolset:

```
%myProgram -obslisten=30123
```

You can add the following if you want to start running the model immediately:

```
-URTS_DEBUG=go
```

From within the toolset, ensure that in the **Component Instance Specification** the Target observability port is set to 30123. Now, whenever is required, you can use the **Attach Target** option in the component instance popup menu to attach to the running instance.



## Component Instance Menu

To connect to a running process, select a component instance for the correct type of instance that has been run manually. Set the Observability Port in the **Component Instance specification** dialog to the same value that was specified as a command line argument. Then use the **Attach Target** menu option from the component instance popup menu to connect to the running instance.

## Using the Command Line

---

The result of a successful build of a component is an executable module. You can execute this module directly from the command line if the target environment is the workstation itself; otherwise, you have to download it to the target platform.

You can also start the model, or component instance, automatically using the target control capability.

### Command Line Arguments

There are only two Services Library predefined command line parameters that can be used, `-obslisten` and `-URTS_DEBUG`.

```
-obslisten=<tcpip_port>
```

```
%myProgram -URTS_DEBUG=<debugger_command>
```

#### **-obslisten**

This parameter instructs your component instance to listen at the specified TCP/IP port for observability connections from the toolset. For example:

```
%myProgram -obslisten=67887 -URTS_DEBUG=go
```

## **-URTS\_DEBUG=**

Use this option to pass commands to the Services Library command line debugger, which runs automatically when the component instance is started. For example:

```
%myProgram -URTS_DEBUG=quit
```

**Note:** Command line parameters with spaces require quotation marks.

### **quit**

Quits the debugger automatically and allow the process to run freely.

### **continue**

Allows you to start running the target and make TO connections at a later time. From the command line, continue is similar to clicking **Run** in the Toolset; it starts the execution while retaining control (unlike quit which gives up control). For example:

```
MyTopCapsule -obslisten=1234 -URTS_DEBUG=continue
```

## **Application-Specific Command Line Arguments**

You can supply additional command line arguments for use by your component instance model, as you would for any other application. If the component instance is run from the toolset, you can specify command line arguments in the **Component Instance specification** dialog. The arguments are passed on the command line after the name of the executable, for example:

```
%myTopActor foo 99
```

See accessing command line arguments from within a model for more information.

## **Loading and Running Component Instances on Embedded Targets**

---

The requirements for running a process on a host platform and on an embedded platform are somewhat different. For clarification, the term host platform refers to the platform on which Rational Rose RealTime is running. Embedded platform refers to a platform that is not running the toolset. For example, before anything can be run on an embedded target, it must first be loaded or downloaded to the target. This step is not required when simply running on a host platform. It is also common to restart the target board, meaning that a soft reboot is performed.

## Utility Scripts

For the reasons mentioned above, the execution options are different when running on a host platform or on a target platform. In order to support loading, resetting, restarting, and running of component instances on several different target platforms, a set of scripts and executables are invoked from the toolset. Rational Rose RealTime comes with a set of supported target control utilities.

## Component Instance Specification

---

The Component Instance specification dialog contains settings that control the way in which the component instance are run or loaded.

### Specification Contents

The Component Instance specification dialog contains the following tabs: **General**, **Detail**, **Purify** (if installed) and **Files**.

### Component Instance Specification - General Tab

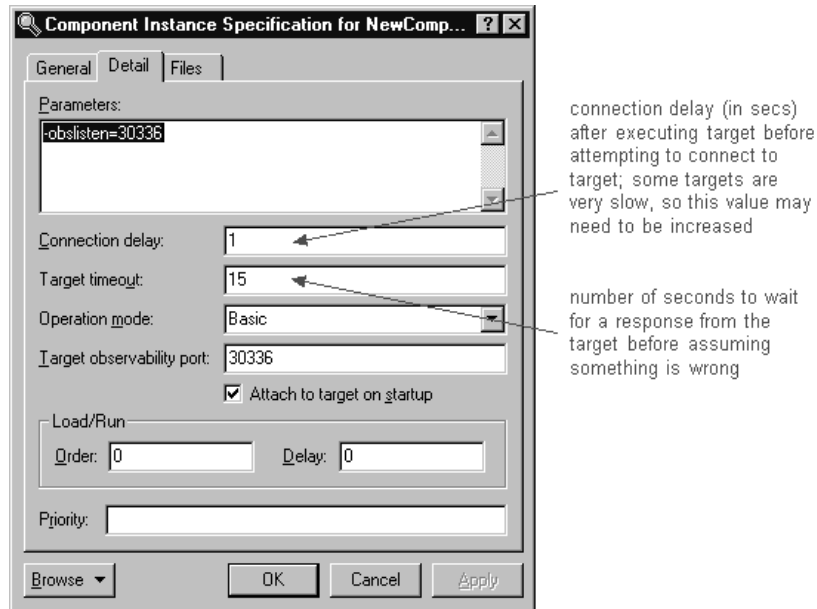
#### Name

The name of the component instance.

**Note:** This is not the name of the actual executable that was created from the build.

## Component Instance Specification - Detail Tab

Figure 128 Component Instance Specification - Detail Tab



### Parameters

Text in this field represents command line arguments that are passed on the command line when the component instance is loaded. The content of this field is passed as is.

### Operation Mode

The operation mode specifies the target control configuration for the component instance. The **Basic** option configures the component instance to use the target control utilities to load and run the component instance. The **Manual** option instructs the toolset not to attempt to load the component instance.

In addition, if you are setting source level breakpoint Probes, you will have to select the debugger that will be loaded by the target control scripts for your platform. Basic mode is implied when one of the debugger options is not selected.



The options are:

- **Basic** - Use the target control utilities to automatically load and run the component instance.
- **Debugger MSDEV** - Use the Microsoft Visual Studio debugger to load and run the component instance, as well as for setting, clearing, and displaying breakpoints. For more information, see *To configure MSDEV Debugger mode*: on page 522.
- **Debugger Tornado** - Use the target control utilities to load and run the component instance with. Use the Tornado debugger for setting, clearing, and displaying breakpoints. For more information, see *To configure Tornado for Debugger mode*: on page 522.
- **Debugger Tornado 2** - Use the target control utilities to load and run the component instance with. Use the Tornado 2 debugger for setting, clearing, and displaying breakpoints. For more information, see *To configure Tornado for Debugger mode*: on page 522.
- **Debugger xxgdb<sup>1</sup>** - Use the GNU xxgdb debugger to load and run the component instance, as well as for setting, clearing, and displaying breakpoints. (UNIX only). For more information, see *To configure xxgdb Debugger mode*: on page 524.

---

1. The **xxgdb** integration works differently from the MSDEV and Tornado. Follow these steps:

1. Build the desired component with the appropriate debug options, for example, **-g**.
2. In the component instance specification, select Debugger-xxgdb for Operation Mode.
3. Start TO.
4. Open the desired State Monitors. You may need to “step” to get access to them.
5. Set breakpoints on the appropriate elements within the desired State Monitor.
6. Restart Target Observability.

Breakpoints are now enabled.

7. Run the model.

8. Remember to continue in the debugger when you hit a breakpoint. The toolset gets no indication that a breakpoint was hit.

9. If you remove breakpoints, they will not take effect until you restart the model again.

- **Manual** - The toolset does not attempt to load the executable. The user must manually load the executable.
- **EMVT**- Use EMVT (Embedded Microsoft Visual Tools) to load and run the WIndows CE component instance. For additional information on using the Windows CE option, see *To configure a component instance for Windows CE, follow these tasks:* on page 518.

## Overview of Observability Options

---

This topic is organized as follows:

- Attach Target Observability on Start-up
- Target Observability Port
- Load/Run

### Attach Target Observability on Start-up

Check this item if you would like the toolset to automatically observe a component instance when it is run by the target control scripts. You can always connect the toolset to the process at some later time.

### Target Observability Port

Specify a TCP/IP port number to use for connecting the toolset's execution environment to the target executable. The port number must not already be in use by another process.

### Load/Run

- **Order** - An integer value representing the relative order in which this component instance is loaded, or run, in relation to other component instances listed and selections in the Build Settings dialog. Lower numbers are run first.
- **Delay** - An integer value representing the number of seconds to delay before the component instance is loaded or run. This is useful when simultaneously running multiple component instances specified in the Build Settings dialog. If you want to Ensure that one component instance has time to start correctly before running the other - for example, if they need to communicate - you can specify a run delay for the second component.

## Observability Options

### Component Instance Specification - Purify Tab

#### Error Call Stack Length

- The maximum number of call stack levels which you want Purify to record for error locations in the program.
- This setting affects whether two errors are considered identical (those with the same message type and error location call stack) and displayed as one message with a count of repeated occurrences, or considered different and displayed as separate messages.
- Purify uses the error location call stack to determine whether a message is a unique or repeat occurrence. Specifying a larger number gives Purify more call stack levels to compare and increases the chances that Purify will display a message as a unique occurrence.

#### Connection Delay

- An integer value representing the maximum number of seconds Rational Rose RealTime takes while attempting to collect process information. This allows Purify time to instrument the executable as necessary. For a large module, you will need to adjust the connection delay to be more than the default of 60 seconds. Then, the toolset waits for the interval specified on the **Connect delay** option on the **Detail** tab before attempting to connect to the target.

#### Default Instrumentation Type

- The level of error checking and coverage monitoring on a per module basis. Select one of the following:
  - **precise** (default) - Provides full run-time error detection and precisely pinpoints problems in any component in the program.
  - **minimal** - Provides quick instrumentation for modules whose errors are of less interest.
  - **exclude** - Excludes DLL's which may cause your program to malfunction when `SetWindowsHook()` is called.

## Display

- **First occurrence only** - Displays only the first occurrence of a message with a count of repeated, identical occurrences, for all Purify sessions
- **Handles in use at exit** - Displays the handles that are in use when you exit a program, for all Purify sessions
- **Memory in use at exit** - Displays allocated blocks of memory, to which there are still pointers, at exit. This allows you to fix large amounts of memory in use in long running programs, to avoid out-of-memory problems
- **Memory leaks at exit** - Displays memory leaks (allocated blocks of memory to which there are no pointers) found when you exit a program, for all Purify sessions

See the Purify documentation for more information and details on Purify, and for descriptions of possible error messages.

## Processor Specification Dialog

---

This dialog allows configuration of the type of processor that this element represents, in addition to the processes (component instances) that will run on the processor.

### Specification Contents

The processor specification dialog contains the following tabs: General, Detail, Files.

### Processor specification - General Tab

#### Name

A name for the processor. The name appears on the deployment diagram, but the name is not used for execution purposes. The actual target id is specified using the address field on the Detail tab.

### Processor Specification - Detail Tab

#### CPU

Name of the type of central processing unit for this processor element.

#### OS

Name of the operating system running on this processor.

## Address

Network address for the processor. This field can contain a hostname or an IP address. For example jhost1 or 145.34.5.6.

**Note:** For systems not connected to a network, you must use 127.0.0.1 in this field.

## Server

In some environments there is a server that handles loading and executing of a component instance for the target RTOS. This is the name or the address of this server.

## Load script

Path to the target control utility directory that contains the scripts and programs that are responsible for loading and unloading processes on that processor. If this field does not point to a valid script directory you will not be able to execute component instances from within the toolset.

## Component Instances

This is the list of component instances that will run on this processor. You can add a component instance to this list by dragging and dropping a component instance from the model browser to this list. Dropping a component instance on a processor results in the creation of a process. You can also right click and select **Insert**. The Create Component Instance dialog appears in which you can select a component to create an instance from and give it a name. See the Processor specification dialog for process details.

## Browse

When you click the **Browse** button, the **Select Directory** dialog appears from which you can locate the Target Scripts directories.

## Using Windows CE

To allow control of component instances for the Windows CE platform, the target control utilities are implemented as a set of external executables and scripts that are invoked from the toolset to perform the various target control tasks.

These scripts and executables for target control are located in the following directory:

```
$Target_scripts = $ROSERT_HOME\bin\tc\win32\wince
```

For a toolset running on a Windows platform, the toolset can control component instances for a Windows CE target platform; a component instance can be run, loaded, and terminated automatically by Rational Rose RealTime.

**To configure a component instance for Windows CE, follow these tasks:**

- *To specify the Windows CE target control configuration for the component instance:* on page 518
- *To configure the Windows CE component instance:* on page 520
- *To run and load the Windows CE component instance:* on page 521
- *To unload the Windows CE component instance:* on page 521

**To specify the Windows CE target control configuration for the component instance:**

- 1 Establish an ActiveSync connection between your Desktop and the Windows CE device.

For information on establishing an ActiveSync connection, see your Windows CE documentation.

- 2 Configure for your Windows CE environment.

Microsoft Embedded Tools includes batch files to configure your environment for different processors. For example in /EVC/WCE300/bin, there is a batch file called WCESH3.bat that sets up an environment for an **sh3** target. Batch files for other targets are available in the same directory.

**Note:** Ensure that the environment variables are configured for your target processor for your specific CPU.

The operation mode specifies the target control configuration for the component instance. For Windows CE, you can specify either **Basic** or **Debugger** modes. The Basic option configures the component instance to use the target control utilities to load and run the component instance. If you want to set source level breakpoint, you can specify a debugger that is loaded by the target control scripts for your Windows CE platform.

For instructions on setting Debugger mode, see *To configure Windows CE for Debugger mode:* on page 523.

- 3 Optional:** To use **Basic** mode, prior to starting Rational Rose RealTime, type the following at the Command Prompt:

```
Set RRT_WINCE_TARGET_DIR=\<directory_name>\
```

where *<directory\_name>* is the name of the location of download the model executable and the TCKill agent for your target. If the directory name is not set, the model executable and the TCKill agent are downloaded to the root directory on the target.

- 4 Start Rational Rose RealTime.
- 5 Open an existing model, or create a new model.
- 6 In the **Model View** tab in the browser, right-click on **Deployment View** and click **New > Processor**.

The **Processor Specification** dialog must be told in which directory to look for the control utilities for the Windows CE platform. The control options on the component instance menu (such as Run and Load) are enabled or disabled depending on the control utilities found in the directory specified for that processor.

- 7 In the **CPU** box, select the appropriate CPU for your **target** processor.
- 8 In the **OS** box, select **Windows-CE**.
- 9 In the **Address** box, specify the network address for the processor.

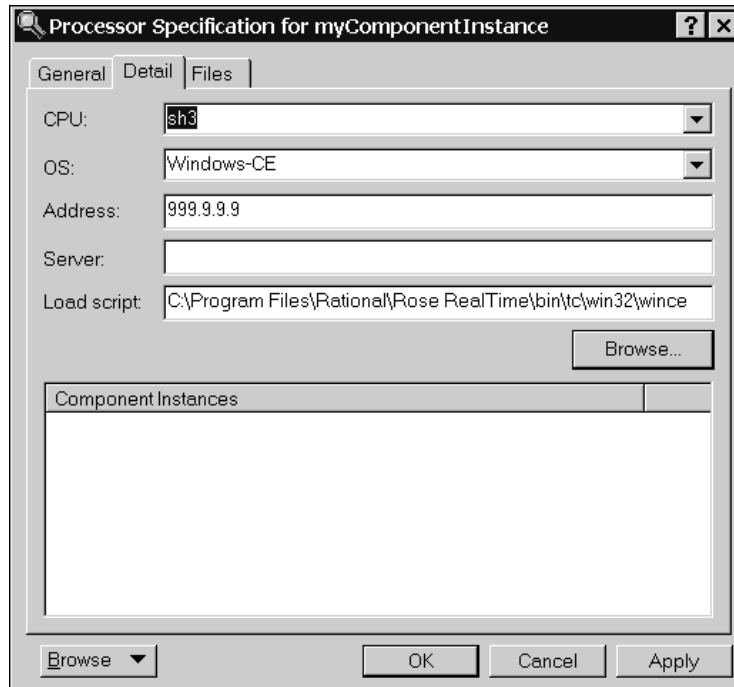
This field can contain a *hostname* or an IP address. For example jhost1 or 145.34.5.6.

- 10 Leave the **Server** box blank.
- 11 In the **Load Script** box, type the following:

```
C:\Program Files\Rational\Rose RealTime\bin\tc\win32\wince
```

**Note:** The path to the target control utility directory that contains the scripts and programs responsible for loading and unloading processes on that processor. You must specify the fully-qualified path. If this field does not contain a valid script directory, you cannot execute component instances from within the toolset.

Your **Processor Specification** dialog will look similar to following:



12 Click **OK**.

#### To configure the **Windows CE** component instance:

- 1 In the **Model View** tab in the browser, drag a component from the **Component View** folder to your Windows CE processor to create a new component instance.
- 2 Select the new component instance.
- 3 Right-click and click **Open specification**.
- 4 Click the **Detail** tab.
- 5 In the **Connection delay** box, specify an integer, (the time in seconds) that specifies how long the toolset waits before listening for a connection from the target.
- 6 In the **Target timeout** box, specify an integer, (the time in seconds) that specifies how long the toolset listens for the connection from the model running on the target.
- 7 In the **Operation mode** box, select **Basic**.
- 8 Click **OK**.



