

**Rational.** Rhapsody

**IBM.**

## **IBM® Rational® Rhapsody® TestConductor Add On**



**User Guide**

***Rhapsody®***

**IBM® Rational® Rhapsody®  
TestConductor Add On**

**User Guide**

**Release 2.8.0**



## **License Agreement**

No part of this publication may be reproduced, transmitted, stored in a retrieval system, nor translated into any human or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual or otherwise, without the prior written permission of the copyright owner, BTC Embedded Systems AG.

The information in this publication is subject to change without notice, and BTC Embedded Systems AG assumes no responsibility for any errors which may appear herein. No warranties, either expressed or implied, are made regarding Rhapsody software including documentation and its fitness for any particular purpose.

## **Trademarks**

IBM<sup>®</sup> Rational<sup>®</sup> Rhapsody<sup>®</sup>, IBM<sup>®</sup> Rational<sup>®</sup> Rhapsody<sup>®</sup> Automatic Test Generation Add On, and IBM<sup>®</sup> Rational<sup>®</sup> Rhapsody<sup>®</sup> TestConductor Add On are registered trademarks of IBM Corporation.

All other product or company names mentioned herein may be trademarks or registered trademarks of their respective owners.

© Copyright 2000-2017 BTC Embedded Systems AG. All rights reserved.

# Contents

---

## Content

<b>Contents.....</b>	<b>4</b>
Contacting IBM® Rational® Software Support.....	9
<b>About this document.....</b>	<b>10</b>
<b>Preliminary Note.....</b>	<b>11</b>
<b>Introduction.....</b>	<b>14</b>
<b>Rhapsody Testing Profile.....</b>	<b>18</b>
Adding the Testing Profile automatically.....	18
Adding the Testing Profile manually.....	18
Using the Testing Profile.....	19
Refining Testing Profile Stereotypes.....	19
<b>Model-based Unit Test Definition.....</b>	<b>21</b>
TestArchitectures.....	22
Replacements.....	24
Dependencies used for Navigation on Replacements.....	25
Interfaces.....	26
Ports.....	27
VariationPoints and Variants.....	27
Inheritance.....	28
Templates and Template Instances.....	28
Automatic TestArchitecture Generation.....	28
Context Menu 'Create TestArchitecture'.....	28
Test scheduling with <<Scheduler>> TestComponents.....	31
Test arbitration with <<Arbiter>> TestComponents.....	32
Creating test executables with TestingConfigurations.....	33
Generate and Build the TestContext.....	33
Using Classes (UML) and Blocks (SysML).....	33
Using Objects.....	33
Using Files (Modules).....	35
Using Parts of composite classes.....	35
GreyBox TestArchitectures for classes and objects.....	35
TestArchitectures with multiple SUT classes or objects.....	36
Updating TestArchitectures.....	37
Up-to-date check for TestArchitectures.....	37
TestArchitectures for MicroC Models.....	38
TestArchitectures for Code centric Models.....	38
Production Code (Black Box) Testing.....	39
Black Box Testing.....	39
Grey Box Testing.....	40
TestCase Definition.....	42
TestCase Definition with Code.....	42
Defining a Code TestCase.....	42

