Rational Software Corporation

User's Guide

RATIONAL QUALITY ARCHITECT REALTIME EDITION

VERSION: 2002.05.20

PART NUMBER: 800-025107-000

WINDOWS/UNIX

Rational®
the software development company

support@rational.com
http://www.rational.com

Part Number: 800-025107-000

Version Number: 2002.05.20

Contents

Preface

This manual provides an introduction to Rational Quality Architect RealTime (RQA-RT). The User's Guide extends Rational Rose RealTime's design automation capabilities to model, debug, and test. You can automatically generate complete unit and integration test harnesses directly from sequence diagram specifications.

Audience

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

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 **Rose RealTime Online Documentation** from the **Start** menu.

- For more information on training opportunities, see the Rational University Web site: http://www.rational.com/university.

Contacting Rational Technical Publications

To send feedback about documentation for Rational products, please send e-mail to our Technical Documentation Department at techpubs@rational.com.

Contacting Rational Technical Support

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

Your Location	Telephone	Fax	E-mail
North America	(800) 433-5444 (toll free) (408) 863-4000 Cupertino, CA	(781) 676-2460 Lexington, MA	support@rational.com
Europe, Middle East, Africa	+31 (0) 20-4546-200 Netherlands	+31 (0) 20-4546-202 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 Technical Support, please be prepared to supply the following information:

- Your name, telephone number, and company name
- Your computer's make and model
- Your computer's operating system and version number
- Product release number and serial number
- Your case ID number (if you are following up on a previously-reported problem)

Introduction to Rational Quality Architect RealTime (RQA-RT)

1

Contents

This chapter is organized as follows:

Overview

The Rational Quality Architect (RQA-RT) User's Guide extends Rational Rose RealTime's design automation capabilities to model, debug, and test. You can automatically generate complete unit and integration test harnesses directly from sequence diagram specifications. This eliminates the expensive and time consuming manual coding of stubs and drivers for debugging and testing.

Rational Quality Architect - RealTime supports C++ and Java languages. For language-specific information, see the appropriate reference guide:

- *Rational Rose RealTime C++ Reference*
- *Rational Rose RealTime Java Reference*

RQA-RT automatically verifies 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.

A use case defines a set of use case instances, where each instance is a scenario that describes the sequence of actions that a system performs. You can capture scenarios as UML sequence diagrams, also known as MSCs. An MSC is a diagram that describes a pattern of chronologically ordered message interactions among objects.

For model analysis, development, and debugging, RQA-RT helps you to deliver higher quality code to market, in less time, with less risk. Early disciplined debugging and testing is a key element of successful iterative design strategy by identifying risks and offering proof that requirements have been satisfied. It is easier and less expensive to correct bugs early in the design cycle.

With RQA-RT, test harness generation is automated. This reduces the time spent creating and maintaining test harnesses, and provides more time for building, debugging and testing components. Test specifications can be built quickly and are reusable. For testing, you can specify a set of scenarios, which completely cover the requirements of a component, including failure modes. Then, the scenarios can be run, one at a time or as a group, to validate any changes made to a component. These test scenarios are represented as sequence diagrams in the RQA-RT Add-In and all testing framework is based on these sequence diagrams.

Early detection of design oversights is also critical for reducing the risk of project failure. For example, race conditions in real time systems are difficult to predict and even more difficult to find. For development, you can catch potential race conditions at analysis time, before even beginning design work. Causal analysis of sequence diagrams has also been enhanced.

RQA-RT Add-In Framework

The RQA-RT Add-In introduces a number of components to support the verification process. When you execute a verification, framework components are generated automatically and displayed in the model browser. For the most part, the framework components are viewed as transparent to the user. However, for advanced users, customization of the components is possible (See *Framework Components* on page 80).

Capsule Under Test

The capsule representing a system or sub-system to test is referred to as the Capsule Under Test (CUT). It normally contains many low level components and may itself be a component of another capsule. Multiple CUT's may be placed in an interaction and tested simultaneously.

Specification Sequence Diagram

For a sequence diagram to qualify as a specification, it must describe the following:

- An interaction between capsule instances with specified roles
- Synchronous and asynchronous communication between ports
- Internal messages

Since RQA-RT supports testing of multiple CUT's (Capsule Under Test), there are very few limitations on what makes a Sequence Diagram a specification. Almost any interaction can be set as specification and run through the verify procedure as long as the capsule instances specify roles with a port interface, and the messages indicate which ports they are sending to and receiving from.

Model Example

Test scenarios are represented as sequence diagrams in the RQA-RT Add-In. These sequence diagrams can be verified through the RQA-RT verify behavior functionality. For an example of a use-case provided through a sequence diagram, see the *ReliableService* Model Example in the following directory:

```
$ROSERT_HOME/Examples/Models/C++/ReliableService
```

Approaches to using Rational Quality Architect RealTime (RQA-RT)

2

Contents

This chapter is organized as follows:

Overview

The RQA-RT functionality is built in to Rational Rose RealTime. You can model a RQA-RT unit test by creating a Use Case diagram to represent it.

There are four approaches to utilizing the RQA-RT Add-In:

- Scenario-Based Debugging
- Specification Verification
- Specification-Based Development
- Regression Testing

Scenario-Based Debugging

RQA-RT gives you full control over the execution of a model. Because debugging is simplified, it encourages frequent, disciplined testing. An executable framework is generated automatically from a sequence diagram specification, the framework is run, and the execution results graphically are compared against the original sequence diagram specification.

RQA-RT can record and save the run-time interactions in a new sequence diagram. Incomplete or partial sequence diagrams can be used to incrementally discover or deepen your understanding of runtime component interactions. By choosing multiple sequence diagrams, an entire set of scenarios can be tested simultaneously.

With RQA-RT there is no need to write smart stubs for missing components or functionality. Simply stub out unavailable components in the interaction. RQA-RT generates portable C++ drivers and harnesses so you can debug on the host without tying up target resources, and validate components directly on the target later.

The RQA-RT Add-In delivers enhancements to the current debugging environment. By combining debugging styles with automation, it provides observation, interaction and generated test harnesses.

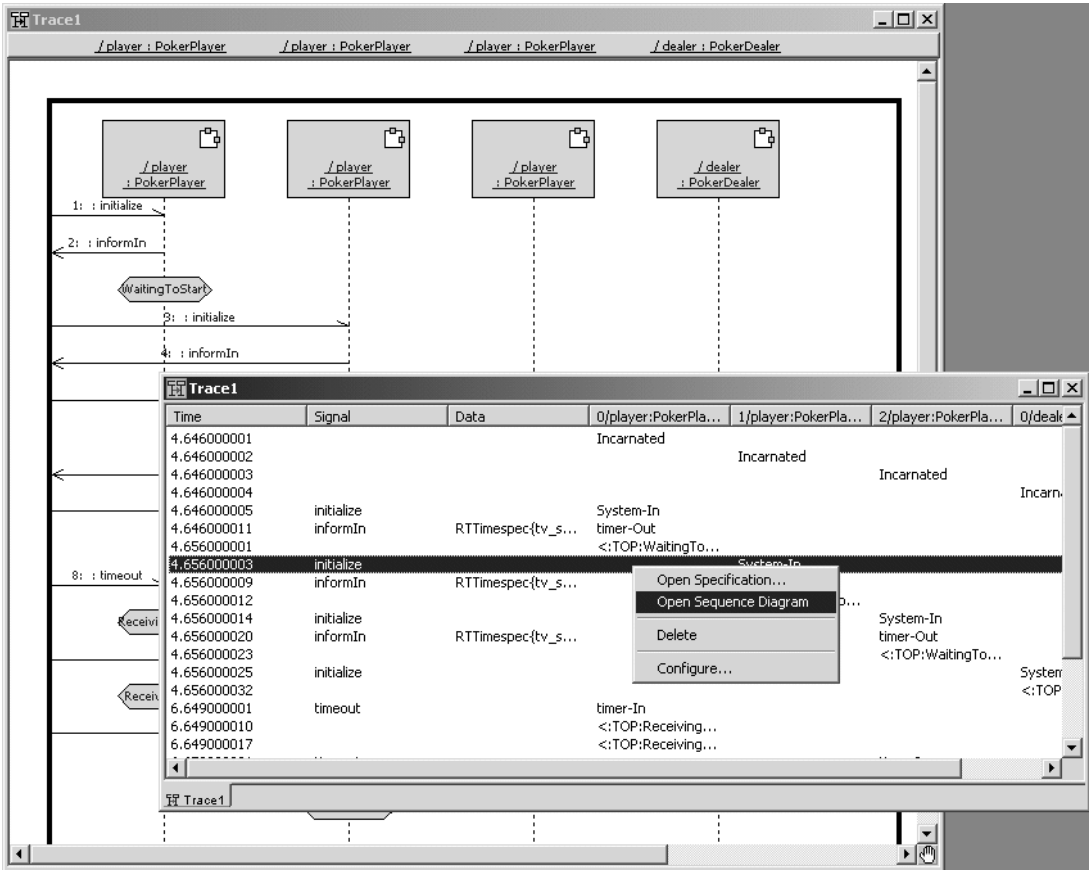
Components of natural evolution in model debugging are:

- Design observation using trace, animation and watch points.
- Interaction through message injection, model breakpoints and source breakpoints.
- Test Harness package can be observed / modified after the test.

The sequence diagram defines an execution scenario of an application. By using the sequence diagram to specify the messages to send to an executing capsule, and by tracing the resulting messages sent from the executing capsule, you can create a fully automatic capsule test and debugging environment using sequence diagrams as the execution scripting language.

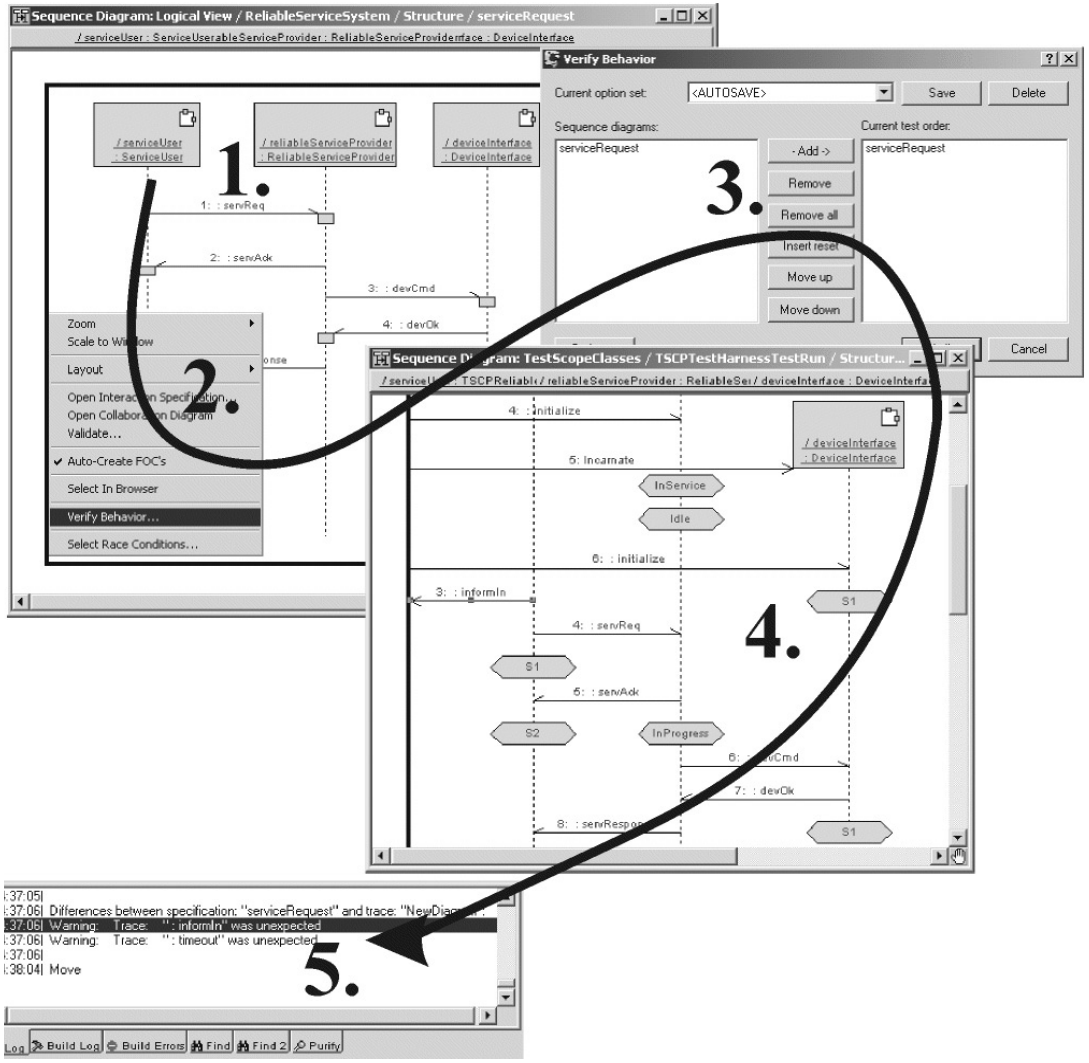
RQA-RT can automatically create executable test harnesses similar to that shown in Figure 1, directly from the sequence diagram, and can use them to automatically observe and verify the behavior of running applications during test or debug. You can test components against a series of tests before submitting the components to the build.

Figure 1 Run-time message tracing and sequence diagram generation



The automatic scenario verification sequence in Figure 2 shows how you can use a sequence diagram to verify the behavior of a component in the wireless network.

Figure 2 Automatic scenario verification



The numbers in the diagram correspond to the following actions:

1 From the sequence diagram(s) associated with a model, generate a test harness to automatically verify run-time behavior.

2 Choose verify behavior from the right mouse context menu of the sequence diagram or collaboration.

3 Select the sequence diagram(s) to verify and select appropriate options for the test run.

4 Select **Verify** to automatically generate a complete executable test harness for the selected sequence diagram(s), including the application and the components.

5 Compare the generated sequence diagram with the specification sequence diagram. Mismatched signals are identified using a causal differencing engine and the results are output to the Rose RealTime Output log where they can be double clicked to graphically navigate to the problem area in the trace / specification diagrams.

Specification Verification

The RQA-RT Add-In lets you test systems from a Black Box or White Box perspective.

With Black Box testing, Specification sequence diagrams are constructed with only the CUT instance. The goal is to verify the message flow at the interface of the system.

With White Box testing, component capsule instances are added to the Specification sequence diagram to allow internal system behavior to be checked.

Specification-Based Development

Because any capsule in a RQA-RT test harness can be a driver, you can design and run a test harness with all capsules stubbed out. This means you can run your test harness with fully executed specifications, before writing any code. To integrate your test harness using specification-based development:

1 Define the capsule structure of the system, at least down to the level of the specification sequence diagrams.

2 Construct the specification sequence diagrams if they have not already been created or supplied as part of the analysis phase.

3 Build a test harness in which every capsule is stubbed out by making every sub capsule a driver on the **Verify** options tab.

The resulting test suite will pass all test cases because every capsule will behave exactly as defined in the specification sequence diagrams.

4 Replace the stubbed capsules with real implementations one capsule at a time.

Note: Real behavior can be introduced once piece at a time within a capsule, since sub-capsules can be drivers too.

As each capsule is implemented it is removed from the test harness driver list. Eventually, no drivers remain. At this point the system is complete and the time spent debugging has been minimized, since the system was integrated from the start.

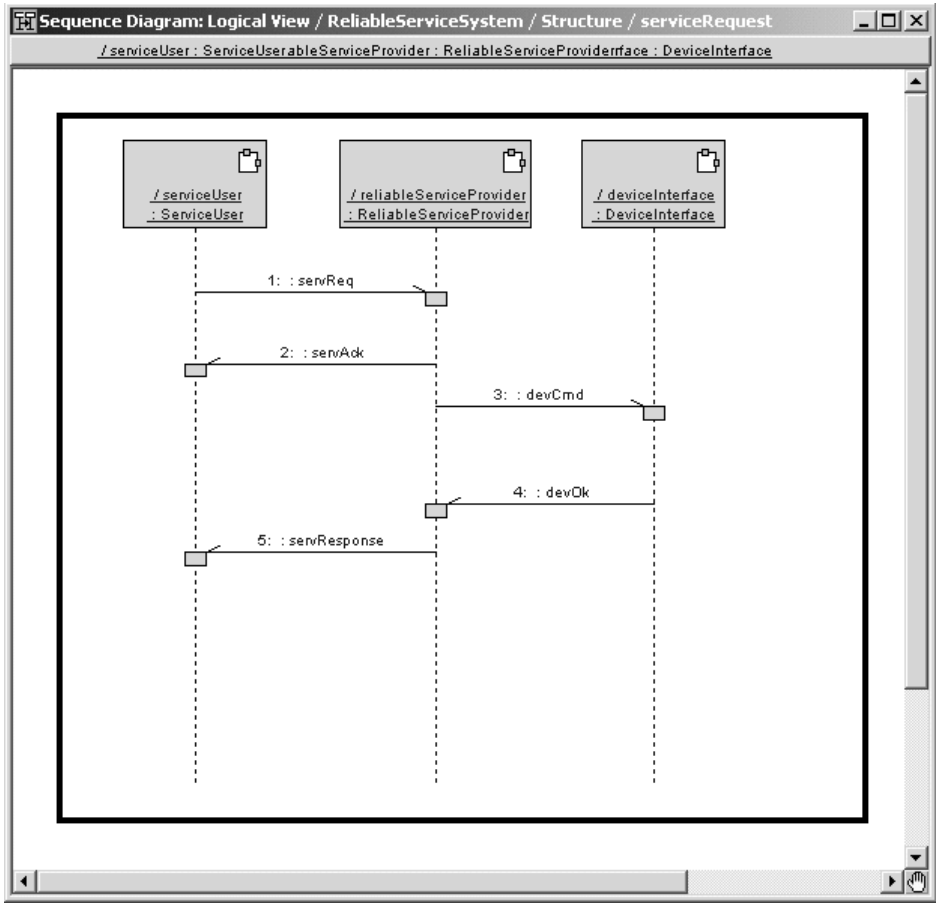
Regression Testing

The RQA-RT Add-In lets you verify system behavior as the system evolves over time:

1. Initially, a specification sequence diagram is constructed with the CUT (see *Capsule Under Test* on page 12) and component capsules of interest. Only CUT interface messages are included - no internal message flow is specified. This specification sequence diagram is executed with the RQA-RT Add-In.
2. The resulting run sequence diagram captures the initial system behavior.
3. To use this sequence diagram as a specification in future tests as a means of monitoring system development, move the trace sequence diagram into its real collaboration by dragging and dropping the sequence diagram in the Rose RealTime browser.