### To run and load the Windows CE component instance:

- 1 In the **Model View** tab in the browser, select the new component instance from the **Deployment View** folder.
- 2 Right-click and click **Load**.

The component instance is loaded onto the Windows CE target.

The component must be successfully built before it can run. If the **Attach Target observability** option was set on the **Component Instance Specification** dialog and a **Target observability Port number** specified, the execution interface displays to allow you to control the execution of the model.

- 3 Right-click the component instance and click **Run**.

On your Windows CE device, the model runs, but it is controlled from the toolset on the Desktop. You can now step through your model and observe its progress in the State Machine.

### To unload the Windows CE component instance:

Because the component instance was loaded onto the Windows CE target, it must be unloaded later.

- 1 In the **Model View** tab in the browser, select the new component instance from the **Deployment View** folder.
- 2 Right-click and click **Unload**.

**Note:** For **Basic** mode, the executable on the Windows CE target device is deleted; the TCKill agent remains on the target. If you wish to remove the TCKill agent, on the Windows CE target, you must manually delete the TCKill agent for your target.

## Using Debugger Modes

You can specify any of the following debugger modes:

- MSDEV - (Microsoft Visual Studio - Windows only)
- Tornado
- Tornado 2
- EMVT - (Microsoft Embedded Visual Tools - Windows only)
- xxgdb - (GNU - UNIX only)

### To configure MSDEV Debugger mode:

To set source level breakpoint probes, you must select a debugger that will be loaded by the target control scripts for your platform.

**Note:** Basic mode (the default) is implied when one of the debugger options is not selected.

- 1 In the **Model View** tab in the browser, select the processor from the **Deployment View** folder.
- 2 Right-click and click **Open Specification**.
- 3 Click the **Detail** tab.
- 4 In the **Operation mode** box, select **Debugger-MSDEV**.
- 5 Click **OK**.
- 6 In the **Model View** tab in the browser, select the component instance.
- 7 Right-click and click **Load**.
- 8 Right-click and click **Run**. If prompted to build the component instance, click **Yes**.

Now, you can set breakpoints and debug your model.

### To configure Tornado for Debugger mode:

To set source level breakpoint probes, you must select a debugger that will be loaded by the target control scripts for your platform.

**Note:** Basic mode (the default) is implied when one of the debugger options is not selected.

If you set the operation mode to **Debugger-Tornado** or **Debugger-Tornado2**, you can set break points and debug your model.

**Note:** Before starting Rose RealTime, you must configure the Tornado environment.

- 1 In the **Model View** tab in the browser, select the processor from the **Deployment View** folder.
- 2 Right-click and click **Open Specification**.
- 3 Click the **Detail** tab.
- 4 In the **Operation mode** box, select **Debugger-Tornado** or **Debugger-Tornado2**.
- 5 In the **Server** box, you must specify the name of server that will be the target server.
- 6 Click **OK**.

- 7 In the **Model View** tab in the browser, select the component instance.
  - 8 Right-click and click **Load**.
  - 9 Right-click and click **Run**. If prompted to build the component instance, click **Yes**.
- Now, you can set breakpoints and debug your model.

### To configure Windows CE for Debugger mode:

To set source level break point probes, you must select a debugger that will be loaded by the target control scripts for your platform.

**Note:** Basic mode (the default) is implied when one of the debugger options is not selected.

If you set the operation mode to **Debugger-EMVT** for the Windows CE target, you can set break points and debug your model.

- 1 In the **Model View** tab in the browser, select a component from the **Component View** folder.
- 2 Right-click and click **Open Specification**.
- 3 Click the **C++ Executable** tab.
- 4 In the **Default Arguments** box, you must specify an argument that instructs the executable on how it will communicate with the toolset. Type the following:  
`-obslisten=<port_on_target_instance>`  
where *port\_on\_target\_instance* is the target observability port.
- 5 Click **OK**.
- 6 In the **Model View** tab in the browser, select the Windows CE component instance from the **Deployment View** folder.
- 7 Right-click and click **Open Specification**.
- 8 Click the **Detail** tab.
- 9 In the **Connection delay** box, specify an integer, (the time in seconds) that specifies how long the toolset waits before listening for a connection from the target.

**Note:** The default value for **Connection delay** is 1 and is not sufficient for this purpose. If you specify 60 in the **Connection delay** box, this time should be quite sufficient.

**10** In the **Target timeout** box, specify an integer, (the time in seconds) that specifies how long the toolset listens for the connection from the model running on the target.

**Note:** If you specify 120 in the **Target timeout** box, this time should be quite sufficient. Depending on the size of your model, you may need to increase this value further.

**11** In the **Operation mode** box, select **Debugger-EMVT**.

**12** Click **OK**.

**13** In the **Model View** tab in the browser, select the component instance.

**14** Right-click and click **Load**.

**15** Right-click and click **Run**. If prompted to build the component instance, click **Yes**.

After the source debugger is loaded, it remains loaded until the **Unload** command for the component instance is selected. This means that the source debugger can remain open while the component instance runs and restarts multiple times.

On the target Windows CE device, it loads the **TCKill** agent for your specific target.

Now, you can set breakpoints and debug your model.

**Note:** For Windows CE, the EMVT debugger mode does not use the **TCKill** agent.

#### **To configure xxgdb Debugger mode:**

Use the GNU xxgdb debugger to load and run the component instance, as well as for setting, clearing, and displaying breakpoints.

**Note:** Basic mode (the default) is implied when one of the debugger options is not selected.

**1** In the **Model View** tab in the browser, select the processor from the **Deployment View** folder.

**2** Right-click and click **Open Specification**.

**3** Click the **Detail** tab.

**4** In the **Operation mode** box, select **Debugger-xxgdb**.

**5** Click **OK**.

**6** In the **Model View** tab in the browser, select the component instance.

- 7 Right-click and click **Load**.
  - 8 Right-click and click **Run**. If prompted to build the component instance, click **Yes**.
- Now, you can set breakpoints and debug your model.

## Unloading a Debugger

### To unload the debugger:

You have to unload target platforms that require loading of modules before they are run.

- 1 In the **Model View** tab in the browser, select the component instance from the **Deployment View** folder.
- 2 Right-click and click **Unload**.

## Device Specification

---

The Device specification dialog contains three tabs: the General tab, the Detail tab, and the Files tab:

### General Tab

#### Name

The name of the device.

#### Stereotype

A stereotype label for the device.

#### Documentation

Use this field to describe the device.

## Detail Tab

### Characteristics

Use the Characteristics text field to specify a physical description of the hardware component. For example, you can describe the kind and bandwidth of a connection, the manufacturer, model, memory, and disks of a machine, or the kind and size of a device. You can set this field only through the specification. This information is not displayed in the deployment diagram.

### Files Tab

A list of referenced files is provided here. The files list popup menu allows you to insert and delete references to files or URLs.

You can link external files to model elements for documentation purposes.

## Connection Specification

---

The Connection specification contains three tabs: the General tab, the Detail tab, and the Files tab.

### General Tab

#### Name

The name of the connection.

#### Stereotype

A stereotype label for the connection.

#### Documentation

Use this field to describe the connection.

## Detail Tab

### Characteristics

Use the Characteristics text field to specify a physical description of the hardware component. For example, you can describe the kind and bandwidth of a connection, the manufacturer, model, memory, and disks of a machine, or the kind and size of a device. You can set this field only through the specification. This information is not displayed in the deployment diagram.

### Files Tab

A list of referenced files is provided here. The files list popup menu allows you to insert and delete references to files or URLs.

You can link external files to model elements for documentation purposes.

## Probe Specification

---

The Probe Specification dialog contains two tabs: General and Files.

### Probe Specification - General Tab

#### Name

The name of the probe, which you can edit if you choose.

#### Activated

Enables the probe.

#### Halt

Halts the execution when this probe detects a message. Upon halting execution, the appropriate **Structure Monitor** diagram opens and this probe will be selected.

#### Trace

Opens the **Trace** window.

#### Threshold

Sets the size of the message buffer on the target.

## Documentation

Use to describe this probe.

## Probe Specification - Files Tab

Allows for linking of external files.

## Probe Specification - Detail Tab

This tab is only available on port probes. It is used to specify and inject messages into the port on which the probe is attached.

### Message list

This tab contains the list of messages that can be injected into the port. The list shows the direction (in/out), priority, signal name, and data of each message.

## Creating Inject Messages

### To create an inject message:

- 1 Right-click in the list and select **Insert** or press the **Insert** key.
- 2 A message editor appears in which you can configure the message that you want to send into or out of the port.

**Note:** You can only choose from the defined signals in the protocol associated with port instance on which the probe is attached.

- 3 Once the message has been defined, press **OK**.

The message appears in the inject list.

### Injected Data Format

The Data area of the inject message is a string representation of the data to be injected with the message. The format of the string depends on the encoding and decoding scheme used by the data type that is being injected.

Therefore, the format of the inject data is linked directly to the encoding and decoding functions. If the encode and decode functions have not been overridden on a data type, the Services Library provides a default ASCII encoder/decoder.

In most cases you will be injecting data using the default ASCII decoder. If this is the case you can use the following syntax to specify the Data area of a message:

**Note:** You do not have to enclose the expression in double quotes.



Default ASCII encoding syntax

```
<type> ::= <type name>{ <attributes> }  
<attributes> ::= <attribute name>{ <attributes> } |  
<basic attribute><basic type>,<attributes> |  
<basic attribute><basic type>  
<basic type> ::= <value> | <basic type>,<value>
```

where

<attribute name> is an attribute of a composite type (e.g., a type composed of other attributes - for example another class)

<basic attribute> is the name of an attribute of a basic type (int, long, short, char, enum, double, float, string)

<value> is the value of an attribute of a basic type

## Examples

### Basic types

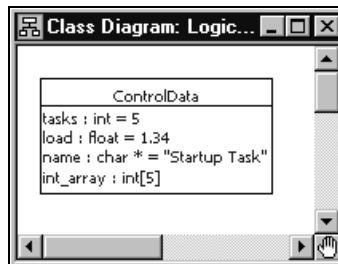
If a signal has a basic type that is a data class

```
int -> int 5
```

```
char -> char'a'
```

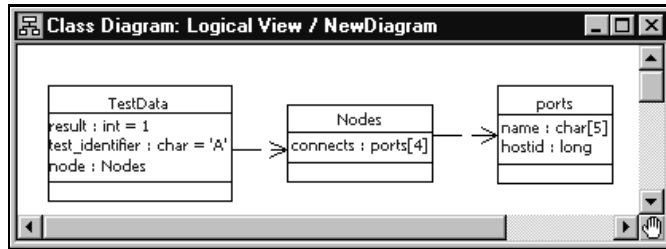
### Classes

Here are two examples of what should be entered into the Data area of an inject message to inject data of the following types. Do not enclose the data in double quotes. The string below each class diagram would be entered as is into the Data area of the inject message.



```
ControlData{tasks 43,load
3.22,name{'N','o','d','e','M','a','n','a','g','e','r','\0'},int
_array 0,0,0,0,0}"
```

**Note:** This example assumes the ControlData class uses the default encode and decode functions with name's NumElementFunctionBody returning a value of 12. If the default encode and decode function are not used, the char \* variable, name, may need to be formatted differently.



```
TestData{result
1,test_identifier'A',node{connects{name'\0','\0','\0','\0','\0'
, hostid 0},{name'\0','\0','\0','\0','\0',hostid
0},{name'\0','\0','\0','\0','\0', hostid
0},{name'\0','\0','\0','\0','\0',hostid 0}}}
```

**Note:** To help determine the format of data types remember that the inject data format will always be the same as you would see the data displayed in a trace window.

## Injecting a Message

To inject a message that shows in the inject list, select the message and from the popup menu choose **Inject**.

If an error occurs parsing the Data area of the inject message, an error will not be returned to the toolset. The message will simply not get injected. The best method of determining whether a message was injected successfully is to open a trace window on the port into which the message is being injected. If the inject is successful you will see the message in the trace.

Inject messages can also be injected, modified, or deleted in the State monitor browser. They are child elements of port probes.

## Contents

This chapter is organized as follows:

- *Code Sync Overview* on page 531
- *Intended Code Sync Usage* on page 532
- *Enabling and Disabling Code Sync* on page 533
- *Identifying Code Sync Areas* on page 533
- *Compiling Code Externally* on page 535
- *Invoking Code Sync from the Toolset* on page 535
- *Reconciling Changes in the Code Sync Summary* on page 535
- *Common Code Sync Errors* on page 536

This chapter describes how you can use Code Sync to make changes to the generated code from outside a model within an IDE (Integrated Development Environment) or editor of your choice, and recapture the changes back into the model.

## Code Sync Overview

---

The purpose of the Code Sync feature is to provide a facility to capture users' changes made to generated code, back into the model. This allows you to externally modify and debug the generated code outside of the toolset.

Modifying generated code helps to reduce the debug cycle on some RTOS's (Real Time Operating Systems), and allows you to make changes using a third-party IDE. Using Code Sync, changes to the generated code can be reconciled and re-integrated back into the "master copy" of the model files.

For the purposes of this feature description, "externally" means "outside the toolset".

## Intended Code Sync Usage

---

### The intended use case for Code Sync is to:

- 1 Build the model from the toolset. See *Starting a Build* on page 441. There must be generated code before Code Sync can function.
- 2 Browse the generated code using a third-party editor or IDE.
- 3 Modify the generated code in designated areas only. See “Identifying Code Sync Areas” on page 533.
- 4 Compile the code externally. See *Compiling Code Externally* on page 535.
- 5 Run the executable externally to test your changes. For more information, see *Running from Outside the Toolset* on page 508. Return to Step 3 above, until the external debugging cycle is complete.
- 6 From the toolset, invoke Code Sync. See *Invoking Code Sync from the Toolset* on page 535.
- 7 From the toolset’s Code Sync Summary dialog box, accept the desired changes. See *Reconciling Changes in the Code Sync Summary* on page 535.

## Limitations

Code Sync cannot be used to create, delete or rename model elements, or to otherwise make structural changes to the model. Such changes must be made using the toolset.

After the generated code has been modified externally, the toolset should not be used to run the externally-built executable until all code Sync changes have been reconciled. For example, although state transitions could be observed and animated by the toolset, the toolset will still show the old transition action code which may be misleading during debugging.

After the generated code has been manually modified, Clearmake cannot provide complete traceability back to model files, and Clearmake cannot provide wink-in. In Clearmake terms, generated code that has been manually modified is no longer considered a *derived object*, but rather a *view-private* file.

## Enabling and Disabling Code Sync

---

In order for the correct Makefile pattern to be generated for Code Sync, Code Sync must be enabled before the code is initially generated from the toolset.

By factory default, Code Sync is enabled on new components. Code Sync can be disabled from the CodeSyncEnabled flag on the Generation tab of each component, if necessary to accommodate a particular *make* utility or to accommodate local coding conventions.

Components that are dependent on a component with Code Sync enabled, do not necessarily need to have Code Sync enabled.

## Identifying Code Sync Areas

---

The generated code can only be modified in certain designated areas. For convenience, these designated areas are tagged using language-specific comments.

### Code Sync Identification Tags

You should only modify code that is delimited by the Code Sync identification tags.

Designated areas for Code Sync are identified in the generated C++ code with the following tags:

```
// {{{USR
<insert or modify code here>
// }}}USR '
```

Designated areas for Code Sync are identified in the generated C code with the following tags:

```
/* {{{USR */
<insert or modify code here>
/* }}}USR */
```

Other similar tags (RME tags) are generated for tracing compilation error messages back to the applicable model element. These tags are irrelevant to Code Sync. Code Sync only recognizes code delimited by the Code Sync identification tags.

## Designated Code Sync Areas

The following areas are designated as available for Code Sync users:

- Action code for transitions in capsules
- Action code for operation implementations in capsules and data classes
- The HeaderPreface, HeaderEnding, ImplementationPreface and ImplementationEnding fields for data classes and capsules
- The CommonPreface field for components
- Guard code for the event triggers on capsule transitions
- Choice-point condition code for capsules
- Entry Action and Exit Action code for capsule states
- The PublicDeclarations, Protected Declarations, and Private Declarations fields for C++ data classes
- The InitFunctionBody, CopyFunctionbody, DecodeFunctionBody, EncodeFunctionBody and DestroyFunctionBody fields for data classes
- The NumElementsFunctionBody field for capsule attributes
- The ConstructorInitializer field for C++ constructor operations

In some cases where a field is omitted or left as its default, the code generator may generate an optimized code pattern that does not provide the empty Code Sync areas or its identification tags. If you wish to use Code Sync area for an area which has been optimized out, you must provide a non-default value for the field (such as a comment) **within the model**, then re-generate before you can modify that Code Sync area.