Testing reactive behavior with Code TestCases.....	42
TestCase Definition with Flow Charts.....	43
Defining a Flow Chart TestCase.....	43
Testing reactive behavior with Flow Chart TestCases.....	43
TestCase Definition with Statecharts.....	43
Defining a Statechart TestCase.....	43
TestCase Definition with Sequence Diagrams.....	44
Defining a Sequence Diagram TestCase.....	44
Failure Analysis in Sequence Diagram TestCases.....	45
TestConductor.h, TestConductor_C.h and TestConductor_C.c.....	45
Support for interfacing Files in C using <<CInterfaceFile>> Stereotype.....	45
TestConductor Support for Testing Private Operations in Rhapsody in C.....	46
TestConductor Support for Testing Private and Protected Operations in Rhapsody in C++.....	47
Support for Rhapsody Action Language.....	48
Model Population – Create Driver Operations and StubOperations.....	48
Driver Operations.....	48
StubOperations.....	49
Clean TestComponent.....	52
Clean TestPackage.....	52
Specifying a TestScenario.....	52
Creating TestCases with the TestCase wizard.....	52
Creating Sequence Diagram TestCases from existing Scenarios using an explicit instance mapping.....	55
Definition of mappings for sequence diagram TestCase creation from existing scenarios.....	56
SDMappings for Replacements.....	58
<b>Test Execution.....</b>	<b>59</b>
Overview.....	59
Testing Configuration.....	59
Tags of the <<TestingConfiguration>> Stereotype.....	60
TestConfiguration Dependency.....	67
Execution Results.....	67
Performing result verification for TestCase execution.....	68
TestCase Execution.....	69
Test Execution Dialog for code, flow chart, statechart based tests.....	70
Test Execution Dialog.....	70
Test Information.....	71
Controlling TestCase execution.....	71
Test Execution Dialog for sequence diagram based tests.....	71
Test Execution Dialog.....	71
Test Information.....	72
Displaying Test Results by witness scenarios.....	73
Automatically adding witness scenarios to the model for failed SDInstances.....	73
Abort Test Execution.....	73
Execution Timeout.....	73
Test Execution Report.....	75
Debugging TestCases.....	76
TestContext Execution.....	76
Starting Test Execution.....	76
Stopping Test Execution.....	77
Execution Timeout.....	77
Ordering of TestCases.....	77
Test Execution Report for TestContext.....	78

TestPackage Execution.....	78
Starting Test Execution.....	78
Stopping Execution.....	79
Execution Timeout.....	79
Test Execution Report for TestPackage.....	79
Computing Model Coverage during Test Execution.....	79
Computing Model Coverage for single TestCases.....	80
Coverage Items.....	80
Choosing the Coverage Kind for Model Coverage.....	81
Model Coverage Measurement and Animation Instrumentation.....	81
Traceability of Coverage Items.....	81
Computing Requirement Coverage.....	82
Computing Requirement Coverage for TestCases and TestContexts.....	82
Transitivity of Dependencies (Refinement of model elements and requirements).....	83
Computing Code Coverage.....	85
TestConductor code coverage criteria.....	86
Command Line Execution.....	90
Command Line Syntax for rhapsody.exe.....	90
Command Line Syntax for rhapsodycl.exe.....	92
Test Execution Report.....	93
TestCase Execution on Targets.....	93
<b>Test Management.....</b>	<b>95</b>
Managing Test Data.....	95
Linking TestCase to Requirements.....	95
TestConductor Dialog.....	96
TestConductor Settings.....	96
General Properties.....	98
TestContext Properties.....	101
Generating Test Reports with Rhapsody ReporterPLUS.....	102
Executing the ReporterPLUS with the Test Report Template.....	102
Using the HTML Test Report.....	103
Using the Test Requirement Coverage Report.....	103
Customizing the Test Report.....	104
Generating Test Reports with Rational Publishing Engine.....	104
Creating the Test Report.....	104
Test Requirement Coverage Report.....	104
Creating Report Templates.....	105
Using the TestConductor API.....	105
Available TestConductor API Commands.....	105
Defining Callbacks for TestConductor functions.....	107
<b>Specifying Requirements with Sequence Diagrams.....</b>	<b>108</b>
Supported Diagram Elements in TestScenarios.....	108
Limitations of design elements (sequence diagrams).....	110
Message Realization.....	110
Ignoring Unrealized Messages.....	110
Virtual Call vs Nonvirtual Call (Rhapsody in C++).....	111
Self-Messages in BlackBox and GreyBox Testing.....	113
SelfMessageRealizationInParts.....	113
Using Time Interval for Delay Driving from TestContext and TestComponents.....	114
Specifying Argument Values.....	114