Note: Batch mode support to automate regression testing is available through scripting capability. For additional information, see *RQA-RT Batch Mode Introduction* on page 87.

Figure 3 Example of an sequence diagram



A sequence diagram can serve as a specification of desired system behavior - this type of diagram is referred to as a specification diagram. A useful form of testing is to use a specification diagram to create a wrapper which drives a system with appropriate input messages. The resulting system behavior, a trace diagram, can be compared with the expected behavior. Manually running the comparison can be tedious and error-prone. The RQA-RT Add-In simplifies this process by providing:

- Support for the creation of a specification diagram
- The automatic generation of one or more drivers
- The execution of the generated test harness and drivers
- Verification of the resulting output trace diagram

Debugging Scenarios

There are several debugging scenarios with RQA-RT capabilities. You can script the execution of individual capsules (for example, unit test) or capsule collaborations (for example, integration test). Some common usage scenarios are:

1. Specify an initial message, and the participants in a specification. Verify to discover what the model will do. Take the trace diagram from the verify and add another stimulus message, verify again to explore the model deeper. Use race condition analysis on the resulting diagram to look for design problems.
2. Specify all the stimulus messages, but no other messages. After a verify, the unspecified messages are filled in. Use race condition analysis on the resulting sequence diagram to look for design problems.
3. Capture the current behavior of a capsule in trace diagrams. After making changes, for example, introducing additional functionality, re-verify the sequence diagrams to make sure that the changes did not break the existing functionality.
4. Smart stubbing. Use the sequence diagram to describe the behavior of not-yet-available functionality capsules that the CUT must later integrate with Test.

   A stub is a component, or complete implementation subsystem, containing functionality for testing or debugging purposes. When you use an incremental integration strategy you select a set of components to be integrated into a build. These components may need other components to be able to compile the source code, and execute the tests. This is specifically needed in integration test, where you need to build up test specific functionality that can act as stubs for things not included or not yet implemented. The styles used are:

   - Stubs that are simply dummies with no functionality other than being able to return a pre-defined value
   - Smart stubs that are more intelligent and can simulate a more complex behavior.

   If you have complete components that cannot be executed on their own, you can use smart stubbing to test these components individually, without the components on which they depend.

Smart stubbing should be used with discretion since it takes more resources to implement. You need to be sure it adds value. You may end up in situations where your stubs also need to be carefully tested, which is very time consuming. RQA-RT automates the creation of smart stubs.

5 Test. Use the sequence diagram to describe the expected behavior of an capsule and then to verify that the implementation has been done correctly.

Creating Specification Sequence Diagrams

3

Contents

This chapter is organized as follows:

Overview

For a specified sequence diagram, the toolset automatically:

- Generates a test framework
- Runs the component under test
- Compares the results of the test run against the specification sequence

The sequence diagram used to drive the test is the specification. The component(s) under testing is the Capsule Under Test (see CUT, *Capsule Under Test* on page 12). A CUT may be comprised of any number of sub-components.

Understanding Rational Quality Architect (RQA-RT)

RQA-RT is an Add-In for Rational Rose RealTime, and can be installed with the Professional Edition of Rational Rose RealTime. After you install Rational Rose RealTime Professional Edition, create new specification sequence diagrams or modify the existing ones to take advantage of the RQA-RT capabilities.

RQA-RT uses the information in your specification sequence diagram to create the test framework. As a result, you must provide more detail in your specification diagram than what you may have provided in previous sequence diagrams. For

example, the specification must contain an instance corresponding to the CUT. Similarly, any sub-capsule of the CUT that plays a role in the specification sequence diagram must be associated with a valid capsule instance.

Finally, because the test harness framework drives the CUT through its external interface, the specification diagram must specify the following:

- the port and signal names for all messages between the CUT(s)
- drivers
- internal components
- if talking through a SAP, the environment

There are two main approaches to create the detailed specification sequence diagram:

- manually build a sequence diagram from scratch
- work from a sequence diagram generated by an RTS trace.

Manually Creating Specification Sequence Diagrams

When manually creating the specification sequence diagrams, use the Rational Rose RealTime sequence diagram editor. To manually create a specification sequence diagram:

1. Create a new specification sequence diagram containing an instance of the CUT(s) to test and driver(s) and, optionally, any components of the CUT(s) that are of importance to the test.

   See *Creating a New Specification Sequence Diagram* on page 27.

2. Specify the interaction between the CUT(s) and the environment.

   See *Specifying Replicated Capsule Instances* on page 31.

Creating a New Specification Sequence Diagram

Creating a specification sequence diagram is no more difficult than creating a normal sequence diagram. You must specify instance information in addition to naming the capsules in the specification sequence diagram. To specify instance information, use one of the following methods:

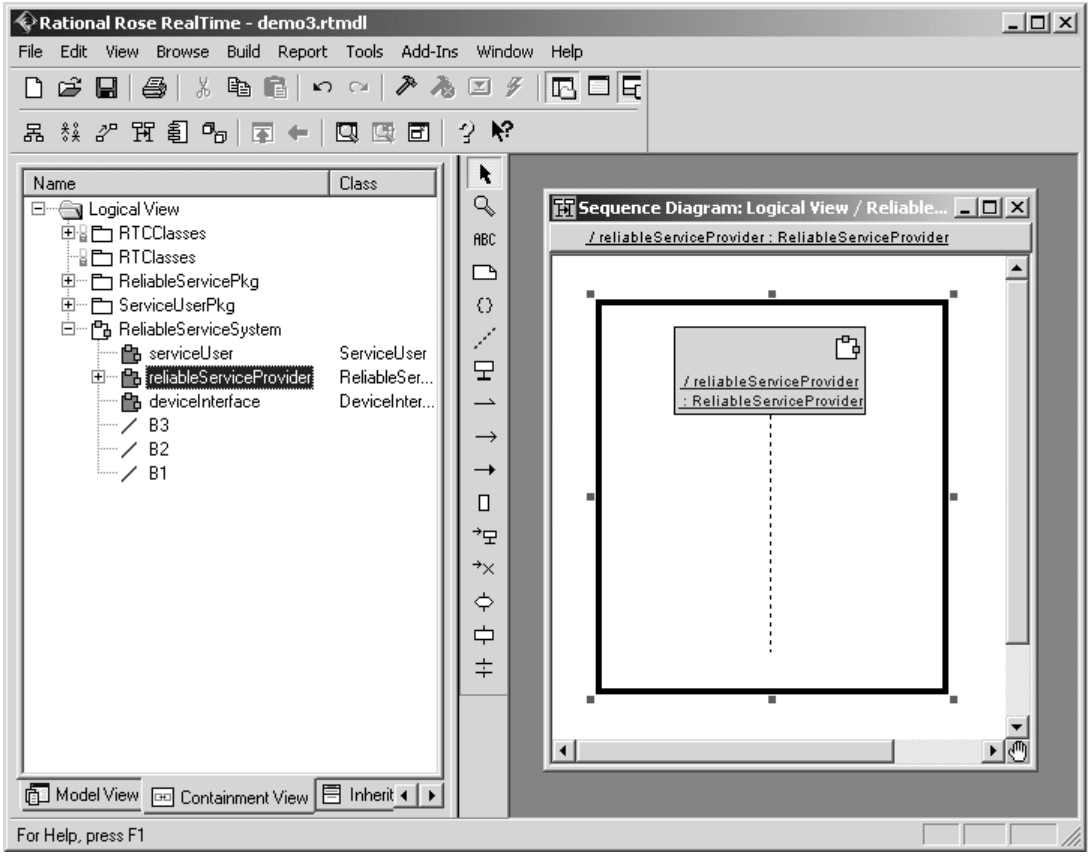
- Drag and drop the instances onto the sequence diagram one at a time. See *Adding Instances One at a Time* on page 27.
- Create all the instances at once. See *Adding all Instances at Once* on page 28.
- Draw the sequence diagram and add the instance information manually. See *Adding Instances Manually* on page 30. This method involves the most work, and is recommended if you have to add instance information to an existing sequence diagram.

Adding Instances One at a Time

To create a sequence diagram containing a fully specified instance of the CUT(s) and any components of interest:

1. In the model browser, select the collaboration or structure diagram on which the sequence diagram is to be based, as shown in Figure 4.
2. Right click on the collaboration or structure diagram and select **New / Sequence Diagram**.
3. Provide a unique name for the newly selected sequence diagram in the model browser.
4. Double click on the new sequence diagram, or right click on it and select **Open**.
5. Go to the containment view browser and find the capsule that contains the instances you wish to test.
6. Drag and drop the applicable capsule roles from the browser onto the new sequence diagram.

Figure 4 Adding instances one at a time



Adding all Instances at Once

This is the quickest way to manually create a specification sequence diagram.

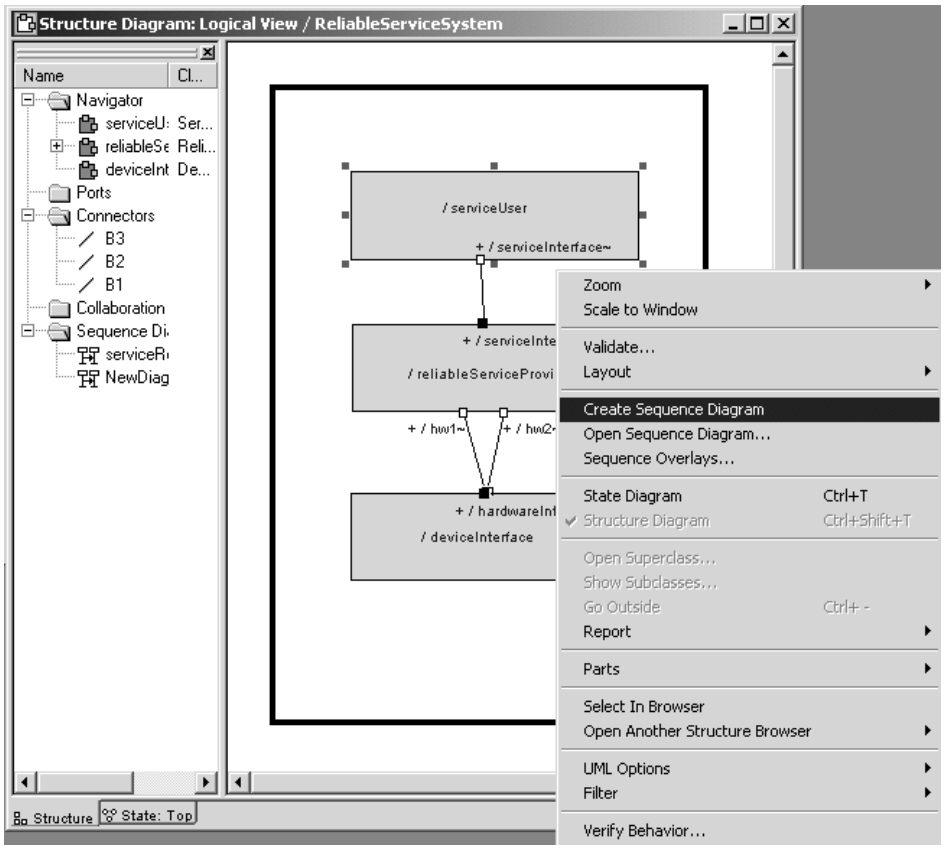
To add all instances at once:

1. Go to the structure diagram or collaboration diagram under which the sequence diagram will be based.
2. Optionally, select one or more capsule roles by shift clicking on them or using the drag selection capability.

3 Right click on the background of the structure or collaboration diagram and select **Create Sequence Diagram** as shown in Figure 5.

4 The sequence diagram opens with interaction instances for all the selected roles. A default name is provided. Right-click and select **Select In Browser** to rename the diagram to something more appropriate.

Figure 5 Adding all instances at once



If you miss any instances in your first pass, you will have to add each missed instance as described in *Adding Instances One at a Time* on page 27.

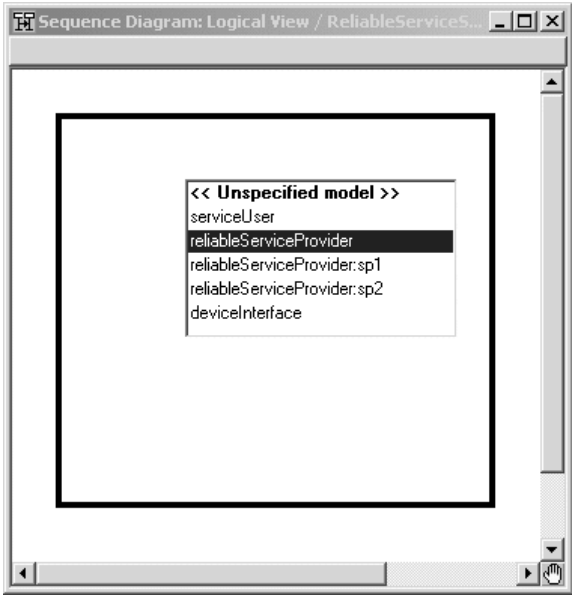
Adding Instances Manually

If you are working with existing sequence diagrams, you will want to add any required instances manually.

To add instances manually:

1. In the model browser, select the collaboration or structure diagram on which the sequence diagram is to be based.
2. Right click on the collaboration or structure diagram and select **New / Sequence Diagram**.
3. Provide a name for the newly selected sequence diagram in the model browser. See Figure 6 for an example of names.
4. Double-click on the new sequence diagram, or right click on it and select **Open**.
5. Add a new instance to the sequence diagram using the **Interaction Instance** tool.
6. From the shortcut menu that appears, choose a capsule instance path on which to base the instance.

Figure 6 Adding instances manually



Specifying Replicated Capsule Instances

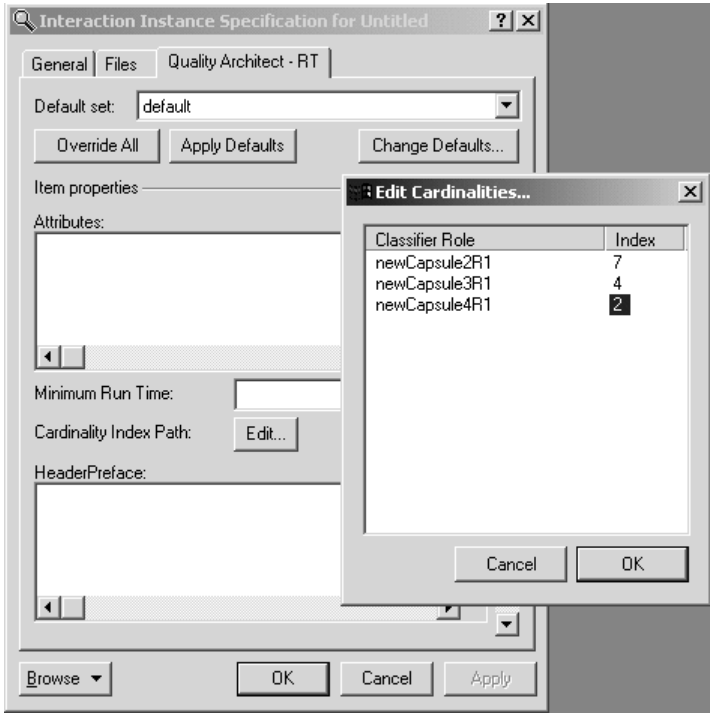
You can either specify the replication index for Replicated Capsule Instances, or accept the default replication index of 0.

To specify a replication index:

1. Open the specification dialog on the capsule instance of the replicated reference.
2. Go to the **RQA-RT** properties tab and click on the **Cardinality Index Path** button to open an **Edit Cardinalities** dialog.
3. In the **Edit Cardinalities** dialog, click on the applicable capsule reference(s), then specify the desired replication index in the index field. (see Figure 7.)

Note: For any reference that has a container capsule(s) that is replicated, the replication index of the parent(s) can be specified. This allows the specification of a reference that is contained in a *nested* hierarchy of replicated capsules.

Figure 7 Edit Cardinalities dialog



The specification sequence diagrams are not automatically updated by the toolset. Therefore, changes that you make to the decomposition hierarchy of the model could invalidate the capsule path or replication indices of specification interaction instances.

Automatically Generating Specification Sequence Diagrams

In some situations, it may be more expedient to use an existing model to aid in the generation of a specification sequence diagram. For example, complex sequence diagrams with many instances and messages may be better constructed automatically. You can use RQA-RT or the message trace feature.

Automatically Generating Specification Sequence Diagrams using the RQA-RT Add-In

To generate a specification sequence diagram using the RQA-RT Add-In:

1. Start building a sequence diagram manually as detailed in *Manually Creating Specification Sequence Diagrams* on page 26.
2. Add the instances that you would like to trace, and any messages involving the environment. Do not add anything else.
3. Verify the behavior of this sequence diagram by clicking **Verify Behavior** from the shortcut menu.
4. Modify the resulting trace sequence diagram to meet the specification sequence diagram requirements detailed above.
5. Save as the new specification sequence diagram.

Automatically Generating Specification Sequence Diagrams using the Message Trace feature

To generate a specification sequence diagram for a single CUT using the message trace feature:

1. Build a wrapper capsule that has internal end ports that mirror the external interface of the CUT.

   Specify the CUT as a sub-capsule of the wrapper capsule.

   - Bind the internal end ports with the external ports of the CUT.
   - Place interface ports that mirror the internal end ports on the wrapper.
   - Have the behavior of the wrapper forward messages into the wrapper on to the component capsule.

2. Create a component and set the target configuration to a known platform.
3. Drag the component to a processor to create a component instance.
4. Build and run the component instance.
5. When target observability is active, place daemons on the wrapper interface ports.

   Create inject messages for the daemons.

6. Open a trace window on the CUT and sub-capsules of interest.
7. Run the RTS and inject messages in the daemons as appropriate.
8. When completed, create a sequence diagram from the trace in the message trace window and save it.
9. Edit the new sequence diagram and ensure that it meets the specification sequence diagram requirements as detailed above.
10. Save it as the new specification sequence diagram.