## Compiling Code Externally

---

Building a model externally is discussed in the *Guide to Team Development - Rational Rose RealTime*. However, since the code will already be generated and manually modified, it is normally sufficient to compile without generating, as shown in the following example:

```
cd /MyHome/OutputDirectory
cd build
make -f Makefile RTcompile
```

In a multi-component model, it is safer to build from the Component Makefile and iterated through each dependent component's compilation. This is particularly true if a header file was manually modified.

```
cd /MyHome/OutputDirectory
make -f Makefile RTcompile
```

Note that this will check for any required generation for each component, then compile each component. If the model has changed, your manual modifications may be lost (overwritten during generation). Consequently, it is recommended that you do not modify the model while you are modifying the generated code.

## Invoking Code Sync from the Toolset

---

To propagate the changes into the model, you need to invoke Code Sync and then decide which changes you want to accept.

Select **CodeSync** from the component's drop down menu. Alternatively, if the component is set as active, click **Build > CodeSync** from the Rose Real Time menu.

Any pending changes to the model-files are written to the file-system. It is not advisable to make further changes at this time, since they will be overwritten upon reconciliation.

If you wish to abort a CodeSync, click the **Stop-Build** icon from the standard toolbar.

## Reconciling Changes in the Code Sync Summary

---

After Code Sync examines the generated code, a Code Sync Summary dialog appears.

This summarizes the differences, for designated code sync areas, between the generated code and the corresponding elements in the model.

## Location

The location within the model element, of the code that was changed by the user.

## Context

The location within the model of the model element, where the changes were made by the user.

## Old code block

The appearance of an element of code within the model. If there is no action code, this block will be empty.

## New code block

The appearance of an element of code from the generated code that has been modified (appears different from the model). If there is no action code, this block will be empty.

## Accepting Changes

### To accept changes:

- 1 From the **Code Sync Summary** dialog, double-click each location you wish to view. The old code block and new code block appears for the selected location.

You can right-click on a change to bring up its context within the toolset. Be sure to return to the Code Sync Summary before modifying the model.

- 2 To reject changes that you do not wish to propagate into the model, deselect the check box(es). These rejected changes may include debug information placed in the Code Sync area while debugging within your IDE.
- 3 Ensure that you have not rejected any code that is required for the model. Click **OK** to accept the selected changes.

Model files are checked out of version control as necessary once the changes are accepted.

## Common Code Sync Errors

---

It is possible to change the model within the toolset before Code Sync is invoked, however, this is not advised. The changed model will be saved when Code Sync is invoked and used during the Code Sync comparison. This can result in either a fatal Code Sync error (if the model changed outside of designated areas), or the model



changes may be interpreted as "old code" in the Code Sync Summary dialog (this may be confusing while reconciling changes, and result in the model changes being overwritten). It is recommended not to change the model before invoking Code Sync.

It is possible to change the model within the toolset after Code Sync is invoked, while the Code Sync Summary dialog is visible; this is also not advised. Code Sync reconciliation is based on the unchanged model, and changes to the model may result in reconciliation results getting lost. You may need to view the model while reconciling Code Sync changes, however, you should not modify the model until the Code Sync Summary dialog has been dismissed (by cancel or accept).

### **Error: Cannot code-sync; file I/O error on: <filename>**

This occurs if the code generator cannot open the expected file during Code Sync, for example, if you have started a Code Sync without a previous code generation.

### **Error: Cannot code-sync <filename> beyond line <lineNum>**

This usually indicates that:

- you have modified the code outside the Code Sync identification tags, or
- you have changed the model (for example, changed the CommonPreface) since it was last generated.

### **Error: Could not find trailing CodeSync tag for [ <LocationSpecifier> ]**

This usually indicates that a starting Code Sync tag does not have a corresponding trailing CodeSync tag, for example, if the trailing tag has been accidentally modified. The Location Specifier (location of modified code, such as ImplementationPreface) and the format of the entire line (including spacing) must match exactly in the two Code Sync tags.

### **Warning: Use tabs for indenting code-sync regions**

The code-generator indents many code-sync regions by one or more tab stops. This warning will appear in the Build Log if, after modification, any line in a code-sync region (including newly-added lines of code) is missing this indentation or is indented with spaces. The region will appear (and continue to reappear) in the Code Sync Summary even if there are no changes to the region. The white-space difference can be resolved by properly indenting the region manually, or by generating the code.

We recommend that you use an editor which indents with tabs. Furthermore, while the tab-width rarely affects the appearance of the generated code, the code-generator assumes a tab-width of eight characters.

## Contents

This chapter is organized as follows:

- *Linking External Files to Model Elements* on page 539
- *Generate Documentation Dialog* on page 540
- *Inserting a Diagram into an MS Word Document* on page 541
- *Using OLE* on page 542

## Linking External Files to Model Elements

---

All model elements can have external files linked to them for maintaining documentation or linking requirements.

To link an external file to a model element:

- 1 Right-click on the model element in the model browser or in a diagram.
- 2 Select **Open Specification** from the selected item's menu.
- 3 Click the Files tab in the **Specification Dialog**.
- 4 Right-click under the **Filename** header.
- 5 Select **Insert File** from the menu.
- 6 Use the File Browser to select the appropriate file to link to.
- 7 Click **Open**.
- 8 Click **OK** to close the **Specification Dialog**.

The link is stored in the model as a relative path. If the file is moved, or the model is relocated, the link may be broken. You can undo and redo the action of adding a link.

# Generate Documentation Dialog

---

The Generate Documentation dialog shows options for creating documentation from the model.

## Report File Name

The name for the report file to be created. A File Browser can be used to select the location by selecting the **Browse...** button.

## Report Title

Give the report a title.

## Report Type

Select the type of document to generate from the following options:

- Logical View Report - generates documentation only for elements in the logical view.
- Component View Report - generates documentation only for elements in the component view

## Attributes and Operations Syntax

- Use Unified Modeling Language Syntax
- Use C++ Syntax

## Report Options

- **Include Operations** - includes all class operations in the document.
- **Include Attributes** - includes all class attributes in the document.
- **Sort** - specifies that the reports appear in alphabetical order
- **Public Operations and Attributes Only** - includes only publicly visible class operations and attributes in the document.
- **Include Documentation** - includes user-specified documentation entered in specification dialogs in the document.

## Generate Selected

Generate documentation for only selected model elements.

**Generate**

Generate documentation for all model elements.

**Cancel**

Cancel the operation.

## Inserting a Diagram into an MS Word Document

---

There are two ways to print a diagram into a Microsoft Word document.

**Option A**

- 1 Click on the diagram you want to put into your document and select **Edit >Select All**.
- 2 Copy the diagram to the clipboard using **Edit > Copy**.
- 3 Position the cursor in the word document where you want the diagram to be placed and select **Edit > Paste**.

**Option B**

- 1 Click on the diagram you want to put into your document and select **Edit >Select All**.
- 2 Click **File > Print**.
- 3 Select **Print to file**.
- 4 Click **OK**.
- 5 Choose the directory in which you want to save the file.
- 6 Type a file name in the **File name** box.
- 7 Click **Save**.
- 8 Open Microsoft Word.
- 9 Select **Insert > Picture > From File...**
- 10 Select the file you saved in Step 6.

**Note:** You will not see the actual diagram in your Microsoft Word document; only a postscript reference is displayed.

## Using OLE

---

OLE is an object-oriented technology, designed for creating, managing and accessing object-based components across process and machine boundaries.

You can create a link between a diagram in your model (the source) and another application such as Microsoft Word. By creating this link, any changes you make to your diagrams are automatically reflected in the document containing the link (the container).

### Creating a Link

After creating and saving your model, copy the contents of a diagram, either by CTRL + C or **Edit > Copy**.

**Note:** If the model is new, it must first be saved for this operation to work.

### Inserting a Link

In an OLE container, for example a Microsoft Word document:

- 1 Select **Edit > Paste Special**.
- 2 Click the **Picture** option and **Paste Link**.
- 3 Click **OK**.

If you select just **Paste** you will get a meta file picture inserted into the container. This meta file is not navigable and becomes native data in the container.

### Navigating

To navigate from your OLE container (for example, your Microsoft Word document) to the application, use the steps for opening OLE linked objects, typically, double-click or **Open** from the **Edit Object** menu. The application opens the diagram independent of the Load of Units setting.

**Note:** Moving your linked files may break the link. It does not, however, affect the object in the container. If the link breaks, you can manually reestablish it from most OLE containers with the Change Source option from the Links dialog.

### Editing Diagrams

Unless the unit is read-only, you can edit your linked (source) diagram. When you modify your diagram, the link is updated to reflect the new state. Depending on the application containing the diagram, you may have to do a manual update to see your changes. Refer to your application manual for details.

## Contents

This chapter is organized as follows:

- *Stereotypes* on page 543
- *Toolset Options* on page 550
- *Add-In Manager Dialog* on page 561
- *Managing Model Properties* on page 562

## Stereotypes

---

This topic describes the following:

- *Creating a Custom Framework for Rose RealTime Models* on page 543
- *Creating a New Stereotype for the Current Model* on page 544
- *Creating a New Stereotype Configuration File* on page 545
- *Creating a New Stereotype for all Rose RealTime Models* on page 545
- *Creating Stereotypes for Classes* on page 548
- *Adding Stereotypes to the Diagram Toolbox* on page 548
- *Creating Stereotype Icons* on page 548
- *Creating a Diagram Icon* on page 549
- *Controlling the Display of Stereotypes* on page 549

### Creating a Custom Framework for Rose RealTime Models

You can create a custom framework from a Rose RealTime model. The contents of the framework define the template to be used when creating new models. For example, if several models with similar characteristics are required, you can create a framework with these characteristics to be used as a template.

### To create a custom framework:

- 1 If you do not have a model file that defines the contents of the framework, create a framework model. You create the framework model in the same way as you would create any other model in Rational Rose Realtime. See *Building Basics* on page 439.
- 2 Optionally, you may create the following files for the model:
  - a documentation file (.TXT) that contains a description of the framework.
  - an icon file (.ICO) that contains the icon to be used as a symbol for the new framework in the Create New Model dialog.
- 3 Select **File > New**. The Create New Model dialog appears.
- 4 Select **New Framework** to enter the Framework wizard.
- 5 When prompted by the wizard, enter the following information:
  - Framework Name - this name will appear as a label for your framework in the Create New Model dialog.
  - Model file - the name of your framework model file (.rtmdl)
  - Documentation file (optional)
  - Icon file, if created (optional)
- 6 Follow the prompts and click Finish to exit the wizard.

## Creating a New Stereotype for the Current Model

You can create a new stereotype by typing a new name in the Stereotype field of a model element's specification. The new stereotype is then available in the Stereotype field for all model elements of that type (which are assigned the same language) in the current model.

If you want the stereotype to be available in all Rose RealTime models, see *Creating a New Stereotype Configuration File* on page 545. If you already have a stereotype configuration file, skip to *Creating a New Stereotype for all Rose RealTime Models* on page 545.



## Creating a New Stereotype Configuration File

The stereotypes in Rose RealTime must be defined in a stereotype configuration file. Rose RealTime is delivered with a default stereotype configuration file, called `DefaultStereotypes.ini`. If possible, add your stereotypes to that file.

### To create a new stereotype configuration file:

- 1 Quit Rose RealTime.
- 2 Create a text file (called, for example, `MyStereotypes.ini`) using Notepad or another text editor, and save it in the Rose RealTime installation folder.
- 3 Edit the new stereotype configuration file. For information on how to create a new stereotype and add it to a stereotype configuration file, see “Creating a New Stereotype for all Rose RealTime Models” on page 545.
- 4 Run the Windows Registry Editor (`regedit.exe`) by selecting **Run** from the **Start** menu. Type “`regedit`” and click **OK**.
- 5 Locate and select the section entitled  
[HKEY\_LOCAL\_MACHINE\SOFTWARE\Rational Software\Rose RealTime\6.5\StereotypeCfgFiles] in the registry list.
- 6 On the **Edit** menu, select **New** and click **String Value**. Give the new registry key the name “file#”, where # is the next consecutive number (1, 2, or 3, etc.).
- 7 Double-click the new key, and enter the name of your configuration file (for example, `MyStereotypes.ini`).

Close the registry. Next time you open a model in Rose RealTime, the stereotypes defined in your new stereotype configuration file will be available in the model.

## Creating a New Stereotype for all Rose RealTime Models

You can quickly create a new stereotype by typing a new name in the **Stereotype** box of a model element’s **Specification** dialog. The new stereotype is then available in the **Stereotype** box for all model elements of that type and language, but only in the current model.

### To create a new stereotype and make it available in all models in Rational Rose RealTime:

- 1 Exit Rational Rose RealTime.
- 2 Optionally, create icons for the stereotype to use in diagrams, lists, and diagram toolboxes. See *Controlling the Display of Stereotypes* on page 549.

- 3 Open the default stereotype configuration file, **DefaultStereotypes.ini** in %ROSBERT\_HOME%.
- 4 In the stereotype configuration file, add a line for the new stereotype in the section called **[Stereotype Items]**. For example, to add the class stereotype **Controller** to an existing configuration file, add a corresponding line as follows:

```
[Stereotype Items]
Class:Model
Class:View
Class:Control
```

- 5 Create a section for the new stereotype and give it the exact same name you specified in Step 4. For example:

```
[Class:Control]
Item=Class
Stereotype=Control
```

- 6 If you created a diagram icon for the stereotype, specify the name of that file (Metafile).

**Note:** You can use the ampersand character, “&”, instead of the folder of the stereotype configuration file. For example:

```
Metafile=&MyStereotypeIconscontroller.emf
```

- 7 To create a diagram toolbox button for this stereotype, specify the name of the file where you created the corresponding small toolbox icon (SmallPaletteImages) and the location of the icon in that file (SmallPaletteIndex). You can also specify the name of the file where the corresponding large toolbox icon is defined (MediumPaletteImages) and the location of the icon in that file (MediumPaletteIndex). For example:

```
SmallPaletteImages=&\MyStereotypeIcons\small_palette_icons.bmp
SmallPaletteIndex=3
MediumPaletteImages=&\MyStereotypeIcons\medium_palette_icons.bmp
MediumPaletteIndex=3
```

- 8 To graphically display this stereotype in specification lists or in the browser, specify the name of the file where you created its list icon (ListImages) and the location of the icon in that file (ListIndex). For example:

```
ListImages=&\MyStereotypeIcons\list_icons.bmp
ListIndex=2
```

- 9 To specify a ToolTip for a stereotype, add descriptive text for customizable option. The format for a ToolTip is:

```
ToolTip=<Name>\n<Description>
```

where:

*Name* is the title for the option that appears on the Customize dialog.

*Description* is the text that appears for the ToolTip.

**Note:** Do not add a space before or after the "\n".

For example, for the

```
[Class:control]
Item=Class
Stereotype=control
Metafile=&\stereotypes\normal\control.wmf
SmallPaletteImages=&\stereotypes\small\control_s.bmp
SmallPaletteIndex=1
MediumPaletteImages=&\stereotypes\medium\control_m.bmp
MediumPaletteIndex=1
ListImages=&\stereotypes\list\control_l.bmp
ListIndex=1
ToolTip=Creates a control\nControl
```

- 10 Add any other settings required to define the new stereotype. For a list of all available settings, information on the meaning of each setting, the possible values, and the default values, please refer to the "Stereotype Configuration File" topic in the online help. Note, however, that you only have to include settings for which you want to give other values than their default values.
- 11 Save your changes to the stereotype configuration file.
- 12 Start Rational Rose RealTime. View the **Log** tab in the Output window to ensure that there are no problems loading your icons.
- 13 If you created a diagram toolbox icon for the new stereotype, and want to add it as a button on a diagram toolbox, see *Adding Stereotypes to the Diagram Toolbox* on page 548.

The new stereotype is now available in Rational Rose RealTime. For information on how to control the display of the new stereotype in diagrams and in the browser, see *Controlling the Display of Stereotypes* on page 549

For detailed samples on user-defined stereotypes, please refer to <http://www.rational.com/products/rosert/>

## Creating Stereotypes for Classes

### To create a stereotype for a class:

- 1 Double-click on the class in the model browser to open the Class Specification.
- 2 Select the General tab.
- 3 In the Stereotype field, type the name of the stereotype for the class, or select it from the pull-down menu beside the field.

You can use any label for the stereotype. It does not have to be one of the built-in stereotype labels.

- 4 Click **OK** to close the specification dialog.