Specifying dataflows.....	115
Specifying Return Values.....	115
Specification of Out and InOut Argument Values.....	116
Interaction Occurrence – Reference Sequence Diagram.....	117
Don't care values.....	117
Range Specification.....	118
Influencing DriverOperation and StubOperation Generation.....	119
User Defined DriverOperations.....	119
User Defined StubOperations.....	120
Influencing DriverOperation and Stub generation using <<RTC_MsgInfo>> tags.....	120
RTC_DriverInitCode and RTC_DriverInitCodeAdditional.....	121
RTC_DriverCallCode and RTC_DriverCallCodeAdditional.....	121
RTC_StubBodyCode.....	122
Deleting <<RTC_MsgInfo>> Tags (User Defined Driver and Stubs).....	122
Influencing DriverOperation and Stub generation using TestActions in TestScenarios.....	122
Clean TestComponent.....	125
Clean TestPackage.....	125
(general) TestActions, TestAssignments and TestConditions.....	125
Preconditions (for SysML/HarmonySE).....	127
Using <check> Conditions / TestCondition.....	127
Using Interaction Operators in SD TestCases.....	129
Using Serialize/Unserialize Functions for User Defined Types.....	130
Using auto generated serialization/unserialization functions.....	130
Using manually defined serialization/unserialization functions.....	131
<b>Failure Analysis.....</b>	<b>132</b>
Failure Analysis using Witness Scenarios.....	133
Failure Analysis for InteractionOccurrences.....	134
Debugging TestCases.....	136
Result Verification.....	136
<b>Using TestConductor from Eclipse.....</b>	<b>137</b>
<b>TestConductor Rhapsody Plugins.....</b>	<b>138</b>
TestConductor Merge Coverage Reports Plugin.....	138
Merging model coverage reports.....	138
Merging code coverage reports.....	139
Merging requirement coverage reports.....	139
TestConductor Rhapsody Quality Manager Plugin.....	140
TestConductor Check Model Plugin.....	141
<b>Appendix.....</b>	<b>143</b>
Definitions of the Rhapsody Testing Profile.....	143
Structure Overview.....	143
UML Testing Profile (UML20TP) Package.....	143
TestArchitecture Package.....	144
TestBehavior Package.....	144
TestConductor (RTC) Package.....	145
TestArchitecture Package.....	145
TestBehavior Package.....	150
TestDocumentation Package.....	154
Automatic Test Generation (ATG) Package.....	155
Formal Testing Package.....	155

TestConductor Assert Macros (C/C++).....	156
Using IntelliVisor for TestConductor Assert Macros.....	158
Testing AUTOSAR Models.....	160
Unit testing of AUTOSAR Software Components.....	160
Migrating animation based TestArchitecture to assertion based TestArchitecture.....	166
Automatic Migration of animation based TestArchitectures to assertion based Testing mode...	167
Functional Limitations.....	168



## Contacting IBM® Rational® Software Support

IBM Rational Software Support provides you with technical assistance. The IBM Rational Software Support Home page for Rational products can be found at <http://www.ibm.com/software/rational/support/>.

For contact information and guidelines or reference materials that you need for support, read the [IBM Software Support Handbook](#).

For Rational software product news, events, and other information, visit the [IBM Rational Software Web site](#).

Voice support is available to all current contract holders by dialing a telephone number in your country (where available). For specific country phone numbers, go to <http://www.ibm.com/planetwide>.

Before you contact IBM Rational Software Support, gather the background information that you will need to describe your problem. When describing a problem to an IBM software support specialist, be as specific as possible and include all relevant background information so that the specialist can help you solve the problem efficiently. To save time, know the answers to these questions:

What software versions were you running when the problem occurred?

Do you have logs, traces, or messages that are related to the problem?

Can you reproduce the problem? If so, what steps do you take to reproduce it?

Is there a workaround for the problem? If so, be prepared to describe the workaround.