Creating Capsule Unit Test Specification Sequence Diagrams

When developing a capsule it may be useful to test its behavior and interface in isolation without any dependencies. This is called unit testing. RQA-RT provides some methods for unit testing that do not depend on outside capsule contributors. You can create a new sequence diagram, or use an existing sequence diagram to perform unit testing.

Unit Testing a Capsule

When unit testing a capsule, you need to create or use existing sequence diagrams from which a testing harness is generated. The testing harness automates a sequence of tests on a capsule or an instance.

To unit test a capsule:

1. Create a new collaboration diagram.
2. Add a role of the capsule to be tested onto the collaboration diagram.
3. Create a sequence diagram for the role of the capsule by selecting the role and right clicking to select **Create Sequence Diagram**.
4. Add an unspecified interaction instance to the sequence diagram to send messages to the CUT (see *Capsule Under Test* on page 12). The unspecified instance is automatically generated as a driver when **Verify Behavior** is run.

   On the sequence diagram, specify the ports for message ends corresponding to the capsule under test on the send message port detail page. Port names on the message side corresponding to the interaction instance with unspecified roles, or port names which you have specified, may be omitted and subsequently generated by RQA-RT.
5. To run the test, select **Tools - Rational Quality Architect - RealTime Edition - Verify Behavior**. A trace is generated each time a test is run.
6. Run RQART to test different use cases based on how the CUT reacts to the messages. For example, after you run a test you can view the results, modify test parameters and run the test again to see what changes.
7. Repeat steps 1 to 6 for each new standalone capsule you create in the test harness.

Note: Ensure that normal restrictions on port connectivity are observed. For example, no port from a capsule under test can be used by two different ports derived from a testing interaction instance.

8. Click **Tools > Rational Quality Architect - RealTime Edition > Verify Behavior** to test the instance.

Customizing your RQART Sequence Diagram

4

Contents

This chapter is organized as follows:

Modifying Capsule Behavior with Message Flows

The RQA-RT Add-In lets you modify the behavior of Capsules Under Test (CUT's) by driving them through their external interface. You can modify the behavior via the concept of drivers - instances that drive the interface of a CUT.

If a driver interaction instance is invoking a timing service, InformIn, the timing service gets generated into the driver capsule.

All messages to and from a CUT's external ports can be from a driver or from another CUT. Although stimulation is primarily focused on CUTs, RQA-RT can also stimulate CUT components to facilitate simulation of SAP/ SPP layer communication. You must specify that these messages are sent to and from the environment. Only system messages can be sent to and from the environment, since the environment is not the driver.

You should also specify system service messages to and from a CUT or its component capsules.

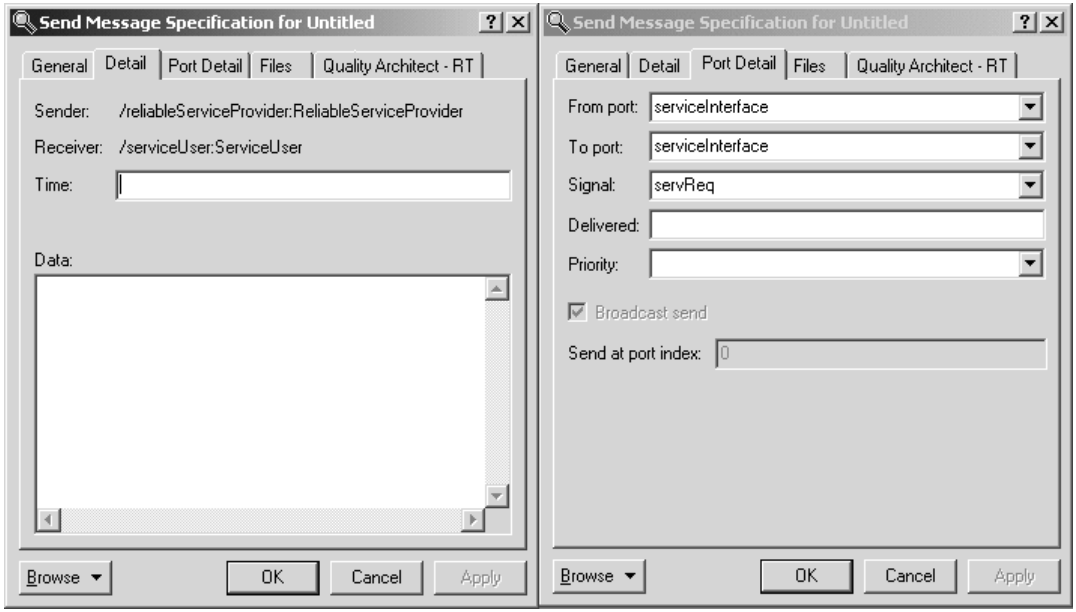
When running multiple specifications in a test, in sequence, ensure that you complete each sequence when using a CUT so that the CUT is self-contained.

Adding Messages to a Specification Sequence Diagram

To properly add messages to a specification sequence diagram:

1. Add a message to the specification sequence diagram by using the **Asynchronous Send Message** Tool from the sequence diagram tool palette.
2. After adding a new message, activate a specification dialog (see Figure 8) by double clicking on the message or right clicking on the message and selecting **Open Specification** from the local menu.
3. In the specification dialog, specify the Signal, Port, Priority, Data and port index attributes of the message. These attributes are discussed below.

Figure 8 Send Message Specification Dialog



Signal

All valid signals for a selected port are listed in the **Signal** combo box.

All valid out-signals for the port through which the following message types are sent, are listed in the Signal pane.

- messages destined for the environment
- messages travelling between capsule instances

For messages originating from the environment, all valid in-signals for the port receiving messages originating in the environment, are listed in the **Signal** pane.

All valid in-signals and out-signals for relay ports are listed in the Signal pane for the associated port.

To specify a message signal, select the desired signal name.

Specification of a signal is optional (*) for any internal message between component capsules. However, for messages to and from the environment or to and from a driver, the signal must be specified. If it is not specified, the RQA-RT Add-In reports an error in the specification sequence diagram.

Although specification sequence diagrams generally use the signal name as the message label, which is untitled or (*) by default, you can modify the message label by using the message specification dialog, or by directly using the Text Tool on the sequence diagram tool palette.

When a trace is specified to trace relay ports, the information is saved between sessions. This is accomplished through toolset simulation. The RTS is not actually tracing the messages through **relay ports**; the toolset adds the sub-capsules to the trace behind the scenes and creates a mapping from the final end-ports to the relay ports the message passed through. This is available through MSC tracing.

If the signal name is specified to be something other than untitled or (*), then modifying the message label will not automatically change the signal name - it will keep its specified name.

Note: Trace sequence diagrams created by RQA-RT always default the message label to that of the signal name.

Passing an Object

RQA-RT supports data specified in the send signals through ASCII-encoded strings. The format is the same used for the data injected in a probe. The format 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.

For **C++**, use the following:

<*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*> an attribute of a composite type (such as a type composed of other attributes). For example, another class.

<*basic attribute*> the name of an attribute of a basic type (such as **int**, **long**, **short**, **char**, **enum**, **double**, **float**, and **string**).

<*value*> the value of an attribute of a basic type.

For additional information on the injected data format, see *Probe Specification* in the online help book *Toolset Guide*.

For **Java**, use the following string:

<type> <constructor_arguments>

where:

type

constructor_arguments

For example, for the following object:

```
new MyType(Friday,"WeekendIsNear")
```

the data field will be:

```
MyType Friday,"WeekendIsNear"
```

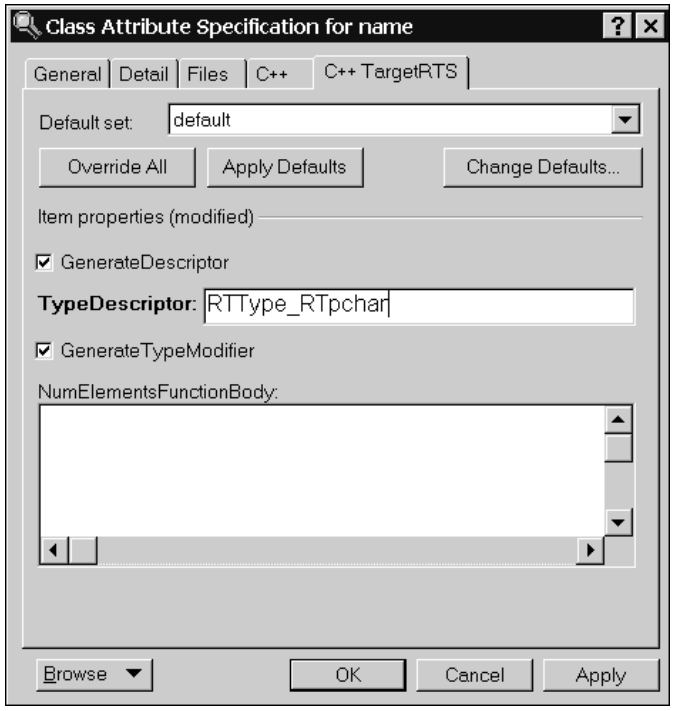
Passing an Object with the String Field

Passing objects by pointers requires additional steps. Below, you will find example steps required to pass the object containing "char*".

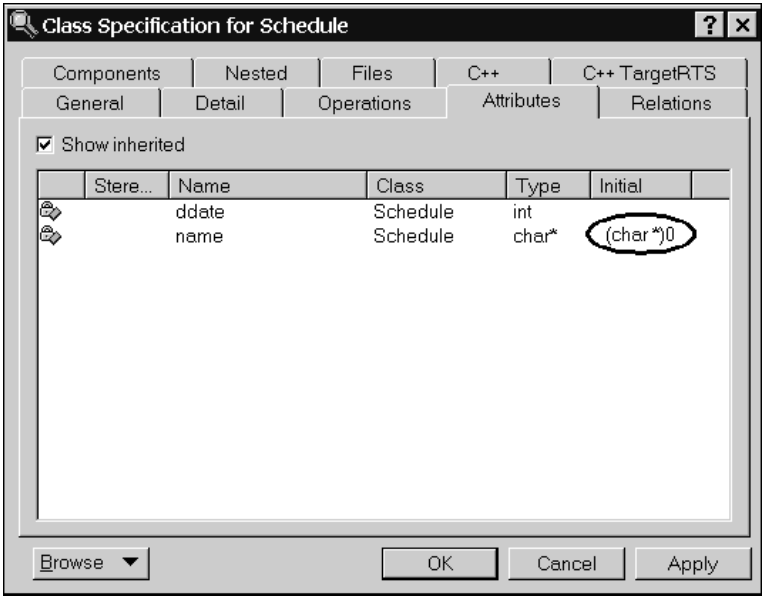
Note: Set the **type descriptor** for this field to **RTType_RTpchar** and specify proper memory allocation and de-allocation in the class constructor, copy constructor, assignment operator, and destructor.

To successfully pass an object with the string field, you must:

- Set the initial value to (char *)0 for the attribute.



- Set the Type Descriptor **RTType_RTpchar** for the attribute (see steps 6 through 8 in the example below).



- Override the default copy constructor (see steps 10 through 23 below).
- Override the assignment operator (see steps 24 through 35 below).
- Override the default destructor (see steps 36 through 40 below).

For example, a class called **myClass**, has the field **char *name**. To ensure proper decoding of this object when passing this object as signal data, do the following:

1 Select a class, right-click and click **Open Specification**.

2 Click the **Attributes** tab.

3 Right-click and click **Insert**.

4 Set the **Name** to *name*, press `ENTER`, and press `TAB` to advance to the **Type** column.

5 Press `F8` and type **(char *)0**.

6 Double-click on the new attribute name.

The **Class Attribute Specification for name** dialog displays.

7 Click the **C++ TargetRTS** tab.

8 In the **TypeDescriptor** box in the **Item properties** area, type the following type descriptor:

   RTType_RTpchar

9 Click **OK**.

Next, you want to override the default copy constructor.

10 Click the **Operations** tab.

11 Right-click and click **Insert**.

12 Set the **Name** to *myClass*, press ENTER.

13 Double-click on the new operation name.

   The **Operation Specification for myClass** dialog displays.

14 Click the **Detail** tab.

15 In the **Parameters** area, right-click and click **Insert**.

16 Set the **Name** to *other*.

17 Press TAB to advance to the **Type** column.

18 Press F8 and type the following:

```
const myClass &
```

19 In the **Code** area, type the following:

```
if( name != (char *)0 )
name = RTMemoryUtil::strdup( name );
```

20 Click **Apply**.

21 Click the **C++** tab.

22 In the **ConstructorInitializer** box, type the following:

```
:name( other.name ) [, other_fileds_initialization]
```

23 Click **OK**.

   The **Operation Specification for myClass** dialog closes and the **Class Specification** dialog displays.

Next, you want to override the assignment operator.

24 Right-click and click **Insert**.

25 Set the **Name** to *operator*=, press ENTER.

26 Press TAB to advance to the **Return Type** column.

27 Press F8 and type the following:

```
myClass &
```

28 Double-click on the new operation name.

The **Operation Specification for operator=** dialog displays.

29 Click the **Detail** tab.

30 In the **Parameters** area, right-click and click **Insert**.

31 Set the **Name** to *rhs*.

32 Press TAB to advance to the **Type** column.

33 Press F8 and type the following:

```
const myClass &
```

34 In the **Code** area, type the following:

```
if( this != &rhs )
{
if( (name = rhs.name) != (char *)0 )
   name = RTMemoryUtil::strdup( name );
[other_field = rhs.other_field];// assign other fields
}
return *this;
```

35 Click **OK**.

The **Operation Specification for operator=** dialog closes and the **Class Specification** dialog displays.

Next, you want to override the default destructor.

36 Right-click and click **Insert**.

37 Set the **Name** to *~myClass*, press ENTER.

38 Double-click on the new operation name.

The **Operation Specification for ~myClass** dialog displays.

39 Click the **Detail** tab.

40 In the **Code** box, type the following:

```
delete [] name;
```

Port

You can specify the receiver and sender ports in the from and to port combo boxes respectively. The port combo boxes list all valid ports for the CUT and component capsule classes. A message port is specified by selecting the desired port name.

Specification of a message output port is optional (*) for any internal message. However, all messages involving the environment must have a port specified; if not, the RQA-RT Add-In will report an error in the specification sequence diagram.

For a message from an instance to the environment, or from an instance to a driver, the name of the port through which the message is sent must be specified.

For a message from the environment to an instance, or from a driver to an instance, the name of the port through which the message is received must be used.

Message Priority

The Priority pane combo box lists all the priorities a message can possess. A message priority is specified by selecting the desired priority level. Specification of a message priority is optional for all messages.

Message Data

The message data area appears in the detail tab of the message specification. This area contains a textual description of the message data that is sent for reference purposes. It is also used by driver capsules as the actual data sent out with the message.

You can specify primitive data to be sent with the message. To send more complex message data such as class objects or arrays, use the composite message data format.

Composite Message Data

RQA-RT supports data specified in the send signals through ASCII-encoded strings.

Note: This format is the same as the inject data format in Rose RealTime. For information on the inject data format see Format of the injected data in the Rational Rose RealTime Toolset Guide.

The Data area of the driver send message is a string representation of the data to be sent with the message. The format of the string depends on the encoding and decoding scheme used by the data type being sent with the message.

Therefore, the format of the driver send 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 send 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

Sending strings within the composite data

If data to be passed contains the string, enter the data in quotes in the data tab. If you need to pass a string as a part of composite data, the following rules apply:

- Specify the initial value for the char field containing the string `(char *)0`
- Set type descriptor for the field to **RTType_RTpchar**.
- Override default copy constructor in order to duplicate memory allocated.
- Override assignment operator in order to duplicate memory allocated.
- Override default destructor in order to prevent memory leaks.

For example, if you are trying to pass an object of a class named **SomeClass** with the field **char *name**, the following alterations will be required:

```
Name: SomeClass
Parameters: Name: other Type: const SomeClass &
if( name != (char *)0 )
name = RTMemoryUtil::strdup( name );

Constructor initializer : :name( other.name ) [,
other_fileds_initialization]

Override assignment operator:
Name: operator=
Return type: SomeClass &
Parameters:Name: rhs Type: const SomeClass &
Code:

if( this != &rhs )
{
if( (name = rhs.name) != (char *)0 )
name = RTMemoryUtil::strdup( name );
other_field = rhs.other_field];// assign other fileds
}
return *this;

Override default destructor:
Name: ~SomeClass
Code: delete [] name;
```

These requirements are caused by the way memory allocation for strings is handled in Rose RealTime.

Coregions

Place coregions around a set of messages to indicate that the events may occur in any order. Coregions can be used in a specification sequence diagram to handle race conditions or simply to indicate message flows where you do not want the message order verified. Coregions are usually placed on both participating interaction instances unless the order is not essential only for one instance.

Getting the Most Out of Your Test Results

To get the most out of your test results, you will want to narrow your test simulation down to the lowest functionality, by eliminating dependencies. This helps you to localize any problems found.

Capsule behavior can be quite complex, therefore, modeling the sent messages may not provide the level of detail required to localize a problem. For each message sent and received, data is passed that may be manipulated and created by the capsule.

A driver is a generated capsule based on a user specified interaction instance. Its behavior simulates the messages coming in and out of the interaction instance. A driver has no concept of this data manipulation from the sequence diagram message order. This is why you will want to take advantage of custom user code to simulate capsule behavior more accurately.

Using Driver and Stub Behavior to Simulate Your Test

You can specify user code and user attributes to simulate some basic behavior and help the drivers behave a certain way. This makes the simulation more accurate. For example, assume the following capsules exist in a sequence diagram:

- Capsule A, as the top capsule
- Capsule B
- Capsule C, as a sub-capsule of Capsule A

You can stub out Capsule C, allowing Capsule C to be a driver, to narrow the test down to include only Capsule A.

Integrating Capsules into a Sequence Diagram

In a sequence diagram with Capsules A, B, C, and D, you will want to test iteratively, by testing each component at a time and integrating it into the diagram. One scenario could be:

1. Unit test each capsule or capsule role. For more information on unit testing, see *Unit Testing a Capsule* on page 34.
2. If Capsules A, B, and C are ready, and D is not, you can stub out D to test the integration of A, B, and C. After you have unit tested A, you can add it to the sequence diagram and test it with D stubbed out. In this case, any messaging sent to D is ignored, but RQA-RT generates simulated behavior for D. This gives D behavior, without functionality, so that you can test the interaction of A in the diagram, unless you add user code to messages and/or local actions.