## Adding Stereotypes to the Diagram Toolbox

### To make a stereotype available as a button on a diagram toolbox:

- 1 The stereotype and a corresponding diagram toolbox icon have to be created and made available in Rational Rose RealTime. For information on how to do that, see *Creating a New Stereotype for all Rose RealTime Models* on page 545.
- 2 Select **Tools > Options**, to open the Options Dialog.
- 3 Select the Toolbars tab. Under Customize Toolbars, click on the diagram type you want to change the toolbar for.  
or in an open diagram, right-click in the diagram toolbar and click **Customize**.

The Customize Toolbar dialog is displayed. The left-most column provides the list of available icons.

- 4 Select the icon you want to appear on the diagram toolbar and click **Add**.

## Creating Stereotype Icons

For each stereotype, four different icons may be supplied:

- A diagram icon (to customize the appearance of model elements with this stereotype in diagrams).
- A small and a large diagram toolbox icon (to be able to add a button for this stereotype to the diagram toolbox). Two different sizes correspond to the Use Large Buttons option on the Toolbars tab of the Options dialog.
- A list view icon (to graphically display the stereotype for model elements in specification lists and in the browser).

## Creating a Diagram Icon

Diagram icons have to be in Windows Metafile format (.wmf) or Enhanced Metafile format (.emf) . You can download drawing packages that support these formats at various shareware sites on the Internet. Enhanced Metafiles are recommended if possible.

- 1 Using a vector-based (as opposed to bitmap) drawing application, draw your icon the size you want it to appear in Rose RealTime. It is best not to use a drawing application that forces the icon to fit a certain area, such as a page, as is the case with PowerPoint.
- 2 Consider the following: Make sure that the scaling factor is set to 100% when deciding on the icon's size. Use colors if you like. If you want the name of the model element to appear within the stereotype icon, leave some blank space for it.

Select the icon and export it in either the Windows Metafile format or the Enhanced Metafile format. If you use CorelDraw, make sure the **Include header** option is checked if you save your selection as a Windows Metafile.

## Controlling the Display of Stereotypes

As stereotypes are refined model element types, it is important to be able to distinguish them in the model. The stereotype can be indicated in several different ways in the browser and in diagrams. See "Stereotype Display" on page 556 for more information.

### Controlling Stereotype Display in the Browser

To control how stereotypes are displayed in the browser:

- 1 On the **Tools** menu, click **Options**, and click the Browser tab.
- 2 Clicking **Show Stereotype Name** displays the stereotype name and icon of stereotypes in the browser. Also clicking **Hide stereotype name if there is an icon for it** hides the name and displays only the icon.

### Controlling How Existing Stereotypes Display in a Diagram

To control how existing stereotypes are displayed in a diagram:

- 1 Select the model element in the diagram.
- 2 Click **Diagram Object Properties** from the **Edit** menu, or use the context menu.
- 3 To control the display of relationship stereotypes, use the **Stereotype Label** option.

- 4 To control the display of, for example, a class, a device, or a component, click **Stereotype Display** and select the appropriate option from the displayed menu.
- 5 To control the display of operation and attribute stereotypes in the class compartment, use the **Show compartment stereotypes** option.

## Controlling the Display of Stereotypes Added to Diagrams

To control how stereotypes that are added to diagrams hereafter are displayed:

- 1 Select **Tools > Options > Diagram** tab.
- 2 To control the display of relationship stereotypes, use the **Show labels on relations and associations** option under **Stereotype display**.
- 3 To control the display of, for example, class, device, or component stereotypes, use the **None**, **Label**, **Decoration and Label**, **Label Only** or **Icon** options under **Stereotype display**.
- 4 To control the display of operation and attribute stereotypes in class compartments, use the **Show stereotypes** option under **Compartment**s.

**Note:** For user-defined stereotypes, the stereotype display may be controlled by the settings in the stereotype configuration file where the stereotype is defined. Any such settings override the settings on the Diagram tab of the Options dialog.

## Toolset Options

---

This section describes the Options Dialog, Customizing the Diagram Toolbox, and the Customize Toolbar Dialog.

### Options Dialog

The Options dialog provides control over many general properties of the model.

The Options dialog contains the following tabs: General Tab, File Tab, Font/Color Tab, Diagram Tab, Filtering Tab, Compartments Tab, Browser Tab, THIDC\_AA1oolbars Tab, Editor Tab, and Language/Environment Tab.

There are some fields which are common to many tabs on the **Options** dialog:

#### Type

Specifies a type, such as Class, Protocol, Role and Attribute. The list of types differs depending on the tab selected.

## **Set**

Shows the setting for the specified type. The default is **default**. This field is only available for cloned types.

## **Clone**

Duplicates the selected type.

## **Remove**

Deletes a select cloned type from the **Set** box.

## **General Tab**

### **At Startup options**

#### **Reload last workspace**

Automatically loads the last workspace (and corresponding model) that were in use when the tool was last shut down.

#### **Show splash screen**

Toggles whether the splash screen at appears at startup. The default is set to true.

#### **Show frameworks dialog**

Sets the display of the Frameworks dialog on startup.

#### **Emulate REI**

When enabled, Rose RealTime emulates Rose 2000 from a COM server perspective at startup. This lets you use Rose 2000 Add-Ins with Rose RealTime. Rose 2000 and Rose RealTime are both REI servers. Which one is used to serve a request depends on the specific launched/shutdown sequence of Rose 2000 and Rose RealTime instances that occurred on the server system. Regardless of whether it emulates REI, Rose RealTime always serves as an RRTEI server.

The option can be overridden by the following command line arguments:

- **-emulateREI**: Emulates REI, regardless of the default specified in the option dialog.
- **-noEmulateREI**: Does **not** emulate REI, regardless of the default specified in the option dialog.

## **Picklists**

### **Show Classes**

The ShowClassesInPickLists setting works with selection lists to provide a defined set of types to choose from. The picklists are used to define such things as return types and argument types. You can also change this setting directly in the rose.ini file.

## **Error Log**

### **Log size**

Sets the number of lines in the error log.

### **Log warnings**

Enables warnings to be sent to the log. The default is set to true.

### **Log commands**

Sends a description of executed commands to the log. The default is set to true.

**Note:** To reduce the amount of entries that appear in the log (meaning only warnings are visible) ensure that this option is not set when running RQART.

## **Undo**

### **Undo level**

Sets the number of undo levels supported. A higher number consumes more memory.

## **Technical Support**

### **Email address**

Sets the email address to which error files are automatically sent if the tool crashes. These error files contain only internal callstack information from the tool. They do not contain any model-specific information.

### **Test button**

Tests the given email address to ensure its validity.

## **File Tab**

### **Save options**

#### **Use Temporary File**



Enable this option to write to a temporary file whose name is derived from the destination file. Once the temporary file has been completely written, the temporary file is copied or renamed to the destination file.

### **Create Backup File**

Enable this option to create a backup file.

If Create Backup Files is true and Use Temporary Files is false, and the destination file already exists, the destination file is copied or renamed to the backup file before the Petal file is written to the destination file.

If Create Backup Files and Use Temporary Files are both true, then once the temporary file has been written successfully, the original destination file, if it exists, is copied or renamed to the backup file. The temporary file is then copied or renamed to the destination file.

### **Keep Two Backup Files**

Enables Rational Rose to maintain two backup files: the most recent and the baseline copy. If Create Backup Files is true when writing a backup file, the most-recent generated name for the backup file is deleted. The newly created backup file is assigned the most-recent generated name for backup files. The oldest copy of the file is saved and remains untouched. The oldest version of the file is retained as a baseline copy.

If a backup file does not already exist, a backup file is created and assigned the most-recent generated name.

### **Update by Copy**

Enable this option to manipulate files by copying. The temporary file is copied to the destination and the destination is copied to the backup. If this option is false, manipulation of the files is done by renaming.

### **Save Settings on Exit**

Use this command to save the arrangement of diagram and specification windows and icons when you exit. The next time you open the model, the arrangement that was last saved is displayed.

### **Use spaces in generated file names**

Toggles whether spaces are removed from generated filenames.

### **Always use generated file names**

Use this option to always use generated file names, rather than being queried every time.

## **Load**

When set, the toolset does not query you for missing scratch pad files.

## **Font/Color Tab**

### **Default font**

Invokes the Font dialog, through which you can specify font characteristics.

### **Documentation window font**

Invokes the Font Dialog, through which you can specify font characteristics to be applied to the documentation window.

### **Code font**

Specify a default font for code boxes.

### **Line Color...**

Changes the color of any lines used on diagrams.

### **Fill Color...**

Changes the color of any element fills used on diagrams.

### **Background color...**

Changes the background color for diagrams.

### **Use Fill Color**

You must select the checkbox to see the icons displayed in the colors set in the Line or Fill Color selection. (If it is not checked, a color is defined, but not applied.)

### **Use background color**

Toggles whether to use background color. If not checked, system defaults are used; otherwise, the color specified in Background color... is used.

## Diagram Tab

### Display

#### Unresolved Adornments

Enables adornment of icons representing components not currently loaded in the model. The unresolved view adornment is a small octagon containing the letter “M” with a slash through it.

#### Collaboration Numbering

Enables the display of message sequence numbers on Collaboration diagrams.

#### Sequence Numbering

Enables the display of message sequence numbers on Sequence diagrams.

#### Focus Of Control

The `DefaultViewFocusOfControl` setting is an advanced notational technique that enhances sequence diagrams. Focus of Control is portrayed through narrow rectangles that adorn the vertical lines that descend from each object. You can also change this setting (`DefaultViewFocusOfControl`) directly in the `rose.ini` file. The `DefaultViewFocusOfControl` default setting is Yes.

#### Show message data

Shows the data associated with a message on the diagram.

#### Default line attributes...

Opens the Line Attributes dialog, which let you define line styles, routing, smoothing, and intersecting links.

- **Line style** - Lets you decide whether line styles are oblique or rectilinear. Note that if you choose Rectilinear, Smoothing is grayed out.
- **Routing** - Lets you decide whether routing is normal, closest distance, or avoids obstructions. Note that if you choose Closest distance, Smoothing is grayed out.
- **Smoothing** - Lets you choose how smooth lines are.
- **Intersecting links** - Lets you choose whether to jump links and specify the type of jump. As well, you can choose whether to reverse jump links.

## Miscellaneous

### Double-click to diagram

The DoubleClick setting specifies what action will occur when you double-click an icon representing a logical package or component package.

When selected, this option indicates that a main diagram is displayed when you double-click on the icon. When this option is not selected, it indicates that the specification of a logical or component package is displayed when you double-click on the icon.

**Note:** If this option is selected and you double-click a Capsule Role in a Structure diagram, another Structure diagram opens. If this option is not selected, the Capsule Role Specification opens. Similarly, if this option is selected and you double-click on a State in a State diagram, another State diagram opens. If it is not selected, the State Specification opens.

### Automatic Resizing

Enables the automatic resizing of icons to accommodate text.

### Class Name Completion

Activates a popup box listing all current class names. You can select one of these names by double-clicking or by hitting the Enter or Tab key when you highlight the correct name.

### Auto-adjust transitions

Enables auto-adjusting transitions on creation. The default is set to true.

### Show Diagram Browsers

If not enabled, diagram browsers are not created the first time a diagram is opened.

**Note:** Once a diagram is opened, its state is saved in the workspace, so this option has no effect.

### Stereotype Display

Use the options to control the display of stereotypes in diagrams. The selection is applied to new model elements (except relationships) that are added to diagrams hereafter.

- **None** - The stereotype is not indicated for new model elements.
- **Label** - The stereotype name is displayed for new model elements. The stereotype name appears inside angle brackets, << >>.

- **Decoration and Label** - The stereotype icon (if it exists) is displayed as a decoration in the upper right hand corner of the view. The label is displayed just under the decoration centered above the name.
- **Decoration Only** - The stereotype icon (if it exists) is displayed as a decoration in the upper right hand corner of the view. No label is displayed.
- **Icon** - The stereotype icon (if it exists) is displayed for new model elements.
- **Show labels on relations** - enables the display of stereotype labels on new relationships. The stereotype names appear inside angle brackets, << >>. The selection is applied to new relationships that are added to diagrams.

To display/hide the stereotype name of a previously created relationship in a specific diagram, select the relationship in that diagram. Select **Edit > Diagram Object Properties > Stereotype Display**. On the displayed menu, select the appropriate option. You can also use the same option on the shortcut menus.

If you want to change the display of previously created stereotypes in a specific diagram, select the stereotype in that diagram. Select **Edit > Diagram Object Properties > Stereotype Display**. On the displayed menu, select the appropriate option. You can also use the same options on the popup menus.

## **Grid**

### **Grid Size**

Specifies the grid pitch in pixels. The value that you enter in the Grid Size edit box is saved to the GridSizeX and Y settings.

### **Snap to Grid**

Indicates that new or moved icons will align with a grid whose pitch is specified by the grid size.

## **UML Options**

### **Aggregation whole to part**

Controls which way an aggregation can be drawn. Aggregates can be drawn whole (client) to part (supplier) or vice versa. The default is set to true.

### **Classifier name on roles**

Lets you turn off the classifier name portion of a role label.

### **Protocol name on ports**

Lets you turn off the classifier name portion on ports.

### **Base UML notation**

Converts the structure diagram so that it uses only UML base notation.

### **Target Observability**

#### **Animation timeout**

Sets a delay for displaying animation of events (state changes) in the state monitor. The delay value is in 1/100ths of a second, i.e., a value of 100 will delay event animation for 1 second. This provides the ability to slow down the animation of a model to make state changes more observable.

### **Filtering Tab**

#### **Class Diagram**

Filters information on the class diagram.

#### **State Diagram**

Filters information on the state diagram.

#### **Structure/Collaboration Diagrams**

Filters information on Structure/Collaboration diagrams.

### **Compartments Tab**

#### **Class**

Display/hide information in the compartments of a class on the class diagram.

#### **Capsule**

Display/hide information in the compartments of a class on the class diagram.

#### **Protocol**

Display/hide information in the compartments of a class on the class diagram.

## Browser Tab

### Stereotypes

#### Show stereotype names

Enable or disable viewing of stereotype names of model elements in the browser.

To display only stereotype icons (if any), select the **Hide Stereotype name if there is an icon for it** option.

To display both stereotype icons (if any) and stereotype names, clear the Hide Stereotype name if there is an icon for it option.

#### Hide stereotype name if there is an icon for it

The **StereotypeBitmapsOnly** setting enables or disables stereotype icons, but not stereotype names, of model elements in the browser. This setting can also be changed in the rose.ini file.

### Class and package name display

#### Show related components

Use this option to toggle whether to decorate referenced components in the browser.

## Editor Tab

### External editor

Specify an external editor to be launched when editing detailed code.

**Note:** If you use an external editor that requires a console terminal, you must specify an application, such as **xterm**, that provides the terminal, followed by the editor command itself.

Example on Solaris: `/usr/openwin/bin/xterm -e /bin/vi`

Example on HPUX: `/usr/bin/X11/xterm -e /bin/vi`

## THIDC\_AA1oolbars Tab

The standard toolbar and diagram toolbox properties can be set on the Toolbar tab. The choices are grouped as follows:

- **Standard toolbar**
  - **Show Standard Toolbar** - Toggles whether the standard toolbar is visible.
  - **Enable docking** - Toggles whether to allow the toolbar to be docked.
  - **Use large buttons** - Toggles whether to display small buttons or large buttons on the toolbar.
- **Diagram toolbar**
  - **Show Diagram toolbar** - Toggles whether the diagram toolbox is visible.
  - **Enable docking** - Toggles whether to allow the toolbox to be docked.
  - **Lock selection** - Toggles whether to lock the current toolbox selections.
  - **Use large buttons** - Toggles whether to display small buttons or large buttons on the toolbox.
  - **Auto show** - Toggle whether the toolbox is displayed for read-only diagrams.
- **Customize toolbars** - Provides a list of toolbars whose layout can be customized. Click on a toolbar button to bring up the Customize Toolbar Dialog for that particular toolbar.

## Language/Environment Tab

### Default Language

Select the language from the available installed language add-ins. When a new class is created, this selection determines which:

- language property tab is displayed for classes
- set of fundamental types is used for picklists
- set of predefined stereotypes is used

When a new component is created, the language is set using this default.

If you do not have any language add-ins, the default language is set to Analysis, which is equivalent to having no default language. If this is the case, analysis types are shown in the picklists and no language property tabs are available.



## Default Environment

Sets the default environment on new components.

## Customizing the Diagram Toolbox

You can access the Customize Toolbar dialog using any of the following:

- Right click anywhere on the toolbox and then click **Customize** from the shortcut menu.
- Double-click anywhere on the toolbox not occupied by a button.
- From the **View** menu, point to **Toolbars** and click **Configure**.

With the exception of the **Separator** button, only one instance of any tool can be placed on the toolbox. Since multiple instances of the **Separator** button are allowed on the toolbox, this button is always available regardless of the number of times it is added to the **Toolbox** buttons list.