# About this document

---

This document is part of the documentation of the IBM<sup>®</sup> Rational<sup>®</sup> Rhapsody<sup>®</sup> TestConductor Add On.

Further documentation can be found in the Rhapsody documentation folder in <RhapInstall>/Doc/pdf\_docs and <RhapInstall>/Doc/html\_docs:

- [TestingCookbook \(open index.html\)](#)  
A collection of TestConductor related questions and answers with examples.
- Tutorials:
  - [TestConductor\\_Tutorial\\_Ada.pdf](#)
  - [TestConductor\\_Tutorial\\_C.pdf](#)
  - [TestConductor\\_Tutorial\\_Cpp.pdf](#)
  - [TestConductor\\_Tutorial\\_Java.pdf](#)
- [RQMTestConductorAdapter\\_HowTo.pdf](#)  
Small document describing the TestConductor Adapter to Rhapsody Quality Manager and how to use the adapter.
- [RTC\\_Release\\_Notes.pdf](#)  
TestConductor Release Notes.
- [RTC\\_User\\_Guide.pdf](#)  
TestConductor User Guide for Rhapsody in Java and Rhapsody in Ada.
- [TC\\_CodeCoverage\\_Limitations.pdf](#)  
Document describing features of the TestConductor Code Coverage Measurement and its limitations.
- [Testing with TestConductor on a small target.pdf](#)  
Document describing TestConductor's generic approach for testing on a target. The approach is based on providing a simple proxy for compilation, download to target, execution control and transfer of results. The document describes also the usage of an example proxy using eclipse.
- [Testing with TestConductor on an Integrity Target.pdf](#)
- [Testing\\_with\\_RTC\\_on\\_a\\_Linux\\_Target.pdf](#)
- [Testing\\_with\\_RTC\\_on\\_a\\_VxWorks\\_Target.pdf](#)

# Preliminary Note

---

The terms SUT, TestContext and TestComponent are defined in the UML Testing Profile, specified by the Object Management Group (OMG). The Rhapsody Testing Profile is based on the UML Testing Profile (cf. section Rhapsody Testing Profile on page 18).

Throughout this document we use the terms SUT, TestContext and TestComponent in a logical and in a technical manner:

- SUT (“System Under Test”) denotes the classes, objects or files to be tested. The SUT is taken 'as is' - without affecting or modifying its behavior. In its logical meaning, SUT can be an individual model element as well as a set of classes, objects or files with their relations among each other.

In its technical meaning, `<<SUT>>` is a new term on Rhapsody meta class `Object` defined by the Rhapsody Testing Profile. Besides `<<SUT>>`, the Rhapsody Testing Profile also defines the new terms

- `<<TestSUTObject>>` – used for global objects considered as SUT. `TestConductor` distinguishes the stereotype `<<SUT>>` for instantiation of SUT classes as part of the `TestContext` and `<<TestSUTObjects>>` for instantiation of SUT classes or SUT objects as global objects outside the `TestContext`. Also classes, objects and files not explicitly instantiated and stereotyped in the `TestArchitecture` are logically regarded as SUT. As a rule of thumb, it can be stated: **all model elements which are not marked to be TestComponents belong to the SUT.**
- `<<TestSUT>>` – for Grey Box Testing (cf. sections GreyBox TestArchitectures for classes and objects on page 35 and Grey Box Testing on page 40), it is necessary to instrument also parts of the SUT with assertions enabling observation. This instrumentation is never applied to original model elements but on copies of the affected model elements (cf. section Replacements on page 24). From the logical point of view, `<<TestSUT>>` is treated like other SUT elements, even though `<<TestSUT>>` technically denotes a testing artifact.
- TestComponent logically denotes a testing artifact that can be instrumented and modified for testing purposes. TestComponent form the environment of the SUT in the `TestArchitecture`. This environment has to conform to the SUT's declarations of relations to other model elements to be able to act as communication partner of the SUT. If the SUT requires relations to e.g. implicit objects or files, then the environment has to provide appropriate candidates for the SUT's relations.