3 To integrate B and C into the diagram, perform unit testing for B and C. Integrate each into the diagram after you have unit tested. This enables you to test the interaction of A and B with D stubbed out, and then A, B, and C with D stubbed out.

To stub out a capsule, use the RQA-RT properties dialog from the context menu of the capsule you wish to stub out.

Integrating Capsule Roles into a Sequence Diagram

You can stub out capsule roles to test a capsule. This lets you test and integrate each capsule role into the capsule before you unit test the capsule.

Generating Behavior for a Driver

The RQA-RT Add-in lets you create a test harness environment to test the interaction between different capsule instances and then verify the results.

You can use the sequence diagrams to specify various states for each test. For example, to start a test from a clean state, insert a reset action in the appropriate place in the sequence diagram. All capsules involved in the previous test will be destroyed.

A driver can be an instance that has generated behavior, or an instance based on a capsule that is not fully implemented.

A driver with generated behavior can interact with the capsules under test (CUT's). To run a test, you need to inject drivers into the system; these drivers produce test results which you can verify after running the test.

If the driver is based on a capsule that is not fully implemented, you need to generate behavior for it in order to test the other instances in the interaction.

The drivers provide shell behavior for capsules that are not yet fully implemented. This lets you run tests using the driver to simulate externally triggered events.

Allowing Path Elements to be Drivers in a Containment Hierarchy

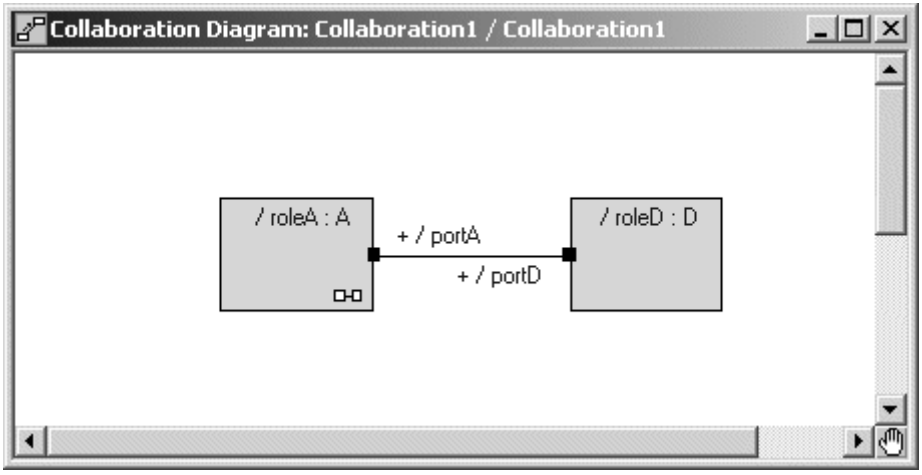
In a containment hierarchy you can allow interaction sequences in a sequence diagram to be drivers as long as they are leaf nodes. Leaf nodes are instances at the bottom of the containment hierarchy within the interaction.

Defining an interaction instance to be a driver/stub causes RQA-RT to generate a new capsule behavior for that instance representing the message flow on the interaction life line.

In the example in Figure 9, the capsule roles are as follows:

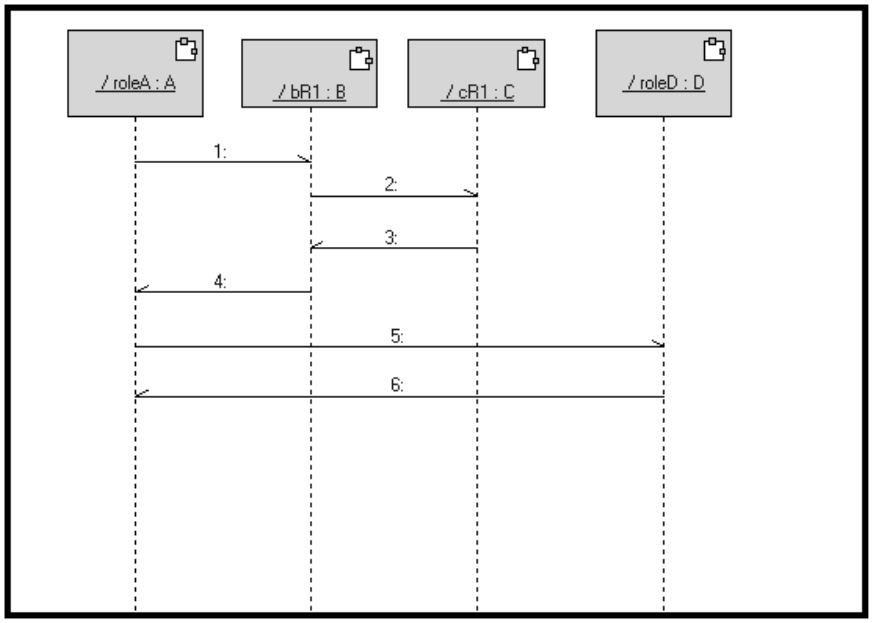
- A is a top level capsule in the collaboration
- B is a capsule role in A
- C is a capsule role in B
- D is a top level capsule in the collaboration

Figure 9 Collaboration diagram with containment hierarchy



Referring to the example in Figure 10, which represents the sequence diagram for the collaboration shown in Figure 9, only C and D can be stubbed out. To stub out B, you would have to either delete C from the specification or make C an unspecified instance. B would then become a leaf node.

Figure 10 Sequence diagram with containment hierarchy



Using Collaboration Diagrams to Capture the Test Environment

Use a Collaboration diagram to capture the test environment, rather than the structure. Place the Top capsule on the Collaboration diagram. From this top reference, RQA-RT can see the Top capsule and all contained capsules.

If you already have a Sequence diagram on the structure that you are trying to copy, then copy the structure into a Collaboration diagram. To do this, use CTRL key and drag the structure from the capsule into a package. This will duplicate the structure into a Collaboration including the sequence diagrams.

Note: You can also create a collaboration on the capsule; however, we recommend that you manage collaborations in packages.

After you create the collaboration, place the Top capsule on the Collaboration diagram. The Top capsule in a collaboration can be used with RQA-RT, and all its contained capsules are accessible in the Sequence diagrams and by RQA-RT.

Sequence Diagrams and Model Management

There are some performance, memory and model management advantages to using collaborations to capture test specifications rather than using capsule structures. Sequence diagrams can be very complex (and large) model elements. There may also be a large number of them in a model. By placing test cases on collaborations in

packages, it is easier to choose whether to load or not load the collaborations into a model. Old Sequence diagrams and collaborations that are no longer required do not need to be shared or added to the model to simplify model navigation. If the collaborations are not shared or added in, the model will load faster and the toolset memory footprint will be smaller. The file size for a controlled unit capsule will be smaller than with Sequence diagrams. Additionally, the code generator performance will also be improved because the code generator will be working with smaller files. Because capsule files will not have test specifications in them, they are not handled (and regenerated) whenever the Sequence diagrams are updated. Another benefit of placing the sequence diagrams on collaborations in packages is that there is less impact for those who share in these capsules because they do not get the Sequence diagrams, unless they also load the collaborations containing the sequence diagrams.

A containing capsule uses a protected port to talk to a contained Capsule Role. The port in the containing capsule is a public relay port: messages shown in the Sequence diagram in the example model flow from the relay port to a Capsule Role.

After you copy a Sequence diagram:

1. Add the Top capsule to the Collaboration diagram.
2. Open the Sequence diagram and change the path to be relative to the Top class you added.
3. Delete the unused roles from the Collaboration diagram.

Making the Container a Driver

If you want to make the containing capsule the driver, then the sub-capsules is not present at run-time. If your test specification references a sub-capsule, you cannot stub out its container. If you want to make the container a driver, then you need to remove the sub-capsules from the test specification.

Example

If you want to test a structure where container capsule **A** contains container capsule **B** (Figure 11), then the test specification would look like that in Figure 12. A problem will occur if you stub out **A** because **B** will no longer exist.

Figure 11 Capsule A Contains Capsule B

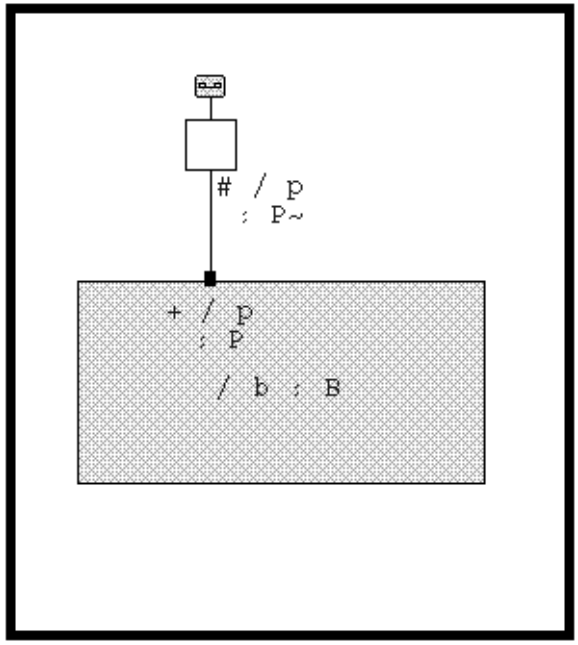
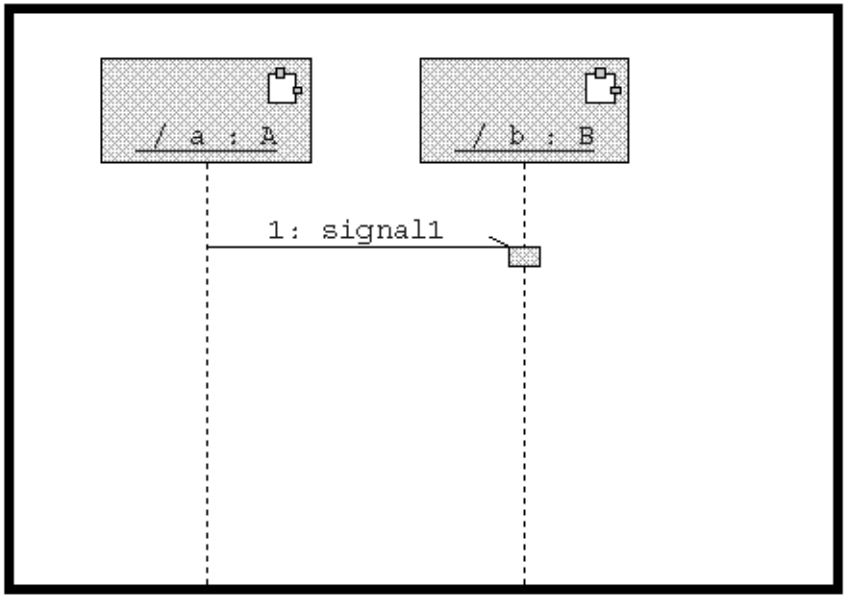
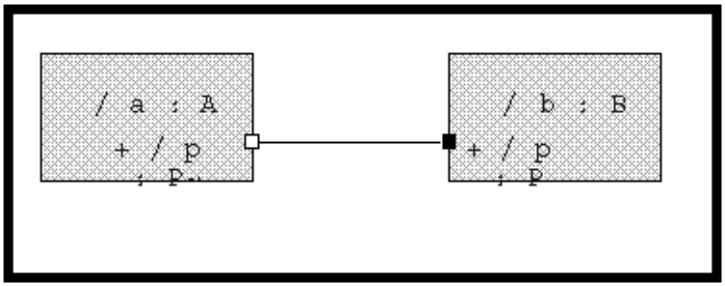


Figure 12 Sequence Diagram for A and B



To solve this problem, remove **B** from the test specification and then make **A** a driver. A problem continues to exist: How do you test **A** independently of **B**? **A** can only be tested with a **B**. You can change the structure to de-couple **A** and **B**; this makes the system easier to test. For example, Figure 13 shows this equivalent design.

Figure 13



A continues to communicates with **B**, and the implementation of **A** and **B** are the same. **A** can now be tested independently from **B**, and **B** independently from **A**. In addition, **A**, **B**, or both **A** and **B** can be stubbed out (manually or automatically). If you want to make the container a driver, then **A** and **B** can be part of the test specification. Occasionally, the container needs behavior. For example, to manage the life cycles of **A** and **B** in dynamic structures (incarnate/destroy), you would likely try to verify the implementation of the container and would not want to stub it out. You may want to stub out **A** and **B** to simplify the testing scenario.

Interaction Instance RQA-RT Properties

Attributes: You can specify attributes that are relevant to the test. For instance, a message may be received by the driver that has data which needs to be stored for future message processing. User code specified in a local action or message send / receive can access attributes defined here to simulate Capsule behavior.

Syntax: Specify only one attribute per line using the following format:

<*Name*> : <*Type*> [= *InitialValue*]

where:

<*Name*> is the name of the attribute.

<*Type*> is the attribute's type.

InitialValue is optional value.

Examples:

To use an attribute that is a pointer

Example:

```
pMyInt : int *
```

To use an attribute that is a user-defined class

Example:

```
pMyClass : MyClass
```

Observe the name of the file header generated for this class and include it into the **HeaderPreface** box if it is an interaction instance. For example, if you defined a user-defined class called MyClass, you would add the following to the **HeaderPreface** box:

```
#include <MyClass.h>
```

We recommend that you put a class into a package, then reference the package in a component that can be copied later. This ensure that the code for the class MyClass will be built.

To use an attribute that is an array

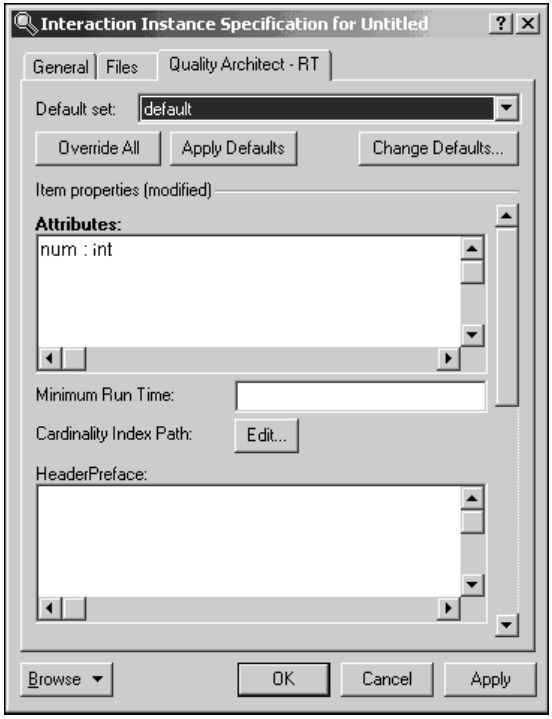
Example:

```
MyArrInt : int [2]
```

Minimum Run Time: Specifies the minimum amount of time that this instance should be active. This ensures that the instance will run long enough to send and receive all of the messages on its lifeline. When all of the interaction instances have exceeded their minimum run time and the last driver message has been sent or received, then the test will complete.

Note: The Interaction Instance Specification for Environment (for example, the "border" of a Sequence diagram) only has one functional field, **Minimum Run Time**. Setting this field sets the minimum runtime for the test as a whole. All other fields are ignored for the Environment instance.

Figure 14 Interaction Instance RQA-RT Properties Specification



Local Action

A local action is part of the actual sequence diagram specification. It shows up on the lifeline of the interaction instance as a rectangular box. By right clicking and choosing **Open Specification**, you can click on the RQA-RT tab to enter your user code. The user code is in the language of the model (C++ or Java) and will become part of the generated driver capsules state machine behavior. The user code for local actions is typically used to model explicit capsule behavior, for example, data manipulation / creation (see Figure 15).

This user-specified attributes property is typically modified in the local action user code.

Figure 15 Local Action RQA-RT Properties Specification

Send Message Sender / Receiver Code

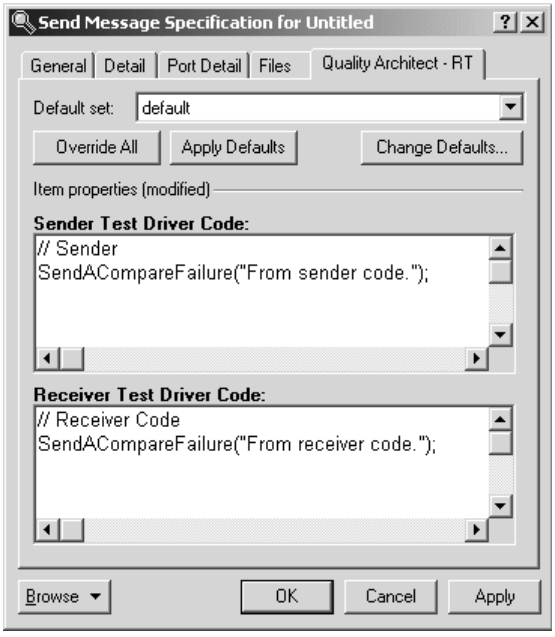
For more implicit driver behavior, for example, behavior for testing, you can use the message send and receive code. This behavior is not part of the specification sequence diagram. Messages sent by a driver can specify **Sender Test Driver Code** to do a custom comparison of signal data. As with the Local Action code, this is code in the same language of the model (C++ or Java).

A helper function **`SendACompareFailure`** allows you to send a custom message to the log that will show up as a difference after Verify Behavior has completed (see Figure 16). Similarly for messages received by a driver, you can specify **Receiver Test Driver Code**. This operates exactly the same as the **Sender Test Driver Code** except your verify message data being received.

If you need to access **`rtdata`** from this code, be sure to use explicit conversion to the data type desired:

```
<desired data type>rtdata
```

Figure 16 Send Message RQA-RT Properties Specification



Race Condition Analysis

A race condition occurs between pairs of events when these events appear in one order in the specification sequence diagram, but occur in either that order or an opposite order when the system is run. Race conditions pose a problem for the RQA-RT process since these arbitrary ordering of pairs of events can cause the trace sequence diagram to look different from the specification sequence diagram. These differences will generate error reports in the Rational Rose RealTime log pane.