## Customize Toolbar Dialog

The customize toolbar dialog (opened from the Options Dialog), allows you to change the arrangement of buttons on various toolbars.

## Toolbar Button List

The Toolbar buttons list contains the ordered list of all the buttons that will appear on the diagram toolbox. Once buttons are moved onto this list they can be moved to any position.

## Add-In Manager Dialog

---

The **Add-In Manager** dialog is used to view, activate or deactivate Rose RealTime Add-Ins.

The dialog shows the add-ins currently loaded, with check boxes beside the add-ins showing which ones are currently activated.

## Managing Model Properties

---

Each Rational Rose RealTime model has its own default properties. These default properties are defined in a property file and are grouped into sets based on:

- **Type of model element** - Class, component, relation, attributes, operations, etc - the objects that make up the model
- **Tool** - Corresponds to a tab in the property specification; a tool can be a programming language tool, such as Java or C++; a database tool, such as Oracle8; a user-defined add-in to Rational Rose, or some other tool.
- **Properties** - The actual properties and property values defined in the set; these must be appropriate to the model element and tool for which they are being defined.

**Note:** You can define multiple sets of default properties for the same tool and model element. For example, you might want one set of properties for a class with a stereotype of Actor and a different set of properties for a class with a stereotype of Interface. Both of these sets are still considered default properties in that they are predefined for the model. Defining multiple sets saves you work by minimizing the need to override properties as you go.

### Displaying or Modifying the Values of Model Properties

- 1 Display a diagram that contains an icon representing the model element.
- 2 Select the model element in the diagram.
- 3 Open the model element's specification. To do so, double-click on an element in a diagram, or click on the element and select **Browse > Specification**.
- 4 Select the **Code Generation** tab. The model property set attached to the element is displayed in the **Set** field. The model properties related to the model element are displayed in the **Model Properties List**.
- 5 To edit a model property value, select it and click on it a second time. This places the model property in edit mode.
- 6 Select your choice from the drop down menu. If no drop down menu is available, you may type in your changes.
- 7 To complete the edit, click outside the edit box.
- 8 Click **OK** or **Apply** to commit the changes to the item.

Model properties that are specified explicitly by the item, and hence override the attached model property set value, are drawn in normal text. Model properties that have been changed since the last apply are indicated by an asterisk in the left column.

## Removing an Overriding Item Level Model Property

Editing a model property automatically makes it an overriding item-level model property.

**To remove the overriding value from the item and once again inherit from the attached model property set:**

- 1 Select one or more model properties and click **Default**.
- 2 Click **OK** or **Apply** to commit the changes to the item.

## Making a Model Property Item Specific

- 1 Select the model property(s) and click **Override**.
- 2 Click **OK** or **Apply** to commit the changes to the item.

## Reinstalling the State and Value of the Last Committed Change

Select the model property(s) and click **Revert**.

## Attaching a Model Property Set to a Single Element or a Collection of Elements

- 1 Display a diagram that contains an icon representing a model element.
- 2 Select the model element in the diagram.
- 3 Open the model element's specification. To do so, double-click on an item in a diagram, select a diagram item and execute the **Specification** command in the **Browse** menu, or select the specification from the shortcut menu.
- 4 Select the **Code Generation** tab. The model property set attached to the item is displayed in the **Set** field. The model properties related to the model item are displayed in the **Model Properties List**.
- 5 Select a different model property set from the **Set** combo box.
- 6 Commits are made as you move from page to page. Also, as you move from set to set or type to type within the set-level model property page, any changes you have made to the currently displayed set are committed.

## Displaying or Editing a Specific Model Property Set

- 1 Select the element from the diagram. If you are selecting a collection of elements, ensure that all the elements are of the same type. Selecting different model elements will result in a warning.
- 2 From the **Tools** menu, select **Model Properties > Edit**. The code generator displays the **Code Generation** tab of the **Options** dialog. The kind of model item chosen is displayed in the **Type** field.
- 3 Select the model property set name in the Set combo box. All the model properties and values will be displayed.
- 4 Modify model property set values by following instructions to edit a specific model property set, as listed above.
- 5 Click **Apply** or **OK** to accept your changes.

**Note:** Changes made to a model property are accepted whenever you activate ANY control in the editor. For example, after editing a model property, you may select another model property to both accept the changes to the original model property and begin editing the newly selected model property.

## Creating a New Model Property Set

- 1 Select a model property set from the Set combo box to base your new model property set off of.
- 2 Click **Clone**.
- 3 Type the new model property set name in the dialog and click **OK**. A new model property set is created as a copy of the current model property set.
- 4 Modify model property set values by following instructions to edit a specific model property set, as listed above.

## Deleting a Model Property Set

- 1 Select a model property set from the **Set** combo box.
- 2 Click **Remove**. The model property set is deleted from the model. An attempt will be made to find all the elements in the model that reference that set and change those elements to reference the default model property set.

# Keyboard Shortcuts

# A

## Contents

This chapter is organized as follows:

- *General Shortcuts* on page 565
- *Editing Shortcuts* on page 568
- *Debugging Shortcuts* on page 569
- *Rational Rose RealTime Keyboard Shortcut Summary* on page 571

## General Shortcuts

---

**Table 2** General desktop navigation

Key Name(s)	Description
ALT + PgDn SHIFT + ALT + PgUp	Previous specification.
ALT + PgDn SHIFT + ALT + PgUp	Next specification.
ALT or META + key	Display the contents of a menu - in combination with the underlined letter in the menu's name
ALT + ALT or Shortcut Menu Key	Display the context menu for selected model element. <b>Note:</b> On Unix, by default clicking the ALT key activates a context menu, so pressing it once changes the focus to the application's main menu. Unix keyboards have two additional keys, a left and right diamond, which are similar to the ALT key. The left diamond works the same as the ALT key by making the main menu active. If you use the right diamond, you must press it three times to change the focus to the main menu.
CTRL + TAB	Move between windows
CTRL + Q	Hide or restore the browser, Output window, Documentation tab, and Specification dialogs.
CTRL + 3	Show/Hide Documentation/Code window.
CTRL + 4	Show/Hide Output window.
CTRL + 5	Show/Hide Specification History window.

Key Name(s)	Description
ESC	Close an open menu or cancel a dialog
ENTER	Perform the action in a dialog
TAB	Move forward between areas of a dialog.
SHIFT+F10	Display the context menu. <b>Note:</b> On UNIX, you will need to click SHIFT + F10 twice to display the context menu.
SHIFT+TAB	Move backwards between areas of a dialog.
SPACE BAR	Select an item in a dialog

**Table 3 General Toolset shortcuts**

Key Name(s)	Description
ALT + ENTER	Activates Hot Link on Specification dialogs.
ALT + LEFT	Opens the previous specification in the Specification History list.
ALT + SHIFT + LEFT	Opens the previous specification without closing the current Specification dialog.
ALT + RIGHT	Opens the next specification in the Specification History list.
ALT + SHIFT + RIGHT	Opens the next specification without closing the current Specification dialog.
CTRL + +	Go Inside
CTRL + -	Go Outside
CTRL + A	Select All
CTRL + B	Browse specification
CTRL + C	Copy
CTRL + E	Expand
CTRL + F	Find - displays the <b>Find</b> dialog
CTRL + SHIFT + F	Replace
CTRL + I	Zoom in
CTRL + SHIFT + R	Relocate
CTRL + L	Change line attribute

Key Name(s)	Description
CTRL + M	Zoom to selected
CTRL + N	Opens a new window for editing, or opens the <b>Create New Model</b> dialog
CTRL + P	Print
CTRL + SHIFT + P	Edit Path Map
CTRL + O	Open
CTRL + R	Browse referenced items
CTRL + S	Save
CTRL + T	Browse state diagram (Creates a diagram if one does not currently exist)
CTRL + SHIFT + T	Browse structure diagram (Creates a diagram if one does not currently exist)
CTRL + U	Zoom out
CTRL + V	Paste
CTRL + W	Fit to window
CTRL + X	Cut
CTRL + Y	Redo
CTRL + Z	Undo
DEL	Delete
ESC	Cancel
F1	Context-sensitive help
SHIFT + F1	Context sensitive help cursor
F2	Refresh
F3	Browse previous diagram
F4	Browse parent
CTRL + F6	Browse next pane
CTRL + F10	Browse component diagram
CTRL + SHIFT + F6	Browse previous pane

Key Name(s)	Description
SHIFT + F6	Browse class diagram
SHIFT + F7	Browse use case diagram
F8	Edit inline
SHIFT + F8	Browse collaboration diagram
SHIFT + F9	Browse sequence diagram
SHIFT + F11	Browse deployment diagram
F12	Options

## Editing Shortcuts

---

**Table 4 Scripting Shortcuts**

Key Name(s):	Description
CTRL + O	Enable or disable word wrap.
UP ARROW	Moves the insertion point up one line.
DOWN ARROW	Moves the insertion point down one line.
LEFT ARROW	Moves the insertion point left by one character position.
RIGHT ARROW	Moves the insertion point right by one character position.
PAGE UP	Moves the insertion point up by one window.
PAGE DOWN	Moves the insertion point down by one window.
CTRL + PAGE UP	Scrolls the insertion point left by one window.
CTRL + PAGE DOWN	Scrolls the insertion point right by one window.
CTRL + LEFT ARROW	Moves the insertion point to the start of the next word to the left.
CTRL + RIGHT ARROW	Moves the insertion point to the start of the next word to the right.
HOME	Places the insertion point before the first character in the line.
END	Places the insertion point after the last character in the line.
CTRL + HOME	Places the insertion point before the first character in the script.
CTRL + END	Places the insertion point after the last character in the script.



Key Name(s):	Description
CTRL + SHIFT + D	Duplicate
CTRL + SHIFT + R	Relocate

## Debugging Shortcuts

**Table 5     Debugging Shortcuts**

Key Name(s):	Description:
CTRL + A	Select all
CTRL + C	Copy
CTRL + F	Find
CTRL + G	Go to line
CTRL + H	Replace
CTRL + N	New script
CTRL + O	Open script
CTRL + P	Print
CTRL + SHIFT + P	Edit path map
CTRL + R	Replace
CTRL + V	Paste
CTRL + X	Cut
CTRL + Y	Redo
CTRL + Z	Undo
DEL	Delete
ENTER or F2	Displays the <b>Modify Variable</b> dialog for the selected watch variable, which enables you to modify the value of that variable.
F5	Runs the current script.
SHIFT + F5	Stops script execution
CTRL + SHIFT + F5	Restarts the current script beginning with the line at which it was stopped using the Break command.

Key Name(s):	Description:
F7	Compiles the current script without executing it
F6	If the watch pane is open, switches the insertion point between the watch pane and the edit pane.
F9	Sets or removes a breakpoint on the line containing the insertion point.
SHIFT + F9	Displays the <b>Add Watch</b> dialog, in which you can specify the name of a BasicScript variable. The Script Editor then displays the value of that variable, if any, in the watch pane of its application window.
F10	Steps through the script code line by line without tracing into called procedures.
F11	Steps through the script code line by line, tracing into called procedures.
CTRL + BREAK	Suspends execution of an executing script and places the instruction pointer on the next line to be executed.

## Build and RTS Shortcuts

**Table 6 Build and RTS Shortcuts**

Key Name(s):	Description:
F5	Runs the selected component instances
SHIFT + F5	Build/Run
CTRL + SHIFT + F5	Restart
F7	Build
F10	Step

## Specification Code Editor Shortcuts

**Table 7 Specification Code Editor Shortcuts**

Key Name(s):	Description:
CTRL + A	Select all
CTRL + C	Copy
CTRL + E	Clear
CTRL + F	Find

Key Name(s):	Description:
F3	Find again
CTRL + H	Launch external editor
CTRL + I	Import
CTRL + L	Select line
CTRL + P	Print
CTRL + R	Replace
CTRL + T	Font
F4	Replace again
CTRL + V	Paste
CTRL + W	Select word
CTRL + X	Cut
CTRL + Z	Undo

## Browser Shortcuts

**Table 8 Browser Shortcuts**

Key Name(s):	Description:
CTRL + B	Browse specifications
CTRL + D	Delete from model
CTRL + SHIFT + G	Get latest
CTRL + SHIFT + I	Check in
CTRL + SHIFT + O	Check out
CTRL + SHIFT + U	Undo checkout

## Rational Rose RealTime Keyboard Shortcut Summary

---

Print the following page to have a convenient hardcopy version of the keyboard shortcuts for Rational Rose RealTime.

# Rational Rose RealTime Keyboard Shortcuts

## General Desktop Navigation

ALT or META + key	Display the contents of a menu - in combination with the underlined letter in the menu name.	SHIFT + F7
ALT + ALT or Shortcut Menu Key	Displays the context menu for the selected element. <b>Note:</b> On Unix, by default clicking ALT activates a context menu. Pressing it once changes the focus to the application's main menu. Unix keyboards have two additional keys, a left and right diamond, which are similar to the ALT key. The left diamond works the same as the ALT key by making the main menu active. If you use the right diamond, press it three times to change the focus to the main menu.	SHIFT + F8

CTRL+TAB	Move between dialogs, or tabs on Specification dialogs.	SHIFT + F9
CTRL + Q	Hide or restore the browser, Output window, Documentation tab, and Specification dialogs.	SHIFT + F11
ENTER	Perform the action in a dialog	F8
TAB	Move forward between areas of a dialog.	F12
SHIFT + F10	Displays the shortcut menu. <b>Note:</b> On UNIX, you must press SHIFT + F10 twice.	
SHIFT + TAB	Move backward between areas of a dialog.	
SPACE BAR	Select an item in a dialog	

## General Toolset Shortcuts

ALT + ENTER	Activates Hot Link on Specification dialog.	
ALT + PgUp	Opens previous specification in Specification History list.	
SHIFT + ALT + PgUp	Opens previous specification without closing the current Specification dialog.	
ALT + PgDn	Opens next specification in the Specification History list.	
SHIFT + ALT + PgDn	Opens next specification without closing the current Specification dialog.	
CTRL + +	Go Inside	
CTRL + -	Go Outside	
CTRL + A	Select All	
CTRL + B	Browse specification	
CTRL + C	Copy	
CTRL + E	Expand	
CTRL + F	Find - displays the <b>Find</b> dialog	
CTRL + SHIFT + F	Replace	
CTRL + I	Zoom in	
CTRL + SHIFT + R	Relocate	
CTRL + L	Change line attribute	
CTRL + M	Zoom to selected	
CTRL + N	Opens a new window for editing, or opens the <b>Create New Model</b> dialog.	

CTRL + O	Open	
CTRL + P	Print	
CTRL + SHIFT + P	Edit the Path Map	
CTRL + R	Browse referenced items	
CTRL + S	Save	
CTRL + T	Browse state diagram	
CTRL + SHIFT + T	Browse structure diagram	
CTRL + U	Zoom out	
CTRL + V	Paste	
CTRL + W	Fit to window	
CTRL + X	Cut	
CTRL + Y	Redo	
CTRL + Z	Undo	
CTRL + 3	Show/Hide the Documentation/Code window.	
CTRL + 4	Show/Hide the Output window.	
CTRL + 5	Show/Hide the Specification History window.	
CTRL + F6	Browse next pane	

CTRL + SHIFT + F6	Browse previous pane	
CTRL + F10	Browse component diagram	
DEL	Delete	
SHIFT + F1	Context sensitive help	
F2	Refresh	
F3	Browse previous diagram	
F4	Browse parent	
ESC	Close an open menu or cancels a dialog.	
F1	Context-sensitive help	
SHIFT + F6	Browse class diagram.	