Technically, `<<TestComponent>>` is a new term on Rhapsody meta class `Class`.

Using only classes as TestComponents would not permit many desired use cases, such as providing test artifacts for singleton objects or files (Rhapsody in C).

Thus, the logical term TestComponent comprise more than only classes in the test environment.

Technically, the Rhapsody Testing Profile defines the following stereotypes and new terms (list not complete) to denote model elements belonging to the test environment for a SUT:

- <<TestComponent>> – new term on Rhapsody meta class Class. A <<TestComponent>> class can inherit from an interface or a model class, replace a model class in the code generation scope (cf section Replacements on page 24), can be a <<Variant>> of a <<VariationPoint>> or can be newly introduced as additional testing artifact to the TestArchitecture – such as a 'DummyDriver' or the TestContext.
- <<TestFile>> – new term on Rhapsody meta class Module. Modules are displayed as 'File' in the Rhapsody browser. Modules are mainly supported for Rhapsody in C<sup>1</sup>.  
Since Rhapsody in C only supports inheritance from interfaces, <<TestFile>> files can inherit from a <<CInterfaceFile>> (cf. section Support for interfacing Files in C using <<CInterfaceFile>> Stereotype on page 45), can be a <<Variant>> of a <<VariationPoint>> or replace a model file in the code generation scope.
- <<TestContext>> – new term on Rhapsody meta class Class. The TestContext is a specific TestComponent, aimed at instantiating SUT and test environment and the relations among the elements belonging to the TestArchitectures. The TestContext owns the TestCases and is the backbone of test organization. Since the TestContext has relations to all elements belonging to the TestArchitecture, the TestContext can also be used for test execution, e.g. providing the SUT with stimuli.
- <<TestComponentInstance>> – instantiation of a <<TestComponent>> class as part of the TestContext.
- <<TestComponentObject>> – a TestComponentObject is either a global object of a <<TestComponent>> class in the TestPackage or it is itself a replacement of an implicit object in TestComponent place.  
TestComponentObjects are needed only for TestArchitectures using global objects, when instantiation of SUT and TestComponents global objects is preferred to instantiating them as parts of the TestContext, e.g. if implicit objects have to be dealt with (cf. sections Replacements on page 24 ff and Using Objects on page 33).

Even though there is a variety of distinguished new terms denoting model elements in TestComponent place, these testing artifacts adhere to common rules regarding generation of driver operations and stubbing and instrumentation with assertions. We therefore refer often to the logical term 'TestComponent' throughout this document – comprising <<TestContext>> class, <<TestComponent>> classes, implicit classes of <<TestFile>> files and implicit classes of <<TestComponentObject>> objects. Similarly we often refer to the

---

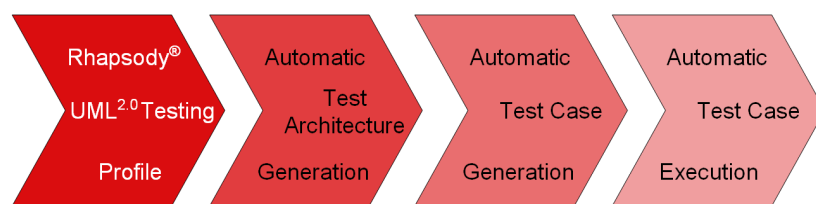
<sup>1</sup>Rhapsody in C++ supports Modules/Files only for use of external sources, while Rhapsody in C also provides code generation for the model element Module.

logical term 'TestComponentInstance' comprising  
<<TestComponentInstance>> parts of a TestContext, global  
<<TestComponentObject>> objects and <<TestFile>> files.