The interpretation of causal relationships between sequence diagram messages helps you to understand the relationship between messages in the system. Race conditions exist when the temporal ordering of events cannot be inferred from a sequence diagram. For example, race condition analysis lets you know when the ordering of messages in a sequence diagram is ambiguous.

Ambiguous ordering can indicate alternate scenario execution paths to consider or can pinpoint subtle but potentially serious design problems. Race condition analysis locates ambiguous message sequences identifying race conditions. It is nearly impossible to locate these sequences by reviewing the code or sequence diagrams.

Three race condition analysis, in order of utility, are as follows:

- Find alternate execution paths in a sequence diagram
- Find design flaws where ordering of messages cannot be guaranteed by the design
- Find situations where sequence diagram message ordering cannot be predetermined, but also doesn't matter and a co-region can be used

Race condition analysis can be used on user defined sequence diagrams, trace sequence diagrams, and target observability traces. During the creation of a specification sequence diagram, the existence of race conditions may not be obvious.

To detect race conditions use the following procedure:

1. Open the specification diagram.
2. Right mouse click on the diagram to open the context menu.
3. Select **Race Conditions...**

When executed, this menu item displays a list of all of the pairs of messages in the specification sequence diagram which are in a race condition. The list items have a menu item that allows you to select (highlight) the pair of messages.

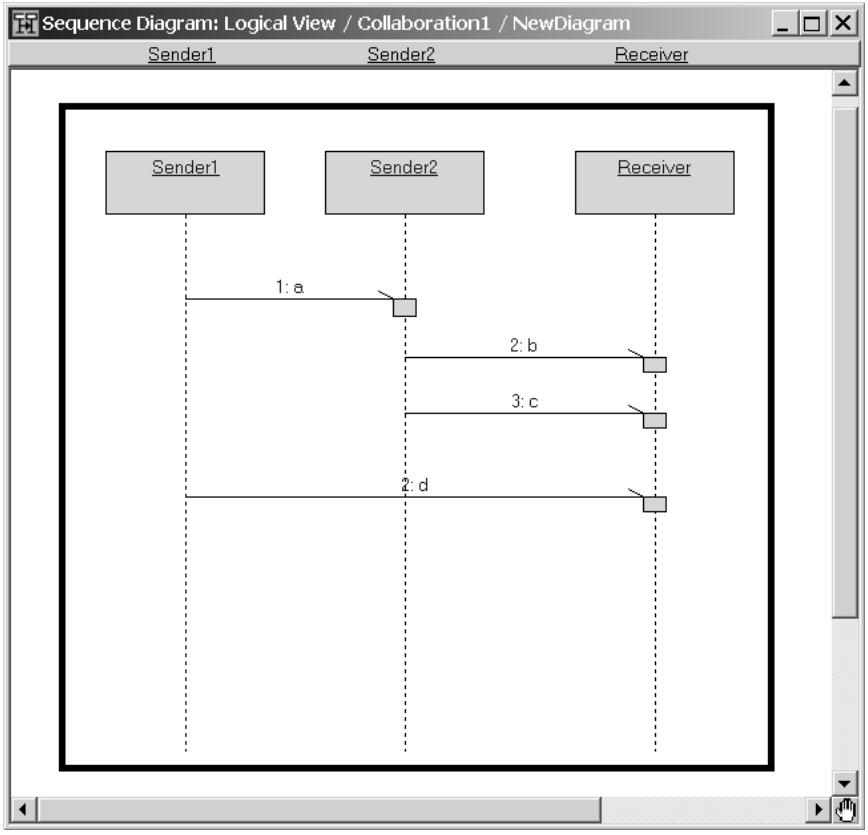
If pairs of messages are already within the same coregion, then no race condition is reported for those messages. If, however, the messages are in different coregions, then a race condition could exist between the pair, and it will be reported.

Race Condition Example

Figure 17 shows a simple sequence diagram with race conditions. In this sequence diagram, there are two race conditions: one between b and d, and the other between c and d. The sequence diagram shows one race condition, c and d, selected. To be valid, this sequence diagram must have coregions placed between b and d, and also between c and d.

Since coregions cannot be nested, the proper coregion placement in this sequence diagram would be to place a single coregion, on Instance2, just before message b and continuing to just after message d. This would allow the proper verification of a trace sequence diagram which had message d being received in its two other valid orderings (just before b, or between b and c).

Figure 17 Sequence Diagram with Race Conditions



Prior Messages and Subsequent Messages

In addition to being able to detect race conditions in a specification sequence diagram, it is also useful to identify the causal relationships between messages. Knowing causal relationships can help you understand message flow within a specification sequence diagram and help in the placement of coregions.

Causal relationships for a particular message can be classified in one of two ways:

- Prior Messages -messages received before the particular message was sent
- Subsequent Messages -messages sent after a particular message was received

In Figure 17, the Prior Messages of message c are a and b. The Subsequent Message of message b is c.

Verifying Specification Sequence Diagrams

5

Contents

This chapter is organized as follows:

Running a Verification

To use the RQA-RT functionality using one of the following methods:

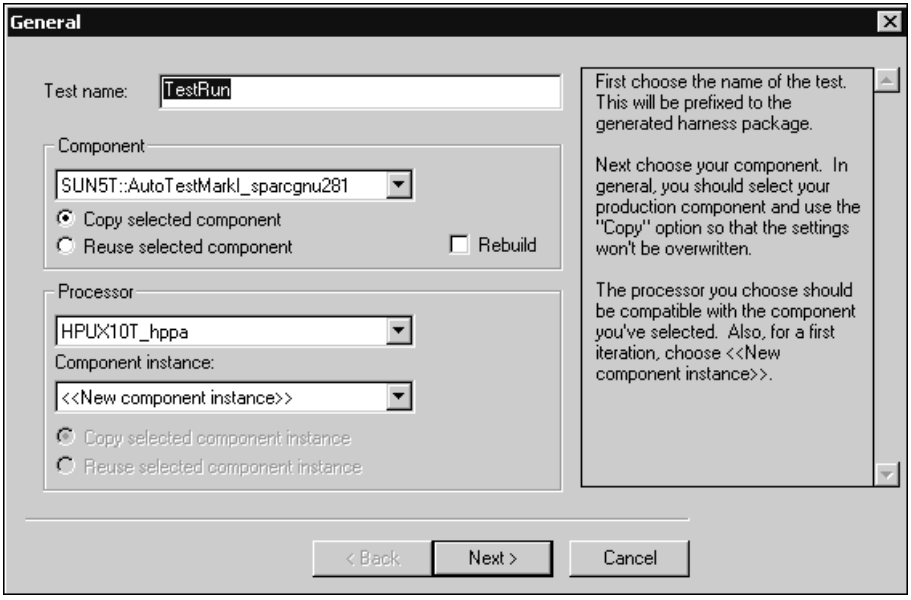
- From the Rational Rose RealTime menu, select **Tools - Rational Quality Architect - RealTime Edition**.
- To verify a sequence diagram from the **Model View** tab in the browser or directly from the diagram view, right-click and select **Verify Behavior**. The sequence diagram appears in the **Current Test order** list (see Figure 23).
- To verify multiple sequence diagrams that are owned by a Collaboration diagram, right-click on the collaboration diagram and select **Verify behavior**. All sequence diagrams owned by the Collaboration diagram appear in the **Current Test order** list.

After you select **Verify Behavior** for the first time, you are prompted (Figure 18) to launch a wizard to help you configure your test.

Figure 18 Selecting Verify Behavior for the First Time

Click **Yes** to launch the wizard, or **No** to open the **Verify Behavior** dialog.

Figure 19 RQART - RT Wizard - General Pane



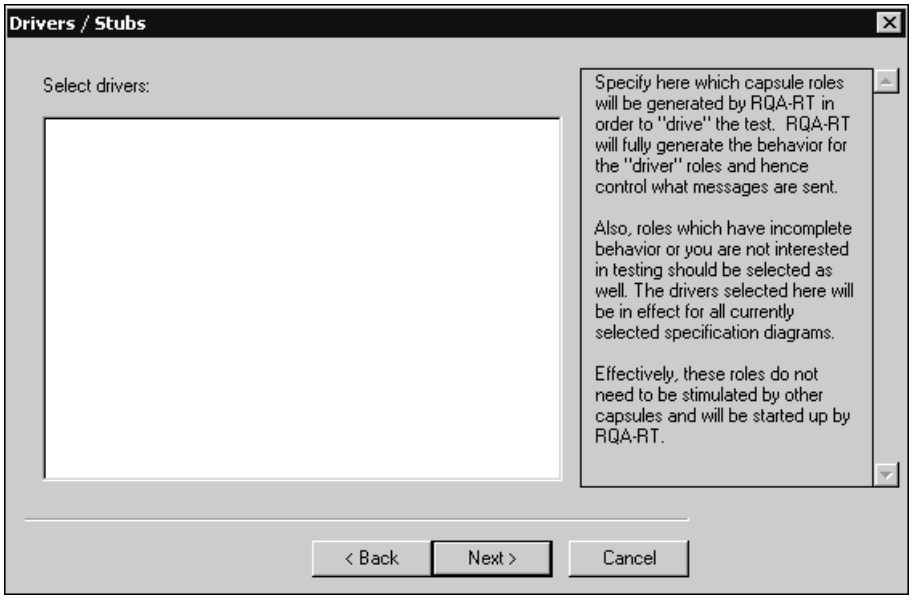
In the **Test name** box, specify a meaningful name for the test. This name will prefix the harness package that is generated.

In the **Component** area, select your desired production component. Select **Copy selected component** to copy the component, or **Reuse selected component** to override current settings. Select **Rebuild** to rebuild the selected component.

In the **Processor** area, select the processor from the drop-down list that is compatible with the component you selected in the **Component** area. For a first iteration, select **<<New component instance>>**.

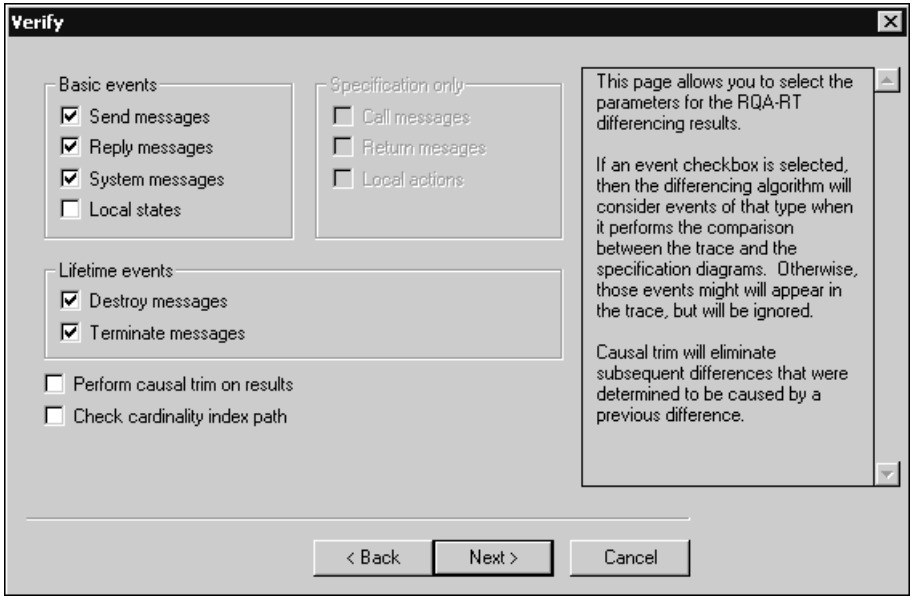
Click **Next**.

Figure 20 RQART - RT Wizard - Drivers/Stubs Pane



In the **Select drivers** area, select the capsule roles (that "drive" the test) for RQA - RT to generate the behavior to determine what messages are sent.

Figure 21 RQART - RT Wizard - Verify Pane



Use this pane to specify parameters for the RQA - RT differencing results.

Basic Events

In this area, you specify the types of events that the differencing will consider when it performs a comparison between the trace and the Specification diagrams. If you do no select any basic events, although those events may appear in the trace, they will be ignored.

Lifetime Events

In this area, you specify the types of events that the differencing will consider when it performs a comparison between the trace and the Specification diagrams. If you do no select any basic events, although those events may appear in the trace, they will be ignored.

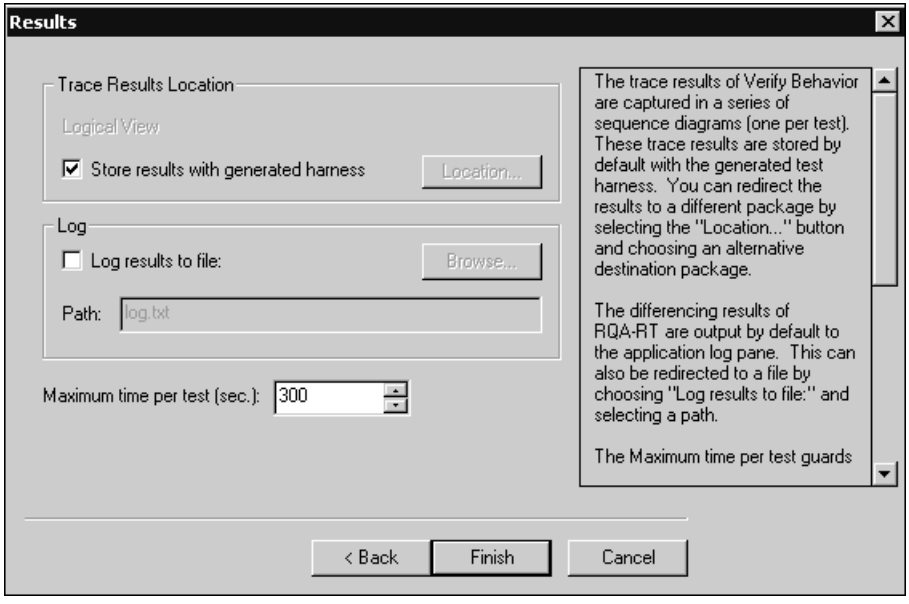
Perform casual trim on results

Where applicable, scale down the results.

Check cardinality index path

Verify the instance numbers for nested capsule roles.

Figure 22 RQART - RT Wizard - Results Pane



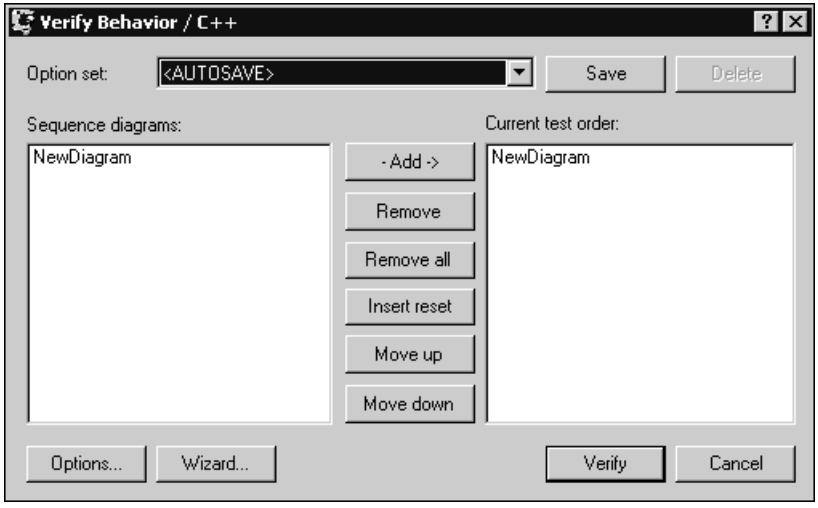
By default, the results appear in the Rational Rose RealTime **Output** window. In the Log area, select **Log results to file**, then specify a location in the **Path** box.

When tests to not end in the expected time, use the Maximum time per test (sec) box to specify the length of time for the test, in seconds.

Click **Finish** to open the **Verify Behavior** dialog, where you can select the Sequence diagrams to include in the test.

Note: You can modify the options you selected in the wizard by clicking **Options** on the **Verify Behavior** dialog.

Figure 23 Verify Behavior dialog



Verifying Multiple Specification Sequence Diagrams

Multiple Specification sequence diagrams can be verified at once. Choose the desired specification diagrams from the left-hand pane of the **Verify Behavior** dialog. (see Figure 23). Click the **Add** button to move them to the test order pane. Change the order of the tests by selecting specific diagrams and clicking the **Move up** and **Move down** buttons.

The Specification sequence diagrams are verified in the order in which they appear in the right hand pane in the **Verify Behavior** dialog. In order to *reset* the instances in between tests, click the **Insert Reset** button. A reset will essentially destroy all instances from the previous test and then recreate them in the subsequent test.

Running Verification on a Capsule Subclass

Sequence diagrams are not inherited by a capsule sub-class in the browser.

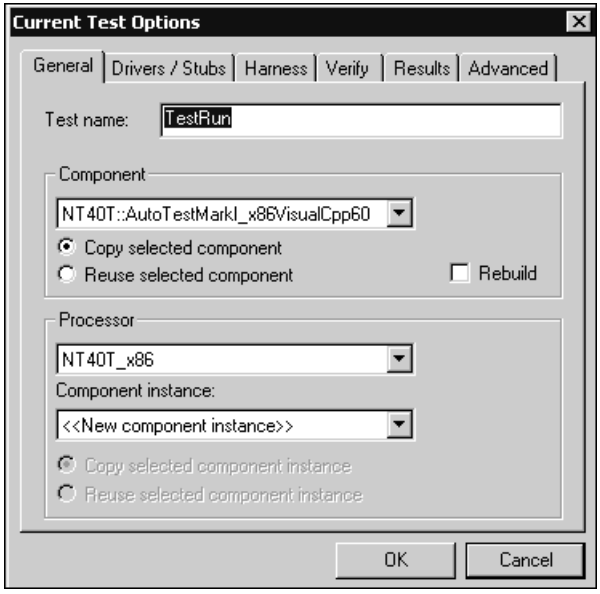
To use the sequence diagram of the super capsule in the subclass:

1. In the **Model View** tab in the browser, select the sequence diagram in the superclass.
2. Hold the control key down and drag the diagram onto the sub class capsule in the browser. Observe that a duplicated version of the diagram now exists in the browser under the subclass.
3. Select the duplicated sequence diagram.
4. To run a verification, follow the steps in *Running a Verification* on page 59 using the duplicate sequence diagram as the specification diagram.