## Debugging Shortcuts

## Editing Shortcuts

## Build and RTS Shortcuts

## Specification Code Editor Shortcuts

# Index

## Symbols

`#{name}_construct` 196

## A

A Workspace 131

About Rose RealTime dialog 39

Abstract 142

accessing source control operations 408

actions (Activity Diagrams) 269

Actions tab 236

Active Component 450

active component 450

Active Component Instances list 450

active component, assigning an 440

activities

Activity Diagrams 264

creating nested 265

history 99, 267

nested 265

specifying actions for 264

types 264

Activity Diagrams 257

Action 95

Action Expression 95, 268

action types 98, 271, 281

Actions 269

Actions Specification dialog 270

Activities 264

Activity Diagram Specification 261

activity history 99, 267, 278

Activity Specification dialog 268

changing assignment responsibility of  
swimlanes 289

creating 260

creating nested activities 265

creating swimlanes 287

Decision Specification dialog 272

Decisions 271

definition 258

deleting swimlanes 287

displaying multiple views of swimlanes 288

End State 274

event 101, 268

example 258

history 99, 267

manipulating nested activities 265

modeling using 258

moving swimlanes 288

nested activities 265

nested statesstates

nested 276

Object Flow 290

Object Flows and Transitions 292

object state 291

Objects 290

Send Event 95

specifying actions for activities 264

Start State 274

state history 99, 267, 278

State Specification dialog 277

StateMachine Specification 262

states 275

Stereotype 273, 277

sub activity 278

sub state 278

sub state history 99, 267

subs activity history 99, 267

Swimlane Specification dialog 289

Swimlanes 286

Synchronization Specification dialog 282

synchronization stereotype 282

synchronizations 281

tools 263

Transition between substates 286

Transition Guard Condition 285

Transition Specification dialog 284

- Transition stereotype 284
- Transitions 283
- Trigger Specification 280
- actor
  - creating 143
  - specification 144
- Actor specification 144
- actor, creating an 143
- Add Capsule command 66
- Add Class Dependencies wizard 168
- Add Classes command 66
- Add commands 66
- Add Components command 67
- Add Interfaces command 67
- Add Protocols command 67
- Add Use Cases command 67
- Add Watch command 73
- AddCodeImportProperties 394
- Add-In Manager 75
- Add-In Manager dialog 561
- adding
  - capsule role 219
  - choice point 242
  - class dependencies 63, 449
  - code to model elements 112
  - color to an FOC 320
  - documentation to model elements 111
  - files to source control 406
  - FOC 309
  - Focus of Control 309
  - icons to a diagram 83
  - instance to sequence diagram 301
  - states 242
  - stereotypes to Diagram Toolbox 548
  - tags to Code 393
- Adding a capsule role 219
- Adding a choice point 242
- Adding a state 242
- Adding and hiding classes, and filtering class relationships 171
- adding classes
  - class
    - adding 178
- Adding code to model elements 112
- Adding documentation to model elements 111
- Adding Icons to a Diagram 83
- Adding instances 301
- Adding stereotypes to the diagram toolbox 548
- Add-ins 561
- Add-ins menu 75
- Aggregating and decomposing state machines 235
- aggregating state machines 235
- aggregation 159
  - creating relationships 160
- aggregation relationships, creating 160
- Aggregation Specification 162
- Aggregation tool 151
- analysis and design 26
- Animation 488
- Application window 37
  - browsers 37
  - diagrams 38
  - menu bar 38
  - Toolbar 38
  - toolboxes 38
- Application-specific command line arguments 510
- Apply Label operation 413
- Assigning an active component 440
- assigning an active component 440
- association
  - changing direction 170
  - End A defined 155
  - End B defined 155
  - properties 155
- association class 152
  - creating 162
- association class, creating an 162
- Association Properties 155
- association relationships, creating 154
- association role
  - association 227
  - multiplicity 227
  - stereotype 227
- Association Role Specification 226
- Association Role tool 225
- Association specification 155
- Association tool 151
- association, changing the direction of an 170

- associations
  - creating relationships 154
- asynchronous send message tool 309
- attach console 485
- attach target 485
- Attach Target option 483
- Attaching a Model Property Set to a Single Element or a Collection of Elements 563
- Attribute Specification 345
- attributes
  - changeability 347
  - copying 97
  - creating 97, 334
  - derived 347
  - initial value 347
  - moving 97
  - naming 430
  - specification 345
  - types 347
  - visibility 346
- Attributes tab 96
- attributes, creating new 334
- automatically saving build results 469
- AutoSave 469

## B

- Background Popup menu 84
- basic mode
  - Windows CE 518
- basic\_string template 202
- Break 198
- breakpoints
  - diagram 503
  - setting 500
  - setting for operations 507
  - setting on state machine without TO 501
  - state machines 501
- Browse
  - Find References 94
- Browse Button 335
- Browse menu 57
- Browse menu operations 58
- Browser Shortcuts 571
- browser shortcuts 571, 572
- Browser tab 559
- Browser, displaying the 81
- Browsers 37, 78
  - RTS 485
- browsers 37
- browsers, multiple 81
- Build and RTS Shortcuts 570
- Build basics 439
- build errors 114, 463
  - Capsule Role name same as Capsule name 464
  - Check Environment Variables 465
  - Compile Fails on Valid C++ Models with VC++ 5.0 or VC++ 6.0 467
  - Compiler not installed correctly 464
  - Ensure that Component has correct Make types configured 465
  - Error Linking Capsule (error from nmake) 467
  - Linking 468
  - Linking wrong Services Library set 464
  - Missing Class Dependencies 464
  - Missing Header Files, Object Files, and Libraries 466
  - name conflicts 466
  - Redefinition of basic types or multiple declarations for X 463
  - Review your compiler flag settings 465
  - Source File Compilation 467
  - System does not understand the make command 465
  - understanding 463
  - Unknown command, command not found, the name specified is not recognized 463
  - unknown compiler message stream 451
  - Unresolved symbol or undeclared identifier 463
  - Windows NT Compilation Command Line Limits 467
- Build Errors tab 114, 451
  - filtering results 114
  - sorting results 114
- Build log 113

- build log 450
  - importing from 470
- Build Log tab 113, 450
  - saving output to a file 113
- Build menu 61, 447
- build output
  - saving 113
- build results
  - filtering 114
  - sorting 114
- build results, reviewing 444
- Build Settings dialog 450
- build shortcuts 570, 572
- build, starting a 441
- Building 439
- building
  - add class dependencies 449
  - assigning an active component 440
  - automatically saving build results 469
  - basics 439
  - Component wizard 450
  - errors 451
  - importing model compilation results 468
  - Load command 449
  - log 450
  - models 437
  - rebuild 448
  - Reload command 449
  - Restart command 449
  - reviewing results 444
  - run 448
  - saving output 468
  - settings 450
  - Shutdown command 449
  - Start command 448
  - starting 441
  - Stop command 448
  - top-level capsule 440
- Building and running models 437

## C

- call message 309
- Calls command 74

- capsule
  - attributes
    - attributes
      - capsule 357
  - definition of 20
  - moving multiple 372
  - naming 430
  - operations 356
  - ports 359
  - stereotype 355
  - use 20
- capsule class
  - creating 353
- Capsule class, creating a 353
- capsule connectors 359
- capsule diagram
  - undocking 354
- Capsule diagrams 354
- capsule diagrams, undocking 354
- capsule instance
  - dragging into a trace 495
- Capsule instance folder 487
- capsule instance folder 487
- Capsule Instance trace 491
- Capsule instance trace 494
- capsule instance trace 494
- capsule instances, dragging into a trace 495
- capsule role
  - adding 219
  - cardinality 220
  - class 219
  - fixed 220
  - name 219
  - plug-in 220
  - specification 219
  - substitutable 220
- Capsule Role Specification 219
- Capsule Role tool 212
- capsule role tool 225
- capsule role, adding a 219
- capsule roles 358
- capsule roles, connecting ports together 221
- Capsule specification 354
- Capsule Specification—Attributes tab 357
- Capsule Specification—Capsule Roles tab 358



- Capsule Specification—Components tab 360
- Capsule Specification—Connectors tab 359
- Capsule Specification—Details tab 356
- Capsule Specification—Files tab 360
- Capsule Specification—General tab 355
- Capsule Specification—Operations tab 356
- Capsule Specification—Ports tab 359
- Capsule Specification—Relations tab 360
- capsule state diagram
  - creating 229
- capsule state machines, creating 229
- capsule structure 207
- capsule structure diagrams 21
- capsule structure, creating 207
- Capsules 20
- capsules
  - components 360
  - relations 360
- Capsules, protocols, ports, capsule state and structure diagrams 19
- cardinality 221
  - capsule role 220
  - port 214
  - port role 218
- Cascade command 76
- change management 27
- Change View Spread
  - Constant Radial 69
  - Decreasing Radial 69
  - Increasing Radial 69
  - Uniform 69
- Change View Spread command 69
- changing association direction 170
- Changing the direction of an association 170
- Check environment variables 465
- Check in operation 412
- Check Model command 70
- check out
  - unreserved 416
- Check out operations 411
- checking out files
  - when edited 405
  - with secondary edits 406
- checkout
  - unreserved 416
- CHello\_construct 196
- choice point
  - adding 242
  - conditions 237
  - specification 237
- Choice Point Specification 237
- Choice Point tool 234
- choice point, adding a 242
- class
  - attributes 341
  - capsule role 219
  - creating 333
  - creating stereotypes 548
  - hiding 178
  - multiplicity 337
  - naming 430
  - nested 342
  - persistence 338
- class dependencies, missing 464
- Class diagram editor, using the 146
- Class diagram toolbox 149
- Class diagram, creating a 145
- class operations
  - implementation 340
  - private 340
  - protected 340
  - public 340
  - show inherited 340
  - stereotype 340
  - visibility 340
- Class Specification 335
- class specification
  - abstract 339
  - concurrency 338
  - formal arguments 339
  - implementation 337
  - language 337
  - private 337
  - protected 337
  - public 337
  - stereotype 336
  - type 336
  - visibility 337
- Class Specification content 335
- Class Specification—Attributes tab 341

- Class Specification—Components tab 344
- Class Specification—Detail tab 337
- Class Specification—Details tab 345
- Class Specification—Files tab 345
- Class Specification—General tab 336
- Class Specification—Nested tab 342
- Class Specification—Operations tab 339
- Class Specification—Relations tab 344
- class, creating a 333
- classes
  - generating component libraries 180
  - generating instantiated 200
  - generating parameterized 200
  - instantiated 203
  - moving multiple 372
  - relationships 203
- classes or diagrams, impact of moving on configuration management 375
- classes, adding and hiding and filtering relationships 171
- classesparameterized 201
- classifier role
  - classifier 226
  - specification 225
  - stereotype 226
- Classifier Role Specification 225
- Classifier Role tool 225
- Cloning a Sequence diagram 302
- cloning a sequence diagram 302
- code editor shortcuts 570
- Code generation 223
- code generation 223
  - state machine 178
- code import process 386
  - analyzing the Code 391
  - importing code 394
  - launching the C++ Analyzer 387
  - preparing the Rose model 386
  - selecting a source file location 388
  - specifying Export Options 388
  - using CodeCycle to Add Tags to Code 393
- Code pane 110
- Code Sync 187
  - considerations 189
  - disable 188
  - enable 188
- code sync
  - designated areas 534
  - disabling 533
  - enabling 533
  - identification tags 533
  - limitations 532
  - overview 531
  - using 532
- Code window 110
- code, adding to model elements 112
- CodeCycle 393
- CodeSyncEnabled 189
- collaboration diagram
  - creating 222
  - editor 222
  - toolbox 224
- Collaboration diagram editor, using the 222
- Collaboration diagram toolbox 224
- collaboration diagram, creating a 222
- Collaboration diagrams, opening 306
- collaboration relationships 223
- collaborations and sequences, relationship between 223
- Coloring Focus of Control 320
- Column check boxes 496
- command line
  - application-specific arguments 510
  - arguments 509
  - using 509
- Command line arguments 509
- command line arguments, application-specific 510
- Command Line Debugger 483
- command line parameter 508
- common build errors, overview 463
- Compartments tab 558
- Compilation 18
- Compilation Command Line Limits, Windows NT 467

- compilation results 468
  - importing 468
  - saving 468
- Compile a simple Hello World program 465
- Compile fails on valid C++ models with VC++ 5.0
  - or VC++ 6.0 467
- compiling
  - code externally 535
- Component
  - step 62
- component
  - assigning for building 440
  - dependencies 461
  - load 63
  - rebuild 61
  - reload 63
  - run 62
  - settings 63
  - shutdown 63
  - start execution 62
  - stereotype 452
  - stop execution 62
- Component Dependencies 461
- component diagram
  - dependency 378
  - editor 377
  - toolbox 379
- Component diagram editor, using the 377
- Component diagram toolbox 379
- component instance
  - attach console 485
  - attach target 485
  - creating 476
  - detach target 485
  - error call stack length 515
  - load 484
  - operation mode 512
  - parameters 512
  - reload 484
  - restart 484
  - run 484
  - run with Purify 484
  - running with Purify 477
  - running without Purify 479
  - runtime exception 497
  - shutdown 484
  - unload 484
  - utility scripts 511
- Component instance menu 484, 509
- component instance menu 484, 509
- component instance options
  - basic 513
  - Debugger MSDEV 513
  - Debugger Tornado 513
  - Debugger xgdb 513
  - Manual 514
  - Windows CE 514
- Component Instance specification 511
- Component Instance Specification—Detail
  - tab 512, 515
- Component Instance Specification—General
  - tab 511
- component instance, creating a 476
- component instance, observing a running 481
- component instance, running a 479
- component instance, run-time exception while
  - running a 497
- component instances, loading and running on
  - embedded targets 510
- Component Specification 451
- Component Specification—Files tab 453
- Component Specification—General tab 452
- Component Specification—References tab 452
- Component Specification—Relations tab 453
- component, creating a 441
- Components tab 97
- Concurrency 291, 338
- concurrency
  - active 338
  - class 338
  - guarded 338
  - sequential 338
  - synchronous 338
- Condition tab 237
- configuration 27
- configuration management
  - impact on moving classes or diagrams 375
- Configure command 72
- ConfigureFromRoseProperties 396

- configuring
  - Tornado 2 for debugger mode 522, 524
  - Tornado for debugger mode 522, 524
- conflicts when demoting 165
- conflicts when promoting 165
- conjugated
  - port 215
  - port role 218
- connecting
  - ports on capsule roles 221
- Connecting ports on capsule roles together 221
- connection
  - characteristics 527
- connection delay 515
- Connector Specification 221
- connector specification 221
  - cardinality 221
  - delay 221
- Connector tool 212
- connectors
  - capsules 359
- Constraint tool 92, 150, 211, 224, 233, 309
- ConstructFunctionName 196
- Constructing Models in Rational Rose
  - RealTime 22
- constructors 195
- contacting Rational customer support xxx
- Containment View tab 80
- continuation junction point 238
- Continue 198
- Controlling how existing stereotypes are displayed in a diagram 549
- Controlling how stereotypes are displayed in the browser 549
- Controlling how stereotypes that are added to diagrams hereafter are displayed 550
- Controlling the display of stereotypes 549
- convert a component 397
- Convert Rose Component Wizard 397
- copying
  - operations 100
- copying attributes 97
- copying signals 363
- copying triggers 236
- co-region tool 310
- Create 30
- Create New Model 39, 45
- Create New Model dialog 30
- Creating
  - ports 212
- creating
  - Activity Diagrams 260
  - actors 143
  - association class 162
  - association relationships 154
  - attributes 334
  - capsule and protocol aggregations 148
  - capsule class 353
  - capsule state diagram 229
  - capsule structure 207
  - classes 333
  - collaboration diagram 222
  - component instance 476
  - component instance tasks 476
  - custom framework (stereotypes) 544
  - dependency relationships 167
  - inheritance relationships 162
  - inheritance tree 163
  - inject messages 528
  - model property set 564
  - nested activities 265
  - nested states 248
  - new attributes 97
  - new operations 340
  - non-wired port using a system protocol 213
  - Object Flow 293
  - operations 100
    - operations
      - creating 334
  - package relationships 170
  - packages 367
  - realize relationships
    - realize relationships
      - creating 171
  - reflexive relationships 170
  - relationships 153
  - scratch pad packages 103
  - sequence diagram 299

- sequence diagram from a trace 492
  - sequence diagram from message trace traces
    - creating sequence diagram from message trace 495
  - stereotype (new) 544
  - stereotype configuration file 545
  - stereotype for all Rose RealTime models 545
  - stereotype icons 548
  - stereotypes 543
  - stereotypes for Classes 548
  - swimlanes (Activity Diagrams) 287
  - use case 141
  - use case diagram 137
  - Creating a Capsule class 353
  - Creating a class 333
  - Creating a Class diagram 145
  - Creating a collaboration diagram 222
  - Creating a component 441
  - Creating a component instance 476
  - Creating a custom framework 543
  - Creating a diagram icon 549
  - Creating a link 542
  - Creating a new diagram 299
  - Creating a New Model Property Set 564
  - Creating a new stereotype configuration file 545
  - Creating a new stereotype for all Rose RealTime models 545
  - Creating a new stereotype for the current model 544
  - Creating a non-wired port using one of the system protocols 213
  - Creating a package 367
  - Creating a port 212
  - Creating a Sequence diagram 299
  - Creating a sequence diagram from a message trace 495
  - Creating a use case 141
  - Creating a use case diagram 137
  - Creating aggregation relationships 160
  - creating aggregation relationships 160
  - Creating an actor 143
  - Creating an association class 162
  - Creating an inheritance tree 163
  - Creating association relationships 154
  - Creating capsule state machines 229
  - Creating capsule structure 207
  - Creating dependency relationships 167
  - Creating inheritance relationships 162
  - Creating nested states 248
  - Creating new attributes 334
  - Creating new operations 334
  - Creating package relationships 170
  - Creating reflexive relationships 170
  - Creating relationships 153
  - creating sequence diagram
    - from browser 300
    - from collaboration diagram 300
    - from structure diagram 300
    - from structure diagram browser 300
  - Creating sequence diagrams
    - from the browser 300
    - from the collaboration or structure diagram 300
    - from the Structure diagram browser 300
  - Creating stereotype icons 548
  - Creating stereotypes for classes 548
  - cross-references 403
  - Customize Toolbar dialog 561
  - Customizing the diagram toolbox 561
- ## D
- data classes
    - state machine 178
  - debugger
    - unloading 525
  - debugger mode
    - Tornado 522, 524
    - Tornado 2 522, 524
    - Windows CE 522, 523
    - xxgdb (Unix only) 524
  - debugger modes 521
  - debugging
    - breakpoints 500
    - importing model compilation results 468
    - setting breakpoints 500
    - source code 498
  - Debugging Shortcuts 569