RQA-RT Options Dialog

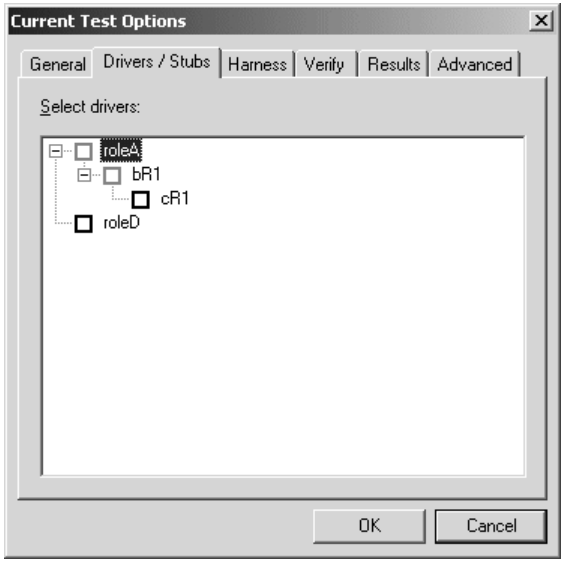
To configure specific settings or options for the test run, you must set the options by clicking **Options** on the **Verify Behavior** dialog. All the options for the test are set to default values. The **Options** dialog (Figure 24) allows you to select parameters that govern the verification run for the selected Specification sequence diagrams.

Figure 24 Options Dialog - General Tab



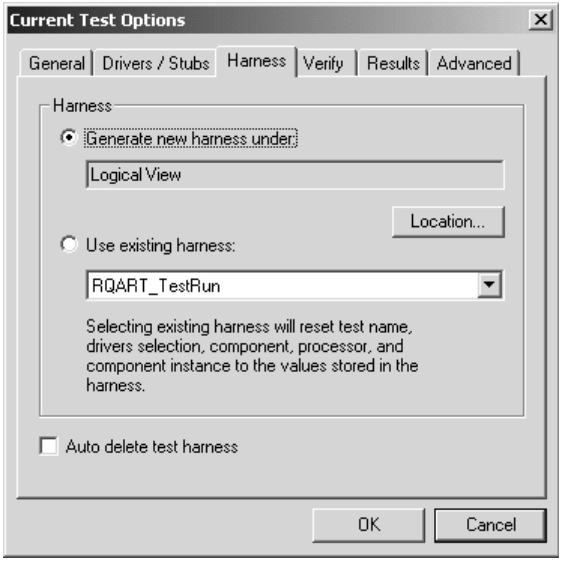
- **Test name** - This is the name given to the test run. It is added as a suffix to the test harness package that is generated.
- **Component** - This allows you to pick a specific component to run the test on. It is necessary for you to set some of the properties of the component, such as the target configuration. This is important so that RQA-RT can run tests on the same target as the production system. Or inversely for simulation on a local target, such as the same environment Rose RT is running on (such as Windows NT/2000, Solaris, and so on).
- **Rebuild** - When this option is selected, a full re-build is performed, and when it is not selected, an incremental build is performed. We recommend that you do not set this option. Only set this option to clear out build problems caused by manual file editing, or same-name-files being overwritten. For RQA-RT automation, use **ReBuildCode**. It's type is boolean where True means re-build, and False means incremental build (default).
- **Processor** - Allows user to choose the processor that the component will run on.
- **Component instance** - You can choose an existing component instance or generate a new one based on an existing component instance. The chosen instance must be the same as the existing component instance.

Figure 25 Options Dialog - Drivers Tab



- **Drivers** - This pane lists all the interaction instances in the sequence diagram. The box beside each name, if checked, indicates that RQA-RT is to generate a driver / stub in place of the real interaction instance. Names of interaction instances with unspecified roles are dimmed. The box beside a dimmed name is always checked, since interaction instances with unspecified roles must have drivers.

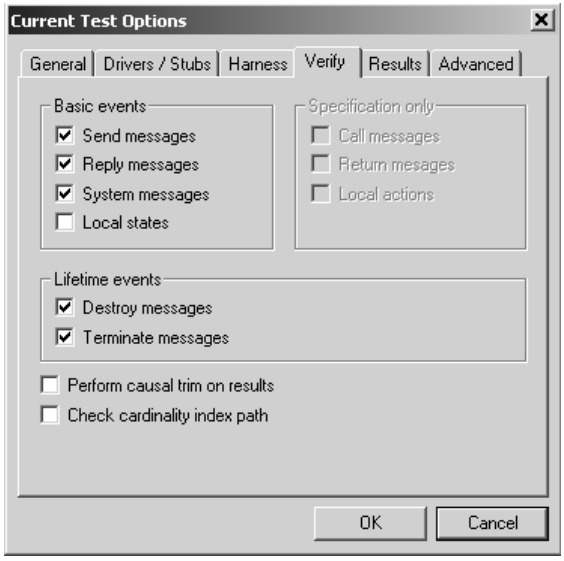
Figure 26 Options Dialog - Harness Tab



- **Generate new harness** - When checked, RQA-RT will generate a new test harness upon beginning the test that will be compiled and run on the target. The location button allows you to specify a specific location to place the generated test harness. This is useful if the destination for the test harness package is inside a scratch pad package. Since the test harness package is not really part of the production model being tested, it is practical to place it inside a scratch pad so that there are no source control implications.
- **Use existing harness** - For performance reasons, it may be more practical to use a previously generated harness instead of regenerating it each time. This ensures the harness is not regenerated and compiled. This is only a viable solution as long as no changes have been made to any of the sequence diagrams being verified in-between tests. The test harness is generated based on the message passing and structure of the sequence diagrams. If changes have been made to the CUT's that don't effect the sequence diagram, i.e. change in behavior / internal structure, then a previous test harness may be reused.

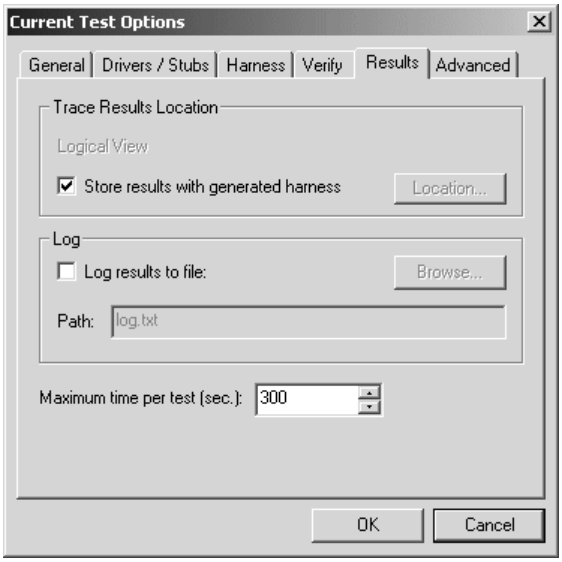
- **Auto delete test harness** - When checked, this option will automatically delete the generated test harness after the test run is completed.

Figure 27 Options Dialog - Verify Tab



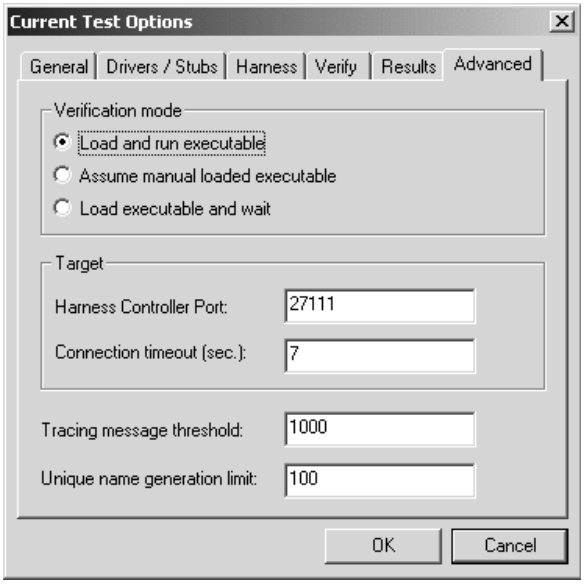
- **Basic events** - Check the box beside these messages to include them in the differencing algorithm comparison.
- **Specification only** - These messages are not supported by the regular Verify Behavior process i.e. not generated on the trace and may appear on the specification diagram only. These check box options are only accessible through the stand-alone differencing command where two specification diagrams may be compared.
- **Lifetime events** - Check the boxes to include comparison of destroy and terminate messages in the differencing algorithm.
- Perform causal trim on results - This option will filter the differencing algorithm results. If a difference causes another difference to occur later on, this can make it unclear what the original problem was. By using the causal trim, only the original difference will appear and differencing caused by previous differences will be filtered out.
- **Check cardinality index path** - Used with specification diagrams containing replicated capsule instances. Values specified in the **Edit Cardinalities** dialog will be used to establish correspondence between interaction instances from the specification diagram and the trace diagram.

Figure 28 Options Dialog - Results Tab



- **Maximum time per test (sec.)** - This field is a safeguard to make sure that the test run will eventually complete. If a test has not completed in the time specified, i.e. a capsule under test doesn't respond to stimulation, the RQA-RT Add-In stops the test and either moves on to another test, or closes the RTS and displays the results of the verification.
- **Log results to file** - Normally results are logged to the output pane in Rose RealTime. For persistent results and / or results useful from overnight sanity tests, it is possible to log results to a file. The specific file path can be chosen or browsed to.

Figure 29 Options Dialog - Advanced Tab

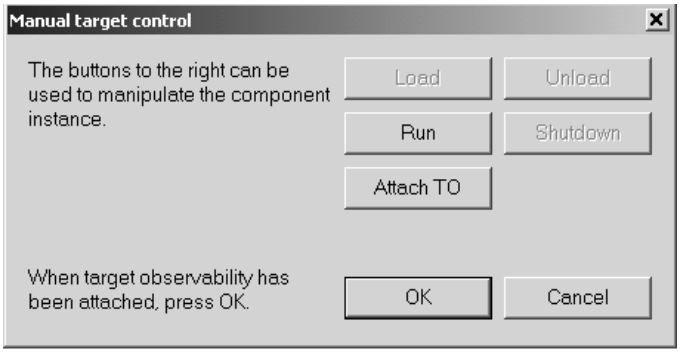


The Mode: radio buttons control access to the RTS. The modes are as follows:

- **Load and run executable** - This is the default Mode option since it provides for a hands-off execution of the verification tool. The RTS is loaded, run, and shut-down automatically.
- **Assume Manual Loaded Executable** - This option is for use with targets that do not support the automatic download and execution of models from the toolset workstation. The option provides a pause in the RQA-RT verification process allowing you to manually load the complied model onto the target outside of the toolset.

  If you select this option, the Manual Target Control dialog appears, as shown in Figure 30. From the Manual Control dialog, you can Load, Attach, and Run a component instance manually.

Figure 30 Manual Target Control



- **Load executable and wait** -This option is enabled only when compiling for the TargetRTS with Target Observability enabled. Selection of this mode requires that you run the model in order for the verification to be completed.

The following fields are for the specific target being tested:

- **Harness Controller Port** - This is a unique identifier used to create the communication socket between RQA-RT and the target executable. If RQA-RT is having trouble communicating with the target, try adjusting this value to something different.
- **Connection timeout (sec.)** - Specifies the maximum allowed wait time for communication to be established with a target. If communication cannot be established within the given time, the target will be reset. The default is 7 seconds.

The remaining options are limit fields that you should not need to set except in rare circumstances:

- **Tracing message threshold** - Sets the tracing level for a Verify Behavior test run.

  Default: 1000 messages. More complex models may require this to be increased if a lot of message passing occurs within the interaction.
- **Unique name generation limit** - Sets the number of generated test harnesses that can be stored within a model.

Verification Run Results

1. If you select the **Load** mode, manually step though or run the models until all tests are completed and the RTS automatically closes.

   OR . . .

   If you select the **Load** for manual target mode, the RQA-RT process is as follows:

   - Select **Verify** in the RQA-RT Verify Behavior dialog. The **Manual Mode** dialog appears. You can **Load**, **Attach**, and **Run** a component instance manually.
   - After the model loads, click **OK** and the toolset will load the RTS.
   - If the toolset cannot establish a connection with the target platform (referenced configuration), an error dialog will display.
   - After the RTS loads, you must select **Run** for the RQA-RT process to continue automatically.

2. If you select the **Run** mode, RQA-RT automatically opens the RTS, runs the models in the RQA-RT framework, and then closes the RTS.

After the RTS closes, the results of these comparisons are appear in the Rational Rose RealTime **Log** tab in the **Output** window.

Verify Behavior Results

The Summary diagrams generated contain the number of passed or failed tests, as well as a list of the passed or failed Sequence diagrams. The resulting Sequence diagrams appear in the **Model View** tab in the browser.

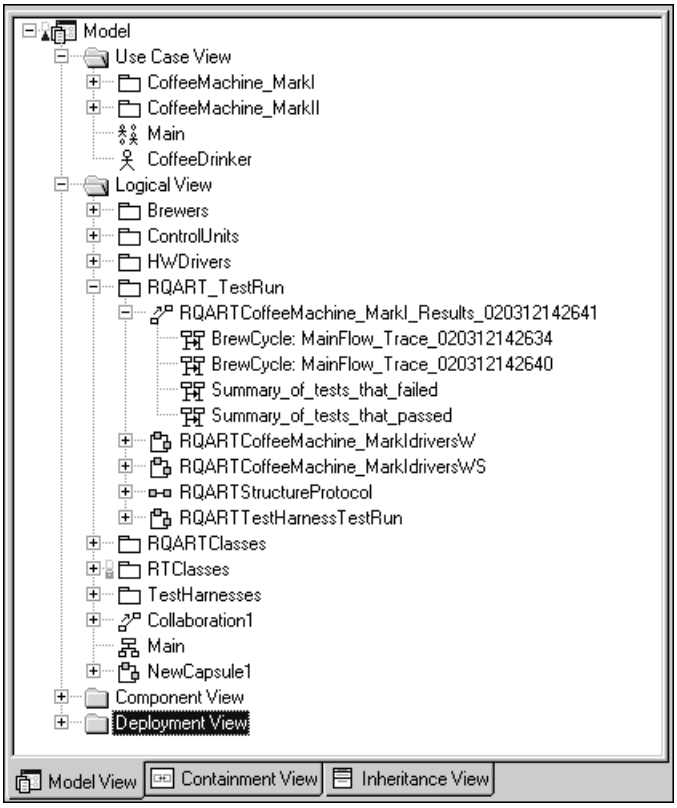
Note: These Sequence diagrams are populated only when you select a Sequence diagram and open it, or if you double-click on one of the notes in a results diagram.

The new Sequence and Summary diagrams are contained within the results collaboration. The results will be in the **Logical View** package in the **Model View** tab under the package named:

RQART_+ <*test_name*>

where **<*test_name*>** is the name that you specified for the test. Figure 31 shows the Summary and Trace Sequence diagrams in the **Logical View** package.

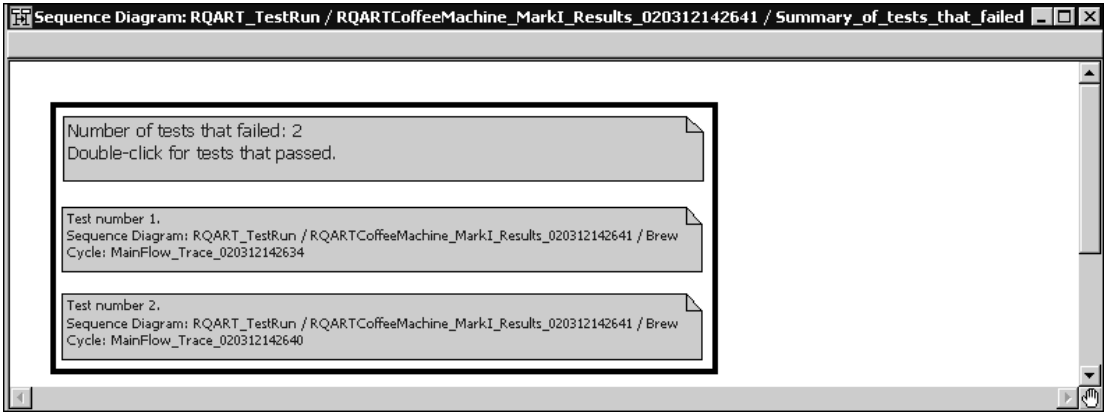
Figure 31 Results on the Model View tab in the Browser



Summary Sequence Diagrams

Figure 32 shows the results for tests that failed and Figure 33 shows the results for tests that passed. The Results diagrams are hyperlinked to each other. Double-clicking on a note in the **Passed** results diagram opens the **Failed** results diagram, and vice versa.

Figure 32 Results that Failed

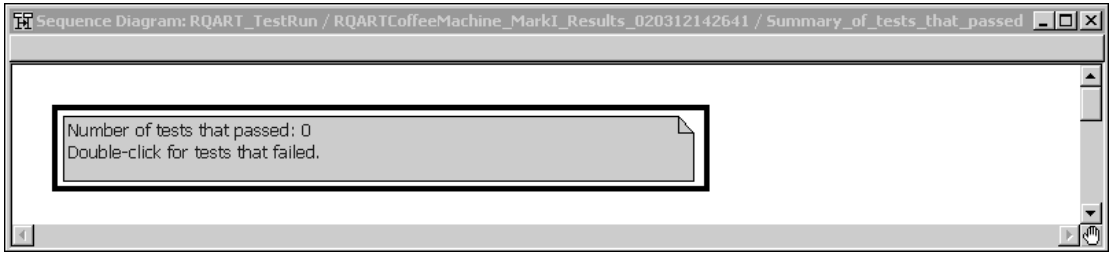


The newly created Summary Sequence diagrams contain a date and time stamp. The format for this stamp is:

YYMMDDHHMMSS

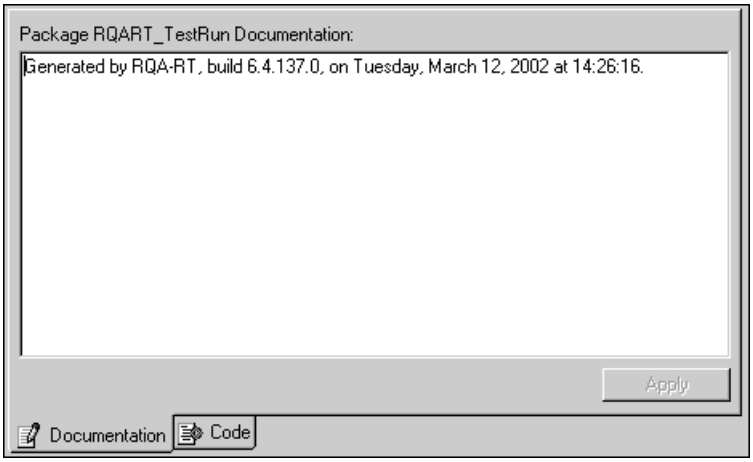
where **YY** is the year, **MM** the month, **DD** the day, **HH** the hour, **MM** the minutes, and **SS** the seconds.

Figure 33 Results that Passed



In the **Documentation** tab (Figure 34) you will see the date and time stamp and the RQA-RT version number for the test.

Figure 34 Documentation tab



Trace Sequence Diagrams

The title of the window for the newly created diagrams (Figure 35) have the following format:

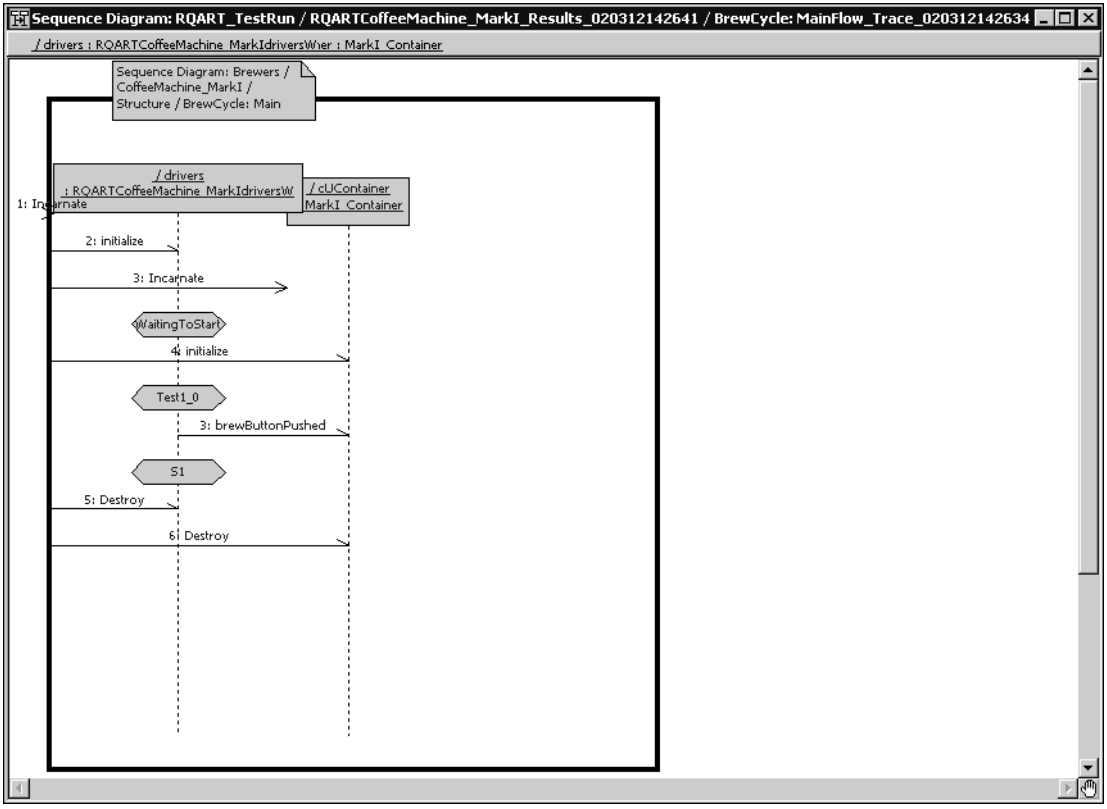
<sequence_diagram>*_Trace_*<date and time stamp>

where ***<sequence_diagram>*** is the name of the sequence diagram being tested, and ***<date and time stamp>*** is the date and time in the format YYMMDDHHMMSS.

For example, BrewCycle:MainFlow_Trace_020308155408

If you select a Trace Sequence diagram, the name appears in the **Documentation** tab (Figure 34), as well as the date/time stamp and RQA-RT version number.

Figure 35 Trace Sequence Diagram



Note: These Sequence diagrams are populated only when you select a Sequence diagram and open it, or if you double-click on one of the notes in a results diagram.

Models Under Source Control

The Sequence diagrams are automatically populated with views when the Sequence diagram is opened unless you clicked **No** to a Source Control prompt, or the model is read only.

Typically, opening a sequence diagram is not considered an operation that would change the model files. However, if the Sequence diagram is unpopulated, an attempt to populate the model causes the model files to change, unless it is prevented from continuing.

Note: If the model is not under source control and is not read only, or if it is under source control but the appropriate unit is already checked out, then the population will proceed silently but the unit will be marked as modified and you will be prompted to save the model upon closing.

Inspecting Rational Quality Architect - RealTime (RQA-RT) Results

6

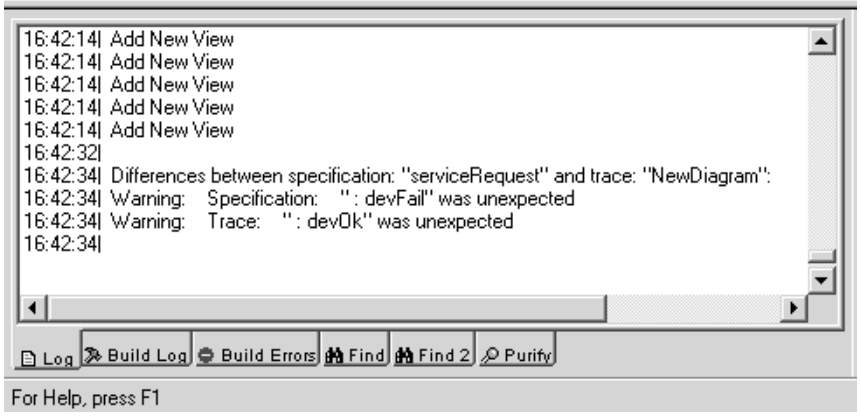
Contents

This chapter is organized as follows:

Overview

Upon completion of verification, the results of the differencing algorithm comparison are displayed in the **Log** tab in the Rational Rose RealTime **Output** window. Each line in the browser list represents an error result related to a specification / trace pair of Sequence diagrams. Each line entry shows: the diagram in which the inconsistency was found (specification diagram or trace diagram), and the name of the message in question (see Figure 36).

Figure 36 Rational Rose RealTime Log tab in the Output Window



If no differences are apparent and the specification successfully matches the trace diagram, then the following message will appear:

Differences between specification: <spec. diagram> and trace: <trace diagram>:None

If differences are found, they will appear as in Figure 36. You can double-click on any warnings or errors that appear in the **Log** tab in the **Output** window as a result of the RQA-RT Verify Behavior Comparison to navigate to the specific model element in the sequence diagram (either specification or trace) which is causing the difference to appear. The Sequence diagram or trace diagram is automatically opened, and the message that is causing the differencing warning is selected and centered in the diagram.

For unsuccessful matches, the dialog box shows one of the following results:

- Warning: <*Specification or Trace*> <*Message Name*> was unexpected
- Warning: no matching instance found for instance <*Instance Name*>
- <*Message Name*> failed to match with custom comparison message: <*User entered string*>

Comparison Rules

In the RQA-RT process, the specification sequence diagrams and trace sequence diagrams are not compared for equality, but rather for *similarity*. The governing rules are as follows:

- **Instances** - a capsule reference is only traced, that is, appears in the trace sequence diagram, if its position in the system hierarchy matches that described in the reference path of a specification sequence diagram. Even though the CUT(s) may have many other component capsules, the verification process ignores these capsules as well as messages to and from them. They will not appear in the trace sequence diagram. This permits the specification sequence diagram to be a blackbox description of the system.

  Unnamed interaction instances are always included in a trace.
- **Event Order** - the mismatched events from the specification sequence diagram will be shown followed by the mismatched events from the trace diagram. You can apply causal dependency elimination through the **Verify Options** tab.
- **Coregions** - if a set of messages is specified within a coregion on the specification sequence diagram, then these events are allowed to occur in any order in the trace sequence diagram.

- **Local States** - if present in the specification sequence diagram, states must be present in the trace diagram having matching names. You can specify whether or not Local States are included in the comparison through the **Verify Options** tab.
- **Local Actions** - actions in the specification sequence diagram are ignored in the Verify Behavior process. They can be compared through the stand-alone differencing functionality if you specify this as an option.
- **Messages** - For any message pair to match, they must satisfy *all* of the following matching criteria:
  - order
  - source instances
  - destination instances
  - port names - the name *empty port* name, if used, must match the name of one other port name, as generated port names may not necessarily be present in the spec diagram, but will be present in the sequence diagram
  - signal name
  - synchronous versus asynchronous mode
- **Mechanism** - identifies a matching interaction instance in trace and spec diagrams. For interaction instances with unspecified roles, follow the name generation rules to determine which interaction instances are a match.

Note: Message data is not compared by default. This is left up to you through the custom data comparison mechanism.

System Ports

System Ports / Unwired ports, also known as SAPs, include the Frame, Log, Exception, and Timer.

Note: Not all of these SAPs can send or receive messages.

The messages travelling through the basic system SAPs, that are provided by the Target Services Library, are not under the control of the test harness. The messages for these services are traced, but it is your responsibility to correctly code the model to ensure that the messages are sent or received.

For example, the test harness does not inject time-out messages. This is done by the Target Services Library after a timer is correctly set up.

Note: Because these messages do not involve the test harness, they are not considered when trying to figure out when a test has completed.

Framework Components

In general, the framework components that support the RQA-RT Add-In are transparent to the user. However, some advanced users may wish to customize some of the components in order to suit their own specific testing requirements. It should be noted, that modifications to the framework may impair the RQA-RT Add-In's functionality.

To avoid possible naming conflicts between your model and elements generated by RQA-RT we recommend that you avoid names beginning with RQART, since RQA-RT prefixes all generated names with this string.

The framework components and their function are as follows:

- **Driver capsules** - the RQA-RT Add-In uses the specified driver Capsules and a set of specification sequence diagram(s) to create a set of driver capsules. Each driver capsule implements the externally visible messages of its matching specification sequence diagram. In the model browser there is a package called RQA-RT Classes that contains the abstract capsule class that the driver capsules are derived from. These capsules are contained a couple of levels deep in the package to account for language and version number of the shared package.
  - **RQARTAbstractTestDriver** - concrete capsule class drivers are derived from this capsule
  - **RQARTTestDriverProtocol** - protocol class that simulates the capsule interface.
- **Harness Capsule** - the capsule within which all the driver capsules and CUT are executed. The structure contains the CUT(s) roles and roles of all the drivers. When loaded and executed in the RTS, the harness capsule automatically incarnates each of the driver capsules in succession and reports on their success or failure. The following capsule classes are added within the RQA-RT Classes package for implementing the harness:
  - **RQARTAbstractTestHarness** - concrete capsule class harness is derived from this capsule. This provides all the necessary infrastructure and utilities required by a concrete harness capsule class.
  - **RQARTCPServer** - capsule used for communicating with the RQA-RT Add-In.

  To clean your model of generated test harnesses, RQA-RT packages, and associated artifacts, select **Rational Quality Architect - RealTime Edition - Remove RQA-RT Artifacts From the Model**.

Troubleshooting

One common problem occurs when the port or signal names in a message are specified incorrectly. You will have to correct these errors and rerun the verification. Typically, RQA-RT provides feedback in the application log if it is unable to generate a harness from the given sequence diagram(s).

Standalone Differencing

The differencing algorithm lets you compare two sequence diagrams of your choice. Since sequence diagrams can be generated from previous traces, you can use standalone differencing to verify the trace from an actual production run of a system, as opposed to traces generated from the test-harness.

Sequence Diagram Differencing

You can compare previous runs and actual production traces with your specification diagrams and you can specify more precise filtering for the differencing to get a more accurate picture of the differences.

To use sequence diagram differencing:

1. Select the specification sequence diagram. A list of packages containing sequence diagrams appears on the left, and all sequence diagrams within the selected package appear on the right.
2. Select the trace sequence diagram to compare.
3. From the differencing options tab, select different filtering options. Filter out invalid messages from the interaction diagram.
4. The differencing results appear on the Real RealTime log pane. You can link the results to the model elements by double clicking.

Verifying a Trace

This feature is a variation of the stand-alone differencing functionality; it uses an output trace as the trace component. Instead of comparing two sequence diagrams, this compares a trace to a sequence diagram as long as target observability is active. The trace is converted to a temporary interaction and then compared with the specification sequence diagram of your choice.

This feature is only relevant when target observability is active and a **Trace** window is open.

Troubleshooting and Known Issues for RQA-RT

The following information explains the troubleshooting and limitation of Rational Quality Architect - RealTime Edition.

Driver Methods for Sending Messages to the Log and Custom Comparison

RQA-RT includes two helper functions in RQA-RTAbstractTestWrapper, the capsule that is the superclass for all generated drivers. These functions are:

```
SendACompareFailure(<some_string>);
```

```
LogAMessage(<some_string>);
```

Both of these functions send a message to the RoseRT log that is hyperlinked to the appropriate message in the trace Sequence diagram. SendACompareFailure also causes the sequence diagram differencing algorithm to fail on that message. These functions can be called in any user-specified code in a Sequence diagram. This includes:

- Local action code blocks from the "Quality Architect - RT" tab of the Local Action Specification.
- The "Sender Driver Test Code" and "Receiver Driver Test Code" code blocks available from the "Quality Architect - RT" tab of the Send Message Specification".

This code will be inserted on the appropriate transition in the generated driver capsule. If you want to see exactly where it is inserted, use the **Find In** feature to search for some identifiable piece of the code (possibly a unique string in a comment) in the generated driver capsule.

Lost Information in *To Port* for a Message

If you load an LF-file into Rational Rose RealTime and perform the conversion, there are a number of instances where the information on the **To Port** of a message is lost. In Rational Rose RealTime 2002.05.00, the conversion was enhanced to determine the receiver port on any message sent between two instances that have a direct logical connection from the sender port. This calculation increases the time required to perform the conversion. Diagrams which have messages between instances that skip relay ports will continue to require the receiver port to be entered manually.

Do not use -runScriptAndQuit when running RQA-RT from a script

When running RQA-RT from a script using the RunVerifyBehavior method do not use the -runScriptAndQuit command line option to cause RoseRT to exit once the script is complete. Since Verify Behavior runs asynchronously after RunVerifyBehavior is

called this will cause RoseRT to exit before Verify Behavior is complete. Use the szScriptOnCompletion parameter of RunVerifyBehavior to specify a script that contains the actions that you want to occur after Verify Behavior is complete. This script can exit RoseRT using the Exit method.

Creation of Container Capsules

RQA-RT does not automatically create container capsules for a nested capsule when the container is not included in the Sequence Diagram.

Converting MSCs in Rational Rose RealTime using the RQA-RT

Problem

Many MSCs have a variable of the same name (prepareSetupReqD), but they can have a different type. When RQA-RT synthesizes these attributes from the multiple driver instances and attempts to generate one test driver, there is a name conflict and it selects the last Sequence Diagram attribute's type as the type for the attribute of the test driver class.

Background

In ObjecTime Developer, attributes are a characteristic of MSCs; each MSC can have its own attributes. These attributes can be considered variables for the environment which acts as a driver in TestScope.

When the MSCs are converted in Rational Rose RealTime using the RQA-RT conversion tool, each MSC is converted to a Sequence Diagram. In this Sequence Diagram, what was the environment in ObjecTime Developer is now a driver capsule that is automatically created. The interaction instance (of the driver) in the sequence diagram has attributes which were converted from the MSC in ObjecTime Developer.

This newly created interaction instance is set as a driver during the **Verify Behavior** operation. As a result, a new RQADriver is created.

Example:

Seq1:: driver has an attribute xAttrib of type Xtype

Seq2:: driver has an attribute xAttrib of type Ytype

Test harness generates a driver class with an attribute xAttrib of type Xtype or Ytype. The compilation occurs and the result will be many errors in the test harness. The errors occur because some of the action code interprets xAttrib as type Xtype and some as type Ytype.

Workaround

Run the Sequence Diagrams from separate test harnesses. This means that you will have completely separate components and will have to compile again.

Creating Messages and Sequence Diagrams

The Item Properties in the Create Message Specification dialog, with the exception of Thread, only apply if you are creating a Capsule Under Test (CUT). Thread applies in every case. The Item Properties entries are as follows:

- Capsule class - a capsule name. Enter a value here only if you want to override the default capsule class associated with the role that is associated with the interaction instance being created.
- Initial data - Enter a value here only if your Capsule Under Test (CUT) requires data on startup. To provide data you must specify an attribute in the driver sending the create message with the appropriate initial values. You cannot provide initial data in create messages sent from the environment.
- Data Descriptor - if you provided initial data, you must specify the type descriptor of the type of data that you specified. This is in the format **RTType**_*<type of initial data>*
- Thread name

Sending Message Specification Data Field Format for Java

- For the data to be passed with the messages, the data type should be specified as *<data_type> <constructor_arguments>*. For example, to pass Integer object referring to the number 5, place the following:
- Integer 5 or java.lang.Integer 5
- If *<data_type>* is omitted, Integer is assumed.
- The *<constructor_arguments>* should contain the exact line to be passed as an argument to constructor. For example, for the MyObject(*Integer, String*) the following line can be used: MyObject 5, "Acme"

  Note: No additional brackets or quotes need to be placed around *<constructor_arguments>*.