- decisions (Activity Diagrams) 271
- decomposing state machines 235
- Defining messages 302
- defining messages in a sequence diagram 302
- Defining state transition trigger events 246
- delay for connector specification 221
- deleting
  - messages from trace 491
  - swimlanes 287
- Deleting a Model Property Set 564
- Deleting messages 491
- demote
  - conflicts 165
- demoting elements 165
- dependencies
  - component 461
- dependency relationships
  - creating 167
- dependency relationships, creating 167
- Dependency Specification 168
- deployment diagram
  - components 382
  - connections 381
  - devices 381
  - editor 380
  - elements 381
  - packages 382
  - processors 381
  - toolbox 382
- Deployment diagram editor, using the 380
- Deployment diagram elements 381
- Deployment diagram toolbox 382
- Description window 109
- Description window, displaying the 109
- designated code sync areas 534
- destroy message tool 310
- detach target 485
- Development process 25
- development process 25
- Device specification 517
- device specification
  - characteristics 526
- Diagram editors 82
- diagram icon, creating a 549
- Diagram tab 143, 555
- diagram toolbox, customizing the 561
- diagram types
  - Activity 257
  - class 58
  - collaboration 58
  - component 58
  - deployment 58
  - sequence 58
  - state 58
  - structure 58
  - use case 58
- diagram, inserting into an MS Word
  - document 541
- Diagrams 24, 38
  - Activity 257
  - Capsule 354
  - State 354
- diagrams 38
- Diagrams tab 120
- Dialog
  - Activity Specification 268
- Dialogs
  - Build Settings 450
  - Customize Toolbar 561
  - Event Editor 239
  - Find 105
  - Generate Documentation 540
  - Options 550
  - Replace 106
  - Select Diagram 57
  - Sequence Validation 318
- disable code sync 188
- disabling
  - code sync 533
- display a nested class 343
- Displaying or Editing a Specific Model Property
  - Set 564
- Displaying or Modifying the Values of Model
  - Properties 562
- Displaying the Browser 81
- Displaying the Calls dialog 74
- Displaying the Description window 109
- docking 81, 110
- documentation
  - linking 539

- Documentation pane 110
- Documentation Report command 66
- Documentation window 110
- documenting
  - model elements 111
- Dragging capsule instances into a trace 495
- Drawing the initial transition 245
- Drawing transitions between states 242

## E

- Edit menu 49
- editing
  - checked out files 406
- Editing a diagram 300
- Editing diagrams 542
- Editor tab 559
- elements 165
  - demoting 165
  - deployment diagram 381
  - exporting 399
  - moving 75
- elements, required 22
- embedded targets 510
- enable code sync 188
- enable source control 405
- enabling
  - code sync 533
- End A and B Detail tabs 158
- End A and B General tabs 157
- End A defined 155
- End B defined 155
- end port 215
- end state
  - Activity Diagrams 274
- Ensure that component has correct make types
  - configured 465
- Entry Actions / Exit Actions 234
- entry actions (state diagram) 234
- Environment 180
- environment settings 132
- error
  - call stack length 515
- Error linking Capsule (error from nmake) 467
- error log
  - validation 319
- errors 463
  - Cannot code-sync 537
  - Cannot code-sync filename beyond line
    - lineNum 537
  - Capsule Role name same as Capsule
    - name 464
  - Check Environment Variables 465
  - Compile Fails on Valid C++ Models with
    - VC++ 5.0 or VC++ 6.0 467
  - Compiler not installed correctly 464
  - Could not find trailing CodeSync tag for 537
  - Ensure that Component has correct Make
    - types configured 465
  - Error Linking Capsule (error from
    - nmake) 467
  - Linking 468
  - Linking wrong Services Library set 464
  - Missing Class Dependencies 464
  - Missing Header Files, Object Files, and
    - Libraries 466
  - name conflicts 466
  - Redefinition of basic types or multiple decla-
    - rations for X 463
  - Review your compiler flag settings 465
  - Source File Compilation 467
  - System does not understand the make
    - command 465
  - Unknown command, command not found,
    - the name specified is not
    - recognized 463
  - Unresolved symbol or undeclared
    - identifier 463
  - Warning
    - Use tabs for indenting code-sync
      - regions 537
  - Windows NT Compilation Command Line
    - Limits 467
- Essential workflows 26
- Event Editor dialog 239
- EventGuard 240
- Exclusions 163
- Executable models 21
- executable models 21

- Executing 439
- Execution basics 476
- execution control 486
- Execution control and information pane 486
- execution overview 476
- execution watch 496
- Execution Watch tab 496
- exit actions (state diagram) 234
- Expand Selected Elements command 67
- Export command 71
- export control 169
- export options 388
- exporting
  - elements 399
  - file 399
  - files 399
- Exporting a file 399
- external library 396
- externally visible junction point 238

## F

- file history 414
- file I/O error on
  - 537
- File menu 45
- File menu operations 45
- file, importing a 385
- files
  - importing 385
- Files tab 98
- Filter Relationships command 68
- filtering 81
  - Build Results 114
- filtering class relationships
  - class relationships
    - filtering 178
- Filtering tab 558
- final state 178
- Final State tool 233
- Find
  - references 94
- Find dialog 105
- Find References 94, 335, 361

- Find tab 115
- finding
  - procedure calls 74
- Finding Specified Text 74
- fixed capsule role 220
- fixing a model 134
- floating 81, 110
- FOC 309
  - activators 319
  - adding color 320
  - definition 319
- Focus of Control 319
  - adding 309
  - definition 319
- Focus of Control, coloring 320
- Font/Color tab 554
- friend 160
- functions
  - main() 19
  - virtual 198
- Further reading 23, 25, 26, 27

## G

- General Shortcuts 565
- generalization 152
- Generalize Specification 163
- Generate 472
- Generate Documentation dialog 540
- Generate local state information 493
- GenerateDefaultConstructor 197
- GenerateStateMachine 186
- generating
  - Component Libraries for Classes 180
  - documentation 539
  - state machine code 185
- generating documentation
  - inserting diagram into MS Word
    - document 541
- OLE 542
- report options 540
- Get Entire Model command 72
- Get operation 411
- global packages 370



- Go To Line command 74
- Graphical notation 168
- guard code 241
- Guarded 338
- guarded operation 351

## H

- Help menu 77
- Hide Selected Elements command 68
- hiding classes 178
- history 199
  - activity 99, 267, 278
  - state 99, 267, 278
  - sub activity 99, 267
  - sub state 99, 267

## I

- icons, adding to a diagram 83
- identification tags for code sync 533
- IDH\_EVENT\_EDITOR\_DIALOG 239
- Impact of moving classes or diagrams on configuration management 375
- Implementation 349
- implementation of workflow 27
- Import Code command 71
- import log messages 134
- importing
  - Classic C++ code from Rose 386
  - code 394
  - compilation results 471
  - elements 385
  - files 385
  - from Build Log 470
  - model compilation results 468
  - process 386
  - Rational Rose Generated Code 136
  - Rose generated code (Limitations and Restrictions) 136
- Importing a file 385
- importing model compilation 468
- Importing Rational Rose generated code 136

- importing requirements (ObjectTime Developer) 133
- in signal 362
- Inheritance
  - Rose RealTime 164
- inheritance
  - demoting 165
  - excluding elements 165
  - promoting 165
  - rearranging hierarchies 166
  - re-inheriting excluded elements 166
  - virtual 164
- Inheritance in Rose RealTime 164
- inheritance relationship
  - creating 162
- inheritance relationships, creating 162
- Inheritance tab 80
- inheritance tree 163
- inheritance tree, creating an 163
- initial state
  - specification 237
- Initial State Specification 237
- initial transition
  - drawing 245
- initial transition, drawing the 245
- inject
  - creating messages 528
- inject data format
  - basic types 529
  - classes 529
- inject messages
  - creating 528
- Inject window 494
- inject window 494
- injected data format 528
- injecting a message 530
- Injecting Messages 483
- inserting
  - diagram into MS Word document 541
- Inserting a diagram into an MS Word document 541
- Inserting a link 542
- Instance browser 498
- instance browser 498
- instances, adding 301

- instantiated class 153
- instantiated class utility 153, 336
- instantiated classes 203
- interaction
  - specification 311
  - stereotype 311
- interaction instance
  - path 310
  - specification 310
  - stereotype 311
- Interaction Instance Specification 310
- interaction instance tool 309
- Interaction Specification 311
- interface scripts 407
- internal transition 235
- Introduction to Naming Guidelines 429
- Introduction to packages 367
- Is Rose RealTime a compiler? 438

## J

- Joining transitions 247
- joining transitions 247
- junction point
  - continuation 238
  - externally visible 238
  - specification 238
- Junction Point Specification 238

## K

- keys 160

## L

- Language/Environment tab 560
- Languages and code generation 18
- Layout tab 122
- Limitations and restrictions of importing Rational Rose generated code 136
- Limitations and restrictions of opening models
  - from ObjecTime Developer 5.2.1 133

- Limitations and restrictions of opening models
  - from Rational Rose 135

- link element 156
- link, creating a 542
- link, inserting a 542
- Linking 468
- linking
  - external files to model elements 539
- Linking external files to model elements 539
- Load command 449
- load component instance 484
- loading
  - component instance on embedded targets 510
- Loading and running component instances on embedded targets 510
- local action
  - receiver 312
  - sender 312
  - stereotype 312
- Local Action Specification 312
- local state
  - receiver 313
  - sender 313
  - stereotype 313
- Local State Specification 312
- Lock Selection tool 92
- log reports
  - interpreting for Purify 479
- Log tab 112
- Logical View
  - moving model elements 75

## M

- main() function 19
- make 472
- make\_command 472
- Makefile pattern 188
- Making a Model Property Item Specific 563
- Managing model properties 562
- managing model properties 562
- Menu bar 38, 44
- menu bar 38

- Menus 43
  - add-ins 75
  - Browse 57
  - Build 61, 447
  - Component instance 484, 509
  - Edit 49
  - File 45
  - Help 77
  - Popup 111
  - Pull-down 111
  - Query 66
  - Report 64
  - Tools 68
  - View 56
  - Window 76
- menus
  - Background Popup 84
- message
  - receiver 314
  - sender 313
- message details, specifying 302
- Message Specification 313
- Message trace configuration dialog 496
- messages reorienting 306
- messages, defining 302
- messages, deleting 491
- messages, moving 307
- Missing class dependencies 464
- Missing header files, object files, and libraries 466
- mode
  - debugger 521
  - xxgdb for debugging 524
- Model
  - specification 129
- model
  - cross-references 403
  - opening 129
  - properties 562
  - setting to improve opening time 406
  - unique Id 125
  - unique ids 125
  - validation scenarios 404
- model elements
  - adding code 112
  - moving 75, 372
- Model Elements tab 371
- model elements, adding documentation to 111
- model management 468
- model properties
  - managing
    - model
      - managing properties 562
- Model Properties, displaying or modifying the values of 562
- model properties, managing 562
- Model Property Item, making one specific 563
- Model Property Set, creating a new 564
- Model Property Set, deleting 564
- Model Property Set, displaying or editing a specific one 564
- Model View tab 80
- modeling
  - elements 22
  - required elements 22
  - using Activity Diagrams 258
- Modeling elements 22
- models
  - building 437
  - creating stereotypes 545
  - executable 21
  - running 437
- models, building and running 437
- models, constructing in Rose RealTime 22
- Modify Variable command 73
- Monitors 488
- monitors
  - animation 488
- monitors, opening 489
- Move Model Elements 75
- moving attributes 97
- moving classes
  - impact on configuration management 375
- moving diagrams
  - impact on configuration management 375
- Moving messages 307
- moving model elements 372
- moving operations 100

- Moving the Insertion Point to a Specified Line in Your Script 74
- moving triggers 236
- Multiple Browsers 81
- multiplicity 158
  - association role 227
  - class 337
- multiplicity from 169
- multiplicity to 169

## N

- Name 429
- Name conflicts 466
- name direction 157
- names
  - assigning 429
  - guidelines 429
  - special case notes 430
- Naming 168
- naming
  - considerations 430
- naming guidelines 429
- naming guidelines, introduction to 429
- navigable 160
- Navigating 80, 542
- navigating 80
- nested
  - states 276
- nested activities 265
- nested class 342
  - deleting 343
  - displaying 343
  - relocating 344
- nested states
  - creating 248
- nested states, creating 248
- new diagram, creating a 299
- New script command 73
- nmake 472
- nologo 472
- non-wired port, creating one using one of the system protocols 213

- Note anchor tool 92, 212, 225, 233
- Note tool 211, 224, 308
- notification
  - port 216

## O

- object flow 290, 291
- object state 291
- Object State changes 292
- ObjecTime Developer 5.2.1, opening models from 132
- Objects 290
- objects (Activity Diagrams) 290
- observability 508
- Observability command line parameter 508
- observability command line parameter 508
- Observability interface 483
- observability interface 483
- observability options 483
  - attach target observability on startup 514
  - delay 514
  - load 514
  - order 514
  - run 514
  - target observability port 514
- observability options, overview of 483
- Observing a running component instance 481
- OLE
  - creating a link 542
  - inserting a link 542
  - using 542
- OLE, using 542
- Online Help 1
- Open Script command 73
- Opening
  - ObjecTime Developer model (limitations and restrictions) 133
  - Rose models (Limitations and Restrictions) 135
- opening
  - models 129
  - models from ObjecTime Developer 132
  - models from Rational Rose 134

- resolving model errors 134
  - Sequence Diagram 492
- Opening a Sequence diagram 223
- Opening Collaboration diagrams 306
- Opening models from ObjecTime Developer
  - 5.2.1 132
- Opening models from Rational Rose 134
- Opening Sequence Diagram 492
- Opening Specifications 84
- operation mode (component instance) 512
- Operation Specification 347
- operations
  - Apply Label 413
  - capsule 356
  - Check in 412
  - check out 411
  - class 349
  - concurrency 351
  - copying 100, 341
  - creating 100
  - exceptions 351
  - Get 411
  - instance 349
  - moving 100, 341
  - naming 430
  - options 349
  - parameters 350
  - protocol 350
  - Refresh status 411
  - return type 349
  - scope 349
  - Show Differences 413
  - Show History 413
  - specification 347
  - Submit all Changes 413
  - Synchronize 411
  - Uncheckout 412
  - visibility 349
- Operations tab 99
- operations, creating new 334
- Options dialog 550
- Orientation field 123
- out signal 362
- output
  - redirecting 472

- Output window 112
- Overriding target control 482
- overriding target control 482
- Overview Navigator 86
- Overview of common build errors 463
- Overview of observability options 483

## P

- Package 142
- package
  - components 371
  - creating relationships 170
  - global 370
  - moving model elements 372
  - relations 370
  - relationships 170
  - specification 368
- package relationships, creating 170
- Package Specification 368
- Package Specification dialog box 371
- Package Specification—Components tab 371
- Package Specification—Detail tab 370
- Package Specification—Files tab 371
- Package Specification—General tab 369
- Package Specification—Relations tab 370
- package, creating a 367
- packages
  - creating 367
  - creating scratch pad 103
  - overview 367
- Packages and class diagrams 368
- packages, introduction to 367
- Paper field 123
- Parameterized Class tool 151
- parameterized classes 201
- parameterized utility class 336
- parent diagram 59
- Parts menu
  - Menus
    - Parts 54
- Persistence 291, 338
- persistence
  - class specification 338