Limitations

- RQA-RT does not support C models.
- Only leaf node instances in the interaction/sequence diagram can be specified as a driver/stub.
- Within a specification sequence diagram, each interaction instance without a specified role must have a unique name. Only one interaction instance per test set can be left unnamed.
- When running multiple specifications in a test, a port on a capsule under test (CUT) can only be connected to one interaction instance with an unspecified role for the set of specifications.
- An interaction instance with an unspecified role cannot have a cardinality index. Care should be exercised using unspecified roles to test capsules and/or ports with cardinality greater than 1.
- If the generated trace sequence diagram is manually compared to the specification sequence diagram, it is important to remember that such comparison is asymmetric - the specification sequence diagram should always be selected before the trace diagram.
- When running multiple specifications in a test, remember that the capsules under test continue to execute between tests. Ensure that each sequence diagram becomes quiescent at the end of each test scenario.
- If you generate a new harness into a controlled package which isn't checked out warnings will be generated.
- In Java, replicated sub-capsules can only be specified as drivers if the replication index is 1.
- Interaction instances with unspecified roles cannot be used to simulate interactions with sub-capsules. An error will occur if an unsupported use of "unnamed" interaction instance is detected.

  Workaround: For example: top capsule A has capsule role B in it's structure. Capsule B has capsule role C in it's structure. If you run a test on the Sequence Diagram located under A's Structure Diagram, "unnamed" interaction instances can communicate with the B, but not C. To use "unnamed" interaction instance with the C, a Sequence Diagram should be located under B's Structure Diagram.

  Note: "unnamed" interaction instances are not a mandatory part of the verification process.

- If a capsule role has a cardinality greater than 1, the cardinality index should be specified on interaction instances.

  RQA-RT support of unspecified interaction instances allows you to easily drive an unconnected port without having to add a driver capsule to the collaboration. More complex tests will require the use of a driver capsule. Regardless, you cannot drive a test using a port that is part of the system currently being tested.

RQA-RT Batch Mode Introduction

7

Contents

This chapter is organized as follows:

Introduction to Batch Mode

One of the more powerful aspects of RQA-RT is in its use for sanity testing of your model/project. Through the use of batch mode, scripts can be constructed that will run the RQA-RT Verify Behavior through a fully automated process. Overnight sanity runs can verify that the current model passes the specification diagram requirements.

Any option / test variation that is accessible through the Verify Behavior dialog can also be accessed through the RQA-RT interface. The interface is COM based so any scripting language that supports COM can be used to run RQA-RT. Rational Rose RealTime ships with a Summit Basic Editor and interpreter that can run RQA-RT scripts, as shown in *Example using Summit Basic Script* on page 91.

Below is a list of the RQA-RT properties and methods that can be externally accessed.

Properties

AutoDelete [type: Boolean] - When set to True the generated test harness is automatically deleted after all of the test runs have completed. If this is set to False, then the generated test harness package will remain as an artifact. This is set to False by default.

Collaboration [type: LPDISPATCH] - Sets the collaboration under which the tests are run. Verify Behavior runs can verify any or all sequence diagrams under a specific collaboration during a test run.

Component [type: LPDISPATCH] - Sets the component that the test will build on.

ComponentInstanceName [type: BSTR] - Qualified name of the component instance, like:

`Deployment View::CPU_PIII::TSCPTestInstancemango`.

If not specified, a new component instance will be created with default parameters. Component instances have to be under `ProcessorName`.

DiffOptions [type: long] - Bitmapped value for differencing options. The bits are used as follows:

#define DIFF_SEND_MASK0x00000001

- send messages are to be included in comparison

#define DIFF_REPLY_MASK0x00000002

- reply messages are to be included in comparison

#define DIFF_SYSTEM_MASK0x00000004

- system messages are to be included in comparison

#define DIFF_STATE_MASK0x00000008

- local states are to be included in comparison

#define DIFF_DESTROY_MASK0x00000010

- destroy messages are to be included in comparison

#define DIFF_TERMINATE_MASK0x00000020

- terminate messages are to be included in comparison

#define DIFF_CALL_MASK0x00000100

- call messages are to be included in comparison

#define DIFF_RETURN_MASK0x00000200

- return messages are to be included in comparison

#define DIFF_ACTION_MASK0x00000400

- unidentified actions are to be included in comparison

#define DIFF_TRIM_MASK0x00001000

- casual trim is to be performed on the results

#define DIFF_PATH_MASK0x00002000

- capsule index path will be taken in to effect

bits set to "1" are treated as "yes"; "0" as "no".

By default the value is equal to DIFF_SEND_MASK | DIFF_REPLY_MASK | DIFF_SYSTEM_MASK | DIFF_DESTROY_MASK | DIFF_TERMINATE_MASK

HarnessLocation [type: BSTR] - Qualified name under which testing packet will be generated. Like "Logical View::User1". By default it is set to "Logical View".

LogPath [type: BSTR] - Sets the file path for the log dump if LogToFile is set to True.

LogToFile [type: Boolean] - When set to True, the output log will be sent to a file. This is useful for overnight sanity, i.e. if a static record of the runs needs to be kept around for reference.

MaxNamingRetries [type: short] - Sets the number of generated test harnesses that can stored within a model.

MaxTime [type: short] - Sets the time out period for ending the test in a worst case scenario, i.e. if the CUT(s) goes into an endless loop, becomes unresponsive, etc.

MaxTraceMessages [type: long] - Sets the tracing level for a Verify Behavior test run. Default: 1000 messages. More complex models may require this to be increased if a lot of message passing occurs within the interaction

ProcessorName [type: BSTR] - Qualified name of the processor used. Like "Deployment View::CPU_PIII". If not set first, processor will be used.

ResultsLocation [type: BSTR] - Qualified name under which the trace results will be generated. Like “Logical View::User”. By default it is set to "Logical View".

ReuseCapsuleName [type: BSTR] - Qualified name of the top capsule in the generated harness specified by ReusePackageName. Like "Logical View::TSCP_mango::TSCPTestHarnessmango". Used only with ShouldGenerateHarness set to True.

ReusePackageName [type: BSTR] - Qualified name of the package containing the harness to be reused. Like "Logical View::TSCP_mango". Used only with ShouldGenerateHarness set to True.

RQARTAuto - Name of the interaction instance in the event you did not specify a name.

ShouldCopyComponent [type: Boolean] - When set to True the component (above) will be copied and the copied component will be used for the test run. If set to False, then the component will be reused.

ShouldCopyInstance [type: Boolean] - True: component instance will be copied (default); False: component instance will be reused.

ShouldGenerateHarness [type: Boolean] - True: new harness will be generated (default). False: existing harness will be reused. If set to true proper values for ReusePackageName and ReuseCapsuleName must be specified.

StoreResultsWithHarness [type: Boolean] - When set to True, the results are saved with the generated test harness. If set to False, then the ResultsLocation property is used to determine where the results are stored.

TargetCommTimeout [type: long] - (seconds) Specifies the maximum allowed timeout for a target to be uncommunicative. If communication cannot be established with the given time, the target will be reset. **Default**: 7 seconds

TargetPort [type: short] - Sets the unique identifier for creating the communication socket between RQA-RT and the target executable.

TestName [type: BSTR] - Name that is prefixed to the generated test harness for identification purposes.

Methods

AddDriver():AddDriver(DriverPath)

- Sets a capsule as a driver for the current run, where:

DriverPath is a text string composite of the drivers' role names separated by a colon (:) :

```
<role_name_1>:<role_name_2>:...:<role_name_N>
```

- <*role_name_1*> is the name of the classifier in the collaboration, <*role_name_2*> is the classifier name in the structure of the classifier <*role_name_1*> and so on.

- In the following example:

```
Capsule1R1:Capsule2R1:Capsule3R1
```

-*TopCapsule* has sub-capsule role `Capsule1R1`, which in turn has a sub-capsule with the role `Capsule2R1`. `Capsule2R1` has a sub-capsule role `Capsule3R1`, which will be a driver.

```
GenerateInstanceDriverBehavior()
```

- generates ports and connectors to the system drivers.

void AddResetToEnd()

- Adds a reset to the current run.

boolean AddTestToEnd(IDispatch* interaction)

- Adds a new test to the current run.

void DeleteOptionsSet(BSTR szSetName)

- Will delete option set 'string' stored in the collaboration specified in the option setboolean LoadOptionsSet(BSTR szSetName).

- Will load option set 'string' from the collaboration specified in the option set. Will return False in case of failure; True otherwise. Collaboration containing the options set in question should be specified prior to execution of this command. This can be done through the Collaboration property.

void OnSelectRaceConditions(IDispatch* pRRTApp)

- Displays the Race Conditions dialog.

```
short OnVerifyBehavior(IDispatch* pRRTApp)
```

- Displays the Verify Behavior dialog.

void OnVerifyTrace(IDispatch* pRRTApp)

- Displays the Verify Trace dialog.

void RemoveAllDrivers()

- Resets all of the capsules set as drivers in the current run to be CUTs.

void RemoveAllTests()

- Removes all tests from the current run.

boolean RunVerifyBehavior(IDispatch* pRRTApp, BSTR szScriptOnCompletion)

- Runs the Verify Behavior functionality in script mode (for example, no UI). This assumes all the options for the current run have been set. The parameter **szScriptOnCompletion** allows additional functionality to run afterwards because, typically, this has to be the last command in a script since the command RunVerifyBehavior is asynchronous, i.e. it returns before having completed.

void SaveOptionsSet(BSTR szSetName)

- Will save current option in collaboration specified in the option set under the name 'string'.

Example using Summit Basic Script

Note: You can load this example script into Rose RealTime from the following directory:

$ROSERT_HOME/RRTEI/SummitBasic/ReliableService.ebs

Example

```
Sub Main
   Dim am As RoseRT.AddInManager
   Dim addIns As RoseRT.AddInCollection
   Dim addIn As RoseRT.Addin
   Dim aName As String
   Dim rqart As Object

   ' Open model containing test collaborations

   Dim pathString As String

   'NT Path
   pathString = "\Examples\Models\C++\ReliableService\"
   'To run On Sun Or HPUX use the following pathString instead:
   'pathString = "/Examples/Models/C++/ReliableService/"

   Dim origModel As String
   origModel = "ReliableService.rtmdl"

   Dim pathMap As RoseRT.PathMap
   Set pathMap = RoseRTApp.PathMap

   Dim actualPath As String
   actualPath = pathMap.GetActualPath("$ROSERT_HOME")
   pathString = actualPath + pathString
   origModel = pathString + origModel
   RoseRTApp.OpenModel origModel

   ' Sample script to start up RQA-RT Verify Behavior

   Set am = RoseRTApp.AddInManager
   Set addIns = am.AddIns
   nAddins = addIns.count
   For i = 1 To nAddIns
      Set addIn = addIns.GetAt(i)
```

```
      aName = addIn.name
      If aName = "OT::QualityArchitectRT" Then
         'Print i, aName
         GoTo foundAddin
      End If
   Next i

foundAddin:
   'Print addIn.ServerName
   Set rqart = addIn.EventHandler

   'Find the logical view
   Dim model As RoseRT.Model
   Set model = RoseRTApp.CurrentModel

foundLogicalView:
   'Set the Collaboration the interactions come from
   Dim collab As RoseRT.Collaboration
   Dim collabDiag As RoseRT.CollaborationDiagram
   Dim collabCollect As RoseRT.ModelElementCollection
   Dim collabName As String
   collabName = "ServiceProviderCollaboration"

   Set collabCollect = model.FindModelElements(collabName)
   For i = 1 To collabCollect.Count
      If collabCollect.GetAt(i).name = collabName Then
         GoTo foundCollab
      End If
   Next i

'An error has occurred.  Provide info on error then terminate
ErrorTerminate1:
   msgbox "No collaboration found with name " + collabName
   Exit Sub
```

```
foundCollab:
   Set collabDiag = collabCollect.GetAt(i)
   Set collab = collabDiag.ParentModelElement
   rqart.Collaboration collab


   'Load the appropriate options set.  Note this is configured 'for
   'NT host in this script.  To run on Sun or HPUX, change the 'NT_
prefix
   'to Sun_ or HPUX_ as appropriate to load the presaved option 'set
for the
   'specific host platform.
   rqart.LoadOptionsSet "NT_ServiceUserDriver"


   'Override the options set to dump the log to a file
   rqart.LogToFile TRUE


   'Set the LogPath Property
   Dim logString As String
   logString = pathString + "ReliableService.txt"
   rqart.LogPath logString


   Dim postScript As String
   postScript = ""
   rqart.RunVerifyBehavior RoseRTApp, postScript


End Sub
```

ObjecTime Developer to Rational Rose RealTime Migration

8

Contents

This chapter is organized as follows:

For information on general model migration issues from Objectime Developer to Rose RealTime, see the main toolset migration guide and on-line help for Rose RealTime. Since RQA-RT is an add-in, the conversion of the sequence diagram RQA-RT artifacts is a separate process.

Conversion Procedure

To convert an ObjecTime Developer model to Rational Rose RealTime:

1. Export your OTD model to Linear Form (*.lf).
2. Open the Linear Form file in Rose RealTime using File Open and change the files of type combo box to be (*.lf).
3. Follow the steps in the migration guide for converting your ObjecTime model to Rose RealTime.
4. To convert the sequence diagrams to the new Rose RealTime format, go to the Tools menu and select in the RQA-RT flyout **Convert ObjecTime RQART Diagrams**.
5. A progress bar displays and any warnings and/or problems that occurred during the conversion appear in the log pane.

This will take your existing sequence diagrams in each capsule (previously MSC's), and create a new collaboration under the same capsule. The new collaboration named `Converted_OTD_MSC` is where the converted sequence diagrams will reside. The original sequence diagrams (MSC's) are left intact and unmodified for reference purposes. You may delete them if you wish.

Differences

There are some significant differences between the original and converted sequence diagrams that are due to the architecture changes in RQA-RT.

First of all, tests are no longer run by the environment (test harness). Tests are now run by the user-specified drivers. Consequently, the conversion routine must create a driver capsule that drives the test in the same way the environment previously did by looking at the messages from the environment, and creating a driver that sends the same messages to the CUT. This is so that tests may have multiple CUT's or drivers to allow maximum flexibility and coverage of different specification scenarios (see Figure 37 and Figure 38).

C++ data is handled somewhat differently in Rational Rose RealTime.

In OTD it was possible to add code segments to the data field in a message. In Rational Rose RealTime this is not the case. The data field is merely dropped in the signal send call.

To accommodate this difference C++ data from OTD messages is mapped onto a new local action, which will appear before the message that contained the C++ data.

Note: RPL data cannot be converted over to Rational Rose RealTime models. A local action will be created, but the RPL code is commented out. You must translate this manually. A warning in the Rational Rose RealTime log will appear, to alert you to this.

Figure 37 Unconverted Sequence Diagram

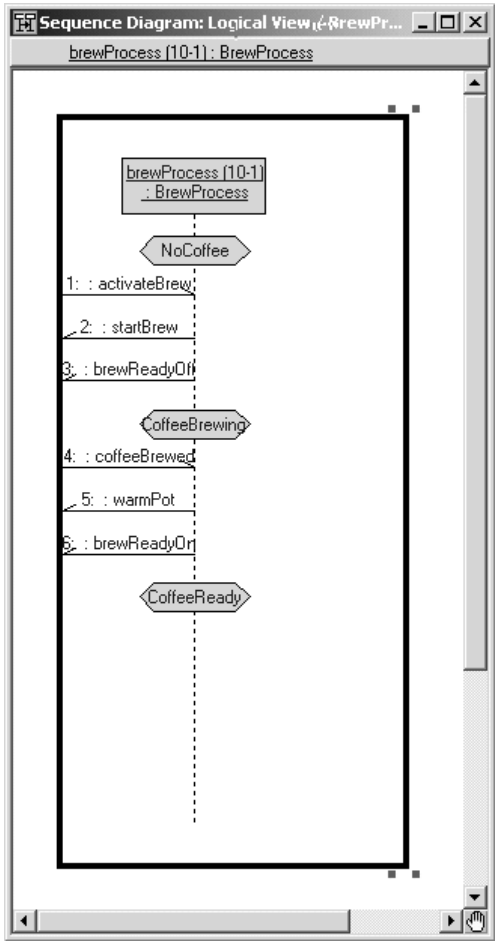
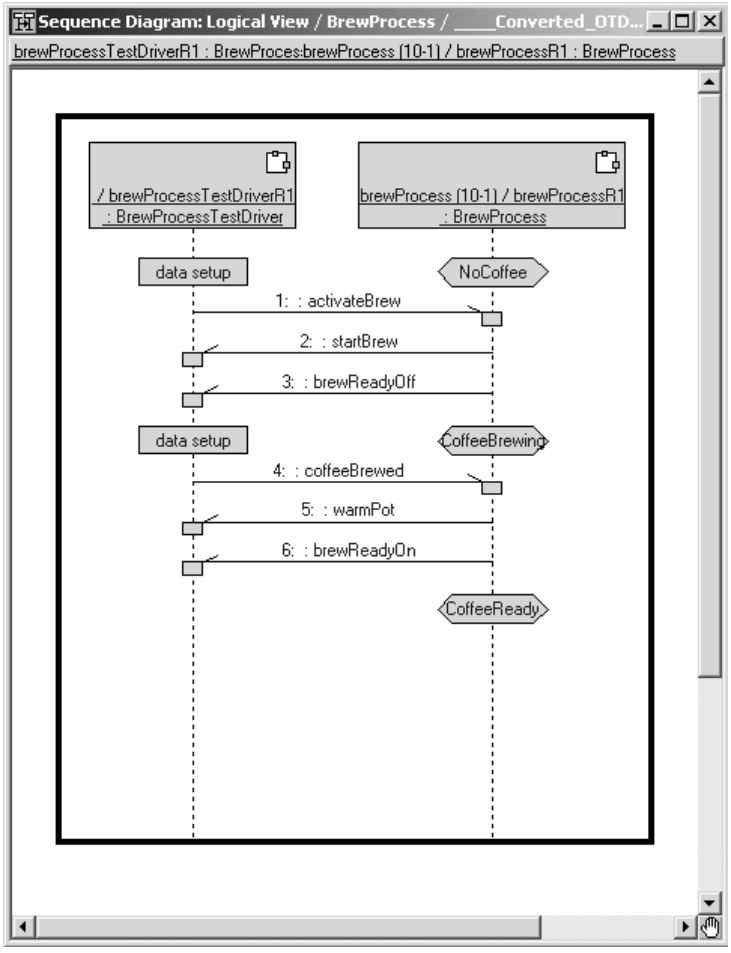


Figure 38 Converted Sequence Diagram



Index

A

B

C

D

F

G

H

I

L

M

O

P

R

S

T

U

V

W