- Polymorphic 200
- polymorphic operation 349
- Popup menu 111
- port
  - cardinality 214
  - conjugated 215
  - creating 212
  - definition of 21
  - end port 215
  - name 213
  - notification 216
  - protected 216
  - protocol 214
  - publish 217
  - receiver 319
  - registration 217
  - sender 319
  - specification 213
  - stereotype 214
  - unwired 213
  - use 21
  - wired 215
- Port Detail 314
- port probe 493
- port role
  - cardinality 218
  - conjugated 218
  - specification 218
- Port specification 213
- Port tool 212
- Port trace 491
- port, creating a 212
- Ports 21
- ports
  - capsule 359
  - connecting to capsule roles 221
- primary edits 403
- print range 120
- Print setup 123
- Print Specifications 119
- Printer field 123
- printing
  - range 120
  - setup 123
  - specifications 119
- private
  - class specification 337
- probe
  - placing on replicated ports 494
  - types 493
  - usage 493
- Probe Break Points 483
- Probe Specification 527
- Probe Specification—Detail tab 528
- Probe Specification—Files tab 528
- Probe Specification—General tab 527
- probe types
  - port 493
  - state 493
- Probes 489, 493
- Probes folder 487
- probes folder 487
- procedure calls, finding 74
- processor
  - address 517
  - component instances 517
  - CPU 516
  - load script 517
  - OS 516
  - server 517
- Processor specification 516
- Processor specification—General tab 516
- processors
  - deployment diagram 381
- project phases 26
- promote
  - conflicts 165
- promoting 165
- promoting elements 165
- protected
  - class specification 337
  - port 216
- protocol 362
  - port 214
  - specification 361
  - stereotype 362
- Protocol specification dialog 361
- Protocol Specification—Components tab 363
- Protocol Specification—Files tab 364
- Protocol Specification—General tab 362

- Protocol Specification—Relations tab 363
- Protocol Specification—Signals tab 362
- Protocols 20
- protocols
  - definition of 20
  - moving multiple 372
  - naming 430
  - use 20
- public class specification 337
- publish
  - port 217
- Pull-down menu 111
- Purify 477
  - log reports 479
  - running component instance without 479

## Q

- qualifiers 160
- Query menu 66
- query operation 349

## R

- Rank 142
- Rational customer support
  - contacting xxx
- Rational Rose generated code, importing 136
- Rational Rose, opening models from 134
- Real-Time services (Services Library) 438
- RealTime Services Library 438
- rebuild 448
- rebuild component 61
- receiver port 319
- redirect command 472
- redirecting output results 472
- redirection\_command 472
- referenced item 61
- referencing
  - External Library 396
- refinement 200
- reflexive relationships 170
- reflexive relationships, creating 170
- refresh execution 487
- Refresh Status of Model command 72
- Refresh status operations 411
- refreshing shared unit status 406
- Refreshing the Browser 81
- Refreshing the watch values 115, 497
- refreshing watch values 115, 497
- registration
  - port 217
- re-inheriting excluded elements 166
- Reinstalling the State and Value of the Last Committed Change 563
- Related Documentation 2
- Relations 143
- Relations tab 101, 143
- Relationship between collaborations and sequences 223
- relationships
  - between collaborations and Sequences 223
- relationships, creating 153
- reload 484
- Removing an Overriding Item Level Model Property 563
- Reorienting messages 306
- Replace command 71
- Replace dialog 106
- replicated ports
  - placing probes 494
- Report menu 64
  - Show Usage 64
- report options
  - generating documentation 540
- Required elements 22
- requirements 26
- restart 484
- restart execution 486
- results 468
- Return 198
- Review your compiler flag settings 465
- Reviewing the build results 444
- Rose RealTime dialog, About 39
- Rose RealTime execution interface 482
- rtBound 240
- RTcompile 472
- rtg\_init1 197
- RTS browser 485

- RTS shortcuts 570, 572
- RTS tab 80
- rtUnbound 240
- Run command 448
- run component 62
- run component instance 484
- running
  - component instance 481
  - component instance on embedded targets 510
  - component instance with Purify 477
  - component instance without Purify 479
  - models 437
  - outside the Toolset 508
  - Purify from outside the Toolset 508
  - with Purify 484
- Running a component instance 479
- running component instance 477
- Running from outside the toolset 508
- Run-time exception while running a component instance 497

## S

- saving
  - automatically 469
  - build output 468
  - build output to a log file
    - log
      - saving build output 113
  - compilation results 468
- Scratch Pad Packages 103
- scratch pad packages
  - creating 103
- Scratchpad 102
- scripting shortcuts 568
- scripts directory 407
- Scroll Bars 85
- Searching and sorting 104
- Searching code 106
- Searching for model elements by name 106, 107
- secondary edits 403
- Select Checked out Units in Browser
  - command 73

- Select Diagram dialog 57
- Selective searching 107
- Selector tool 91, 150
- Semantics tab 352
- send
  - asynchronous message tool 309
  - synchronous message tool 309
- Send arguments 98, 271, 281
- Send target 98, 271, 281
- sender port 319
- sequence diagram
  - add FOC 309
  - adding instances 301
  - asynchronous send message tool 309
  - call message 309
  - cloning 302
  - constraint tool 309
  - co-region 310
  - create message tool 309
  - creating 299
  - creating from browser 300
  - creating from collaboration diagram 300
  - creating from message trace 495
  - creating from structure diagram 300
  - creating from structure diagram browser 300
  - defining messages 302
  - destroy message tool 310
  - editing 300
  - Focus of Control 309
  - interaction instance tool 309
  - local action tool 310
  - local state tool 310
  - moving messages 307
  - Note tool 308
  - reorienting messages 306
  - specifying message details 302
  - synchronous send message tool 309
  - toolbox 307
  - using the editor 305
- Sequence diagram editor, using the 305
- Sequence diagram toolbox 307
- sequence diagram, creating a 299
- sequence diagram, creating one from a message 495
- Sequence Diagrams 483



- Sequence diagrams, opening 223
- Sequence Overlays 223
- sequence overlays 223
- Sequence relationships 223
- Sequence Validation dialog 318
- Sequential 338
- sequential operation 351
- Services Library 19, 438
- setting
  - breakpoints 500
- shortcuts 565
  - browser 571, 572
  - build 570, 572
  - code editor 570
  - debugging
    - debugging shortcuts 569
    - general 565
    - RTS 570, 572
    - scripting 568
  - shortcuts, debugging 569
- Show Access Violations command 64
- Show Code Occurrences command 65
- Show Differences operation 413
- Show History operation 413
- Show Inherited command 96
- Show Part Of Ancestors command 66
- Show Part Of Descendants command 66
- Show References command 65
- Show Unit Versions command 73
- Show Usage command 64
- shutdown 484
- shutdown execution 487
- Signal Specification 364
- Signal Specification—Files tab 365
- Signal Specification—General tab 365
- signals 362
  - copying 363
  - data class 365
  - protocol 362
- sorting
  - build results 114
- Sorting in the browser 104
- Sorting in the class specification 104
- Source Code Debugging 483
- Source code debugging 498
- source code debugging 498
- source control
  - accessing operations 408
  - adding files 406
  - Apply Label 413
  - Check in 412
  - Check out 411
  - checking out files
    - automatically 405
  - checking out files automatically 406
  - enabling 405
  - Get 411
  - interface scripts 407
  - location of interface scripts 407
  - operations 410
  - primary edits 403
  - Refresh status 411
  - scripts directory 407
  - secondary edits 403
  - settings 404
  - Show Differences 413
  - Show History 413
  - status options 402
  - Submit all Changes 413
  - supported systems 407
  - Synchronize 411
  - Uncheckout 412
    - unreserved check out 416
    - versionable elements 407
- Source Control command 72
- source control status 402
- Source File Compilation 467
- Special Case Notes 430
- Specification Code Editor Shortcuts 570, 572
- Specification Content 451
- Specification dialogs 92
- Specifications tab 121
- Specifications, opening 84
- specify event 239
- specifying history 199
- Specifying message details 302
- Specifying the transition 245
- Spreadsheet-type functionality for list controls
  - within a specification dialog 93
- Start command 448

- start execution 486
- Starting a build 441
- startup options 551
- Startup screen 29
- state
  - adding 242
  - drawing transitions between 242
  - end 274
  - history 99, 267
  - naming 430
  - start 274
- State Diagram 354
- state diagram
  - aggregating 235
  - choice point tool 234
  - constraint tool 233
  - decomposing 235
  - editor 230
  - elements 231
  - entry actions 234
  - exit actions 234
  - final state tool 233
  - Note anchor tool 233
  - Note tool 233
  - specification 234
  - state tool 233
  - state transition tool 233
  - toolbox 232
  - transition to self tool 234
  - using the navigator 232
- State diagram editor, using the 230
- State diagram toolbox
  - Toolboxes
    - State diagram 232
- State diagrams 21, 246
- state machine
  - data classes 178
- state machines, adding and decomposing 235
- state probe 493
- State Specification 234
- State tool 233
- State trace 491
- State Transition tool 233
- state transition trigger events, defining 246
- state, adding a 242
- states
  - creating nested states 248
- step through execution 486
- Stereotype 336
- stereotype 142, 156
  - Activity Diagrams 273, 277
  - association role 227
  - capsule 355
  - class operations 340
  - class specification 336
  - classifier role 226
  - component 452
  - configuration file 545
  - creating a new configuration file 545
  - creating icons 548
  - display 556
  - icons (creating) 548
  - interaction 311
  - interaction instance 311
  - local action 312
  - local state 313
  - message 313
  - port 214
  - protocol 362
- stereotype icons, creating 548
- stereotype, creating a new 545
- stereotype, creating a new one for all Rose Real-  
Time models 545
- stereotype, creating a new one for the current  
model 544
- Stereotypes 543
  - controlling display 549
- stereotypes
  - adding to Diagram Toolbox 548
  - creating 544
  - creating a custom framework for models 543
  - creating for all Rose RealTime models 545
  - creating for classes 548
  - details 543
- stereotypes for classes, creating 548
- stereotypes, controlling how existing ones are dis-  
played in diagrams 549
- stereotypes, controlling how those that are added  
to diagrams hereafter are  
displayed 550

- stereotypes, controlling the display of 549
- stereotypes, controlling their display in the
  - browser 549
- Stop command 448
- stop execution 486
- Structure Diagram
  - Diagrams
    - Structure 354
- Structure Diagram toolbox 211
- Structure diagram toolbox 211
- Structure Editor 208
- Structure editor, using the 208
- sub state 99, 267
- Submit all Changes operation 413
- Submit All Changes to Source Control
  - command 73
- substitutable capsule role 220
- supported source control systems 407
- swimlanes
  - changing assignment responsibility 289
  - creating 287
  - deleting 287
  - displaying multiple views 288
  - moving 288
- swimlanes (Activity Diagrams) 286
- synchronizations 281
- Synchronize Entire Model command 72
- Synchronize Model with File System
  - command 73
- synchronize operations 411
- Synchronous 338
- synchronous operation 351
- synchronous send message tool 309
- syntax
  - redirecting results 472

## T

- Tabs 80, 95
- tags
  - adding to code 393
- target
  - connection delay 515
  - default instrumentation type 515

- target control 507
  - overriding 482
- Target control programs 482
- target control programs 482, 517
- target display
  - first occurrence only 516
  - handles in use at exit 516
  - memory in use at exit 516
  - memory leaks at exit 516
- target scope 159
- targets
  - loading and running 510
- tasks
  - creating a component instance 476
  - debugging source code 499
  - observing a running component instance 481
  - running a component instance with
    - Purify 477
  - running component instance without
    - Purify 479
- TCKill 519
- Text tool 91
- The Script Editor Window 73, 74
- The toolbar 39
- Threshold field 496
- threshold field 496
- Tile Horizontally command 76
- Tile Vertically command 76
- To open a monitor 489
- Toolbar 38
- toolbar 39
  - create new model 39
  - open existing model 40
- Toolbar button list 561
- Toolbars tab 560
- Toolboxes 38, 91
  - Class diagram 149
  - Collaboration diagram 224
  - Component diagram 379
  - Deployment diagram 382
  - Sequence diagram 307
  - Structure diagram 211
  - Use case diagram 139

- tools
  - constraint 92
  - lock selection 92
  - note 92
  - note anchor 92
  - selector 91
  - text 91
  - zoom 91
- Tools menu 68
- Toolset dependencies 205
- Toolset options 550
- Top-level capsule 440
- top-level capsule
  - capsule
    - top-level 440
- Tornado
  - debugger mode for 522, 524
  - unloading debugger 522, 524
- Tornado 2
  - debugger mode for 522, 524
  - unloading debugger 522, 524
- trace
  - creating sequence diagram from 492
- Trace configuration 491
- trace configuration 491
- Trace window 490
  - types 491
- Trace windows 490
- Traces 483
- traces
  - capsule instance 491, 494
  - dragging capsule instances 495
  - event message 494
  - messages 491
  - port 491
  - state 491
  - types 491, 492
  - using different types 492
- traces, when to use the different kinds of 492
- transition
  - internal 235
  - specification 235
- Transition Specification 235
- Transition to Self tool 234
- transition trigger event
  - define a new event in a protocol 246
  - defining a new event in a capsule 246
  - defining a new event in a data class 246
  - defining a new state 246
- transition, specifying the 245
- transitions
  - Activity Diagrams 283
  - drawing an initial transition 245
  - drawing between states 242
  - joining 247
  - specifying 245
- transitions, drawing between states 242
- transitions, joining 247
- trigger 178, 185, 239
- trigger Stereotype 183
- triggers
  - copying 236
  - list 235
  - moving 236
- Triggers tab 235
- Troubleshooting
  - Import Log Messages 134
- troubleshooting
  - build errors 463
  - Cannot code-sync 537
  - Cannot code-sync filename beyond line
    - lineNum 537
  - Capsule Role name same as Capsule
    - name 464
  - Check Environment Variables 465
  - Compile Fails on Valid C++ Models with
    - VC++ 5.0 or VC++ 6.0 467
  - Compiler not installed correctly 464
  - compiling a component 443
  - Could not find trailing CodeSync tag for 537
  - Ensure that Component has correct Make
    - types configured 465
  - Error Linking Capsule (error from
    - nmake 467
  - Linking 468
  - Linking wrong Services Library set 464
  - Missing Class Dependencies 464
  - Missing Header Files, Object Files, and
    - Libraries 466

- name conflicts 466
- Redefinition of basic types or multiple declarations for X 463
- Review your compiler flag settings 465
- setting breakpoints 500
- Source File Compilation 467
- System does not understand the make command 465
- unable to compile a component 443
- Unknown command, command not found, the name specified is not recognized 463
- Unresolved symbol or undeclared identifier 463
- Warning
  - Use tabs for indenting code-sync regions 537
- Windows NT Compilation Command Line Limits 467
- Tutorials 2

## U

- UML Options 209
- UML options 209
  - Base UML notation 209
  - Show Classifier Name on Roles 209
  - Show Protocol Name on Ports 209
- Uncheckout operations 412
- Undocking the capsule diagrams 354
- unidirectional aggregate association 152
- unidirectional association 151
- unique Id's 125
- unique ids 125
  - cautions 126
  - correct merge scenario 127, 128
  - incorrect merge scenario 127
  - model elements not having 125
- unit information 407
- Unit Information tab 101
- Unknown compiler message stream 114, 451
- unload
  - Windows CE component instance 521

- unloading
  - debugger 525
- unreserved check out 416
- unwired port
  - create using a system protocol 213
- updating cross-references 403
- Usage tips 139
- use case
  - creating 141
  - specification 141
- Use case diagram editor, using the 138
- Use case diagram toolbox 139
- use case diagram, creating a 137
- Use case specification 141
- use case, creating a 141
- User-specific Working Environment Settings (.rtusr and .rtwks) 132
- Using OLE 542
- Using sort 104
- Using the Class diagram editor 146
- Using the collaboration diagram editor 222
- Using the Component diagram editor 377
- Using the Deployment diagram editor 380
- Using the Sequence diagram editor 305
- Using the state diagram editor 230
- Using the structure editor 208
- Using the use case diagram editor 138
- Utility scripts 511
- utility scripts 511

## V

- Valid Applications 168
- validation error log 319
- Validation tab 350
- View menu 56
- virtual functions 198
- virtual inheritance 164
- virtual operations
  - overriding 200
- Visibility 337
- visibility
  - implementation 349

## W

- Watch tab 115
- watch values
  - refreshing 497
- Watches 483
- watches
  - refreshing values 115
- WCESH3.bat for Windows CE 518
- What's This Help 1
- When to use the different kinds of traces 492
- Where to start 438
- Window menu 76
- Window Selectors command 77
- Windows CE
  - Basic Mode 518
  - configuring a component instance 518
  - connection delay 520, 523
  - debugger mode 522, 523
  - location of script 519
  - RRT\_WINCE\_TARGET\_DIR 519
  - Target Timeout 520, 524
  - tasks to configure component instance 518
  - unload 521
  - WCESH3.bat 518
- Windows CE operation for component instance 514
- Windows NT Compilation Command Line Limits 467
- wired port 215
- word wrap 110
- workflow 26
  - analysis and design 26
  - change management 27
  - configuration 27
  - implementation 27
  - testing 27
- workflows 26
- working environment settings 132
- Working with the Dialog Editor 74
- workspace
  - definition of 131

## X

- xxdgb
  - debugger mode 524

## Z

- Zoom tool 91