# Introduction

---

Welcome to the User Guide for IBM® Rational® Rhapsody® TestConductor Add On. TestConductor is part of the Rhapsody Testing Environment which is based on three main components: “Automatic TestArchitecture Generation”, “Automatic TestCase Execution” and “Automatic TestCase Generation”. These three components are developed along the UML Testing Profile as implemented in Rhapsody.



*Figure 1: Rhapsody Testing Profile and TestConductor*

TestConductor supports the two main features “Automatic TestArchitecture Generation” and “Automatic TestCase Execution” of the Rhapsody Testing Environment. The optional IBM® Rational® Rhapsody® Automatic Test Generation Add On (ATG) supports the feature “Automatic TestCase Generation”.

In the Rhapsody Testing Environment the implementation of TestCases can be chosen out of:

- Sequence diagrams
- Statecharts
- Flow charts (only Rhapsody in C/C++)
- Pure code

The Rhapsody Testing Environment provides the ability to test a design against its requirements. Advantages of using sequence diagrams as TestCases are:

- Graphical definition
- Monitors/drivers
- Parameterized sequence diagrams
- Color-coded failure sequence diagrams

TestConductor is a model based testing environment used to debug and test object-oriented embedded software designed in Rhapsody. TestConductor supports unit testing as well as software integration testing based on graphical test definitions using sequence

diagrams. TestConductor supports Rhapsody in C++, Rhapsody in C, Rhapsody in Java and Rhapsody in Ada. In Rhapsody in C++, Rhapsody in C, Rhapsody in Java, and Rhapsody in Ada TestCases can be defined also by statecharts or pure code. For Rhapsody in C++ and Rhapsody in C TestCases can also be defined by FlowCharts.

Using sequence diagram related TestCases, TestConductor supports an advanced graphical failure analysis. These features make it easy to define and execute extensive test suites, as well as to create complex tests drivers and test monitors.

This document regards only the so called assertion based testing mode, which is applicable to Rhapsody in C and Rhapsody in C++ only. For Rhapsody in Java or Rhapsody in Ada (or when using the so called animation based testing mode for C++ or C) please have a look into the document `RTC_User_Guide.pdf`.

## Using TestConductor

This manual assumes that Rhapsody and TestConductor are already installed on your system, and that you have a valid license. If you need assistance with installation or licensing, contact customer support.

To execute tests, TestConductor relies on the compiled and linked model code of the TestArchitecture. Therefore, the project with the system under test must be in a state such that you can compile and run the TestArchitecture.

## Rhapsody Testing Profile

The Rhapsody Testing Profile contains new terms and stereotypes that can be used to model test artifacts in Rhapsody. It is based on the official UML Testing Profile. However, several elements defined in the UML Testing Profile are currently not part of the Rhapsody Testing Profile, while the Rhapsody Testing Profile contains additional elements that are not part of the UML Testing Profile. These additional elements are used for test activities that are not addressed by the UML Testing Profile, for instance stubbing.

The definitions of the Rhapsody Testing Profile are listed and explained in detail in appendix Definitions of the Rhapsody Testing Profile on page 143 ff.

## Automatic TestArchitecture Generation

The *automatic TestArchitecture generation* – first supporting layer of the Rhapsody Testing Environment and part of TestConductor – automates the complex task of creating the test environment for e.g. arbitrary classes of the UML design. The TestArchitecture allows modeling of all kinds of test artifacts, including driver or stub code, and the relations between tests and tested requirements or functionality without modifying the tested design. TestConductor supports a strict separation of design elements from test elements while enabling traceability and reusing modeled information.

From the Rhapsody project the user easily initiates the automatic generation of a TestArchitecture including:

- Creation of a new TestPackage
- Creation of a new TestContext including

- System under test (“SUT”)
- TestComponents
- Links between SUT and TestComponents

TestConductor offers two different modes for TestArchitecture creation (see section TestArchitectures on page 22 ff for details):

- TestArchitecture using parts
- TestArchitecture using global objects

While using parts is appropriate for testing classes, implicit objects, singletons in Rhapsody in C and files can not be instantiated as parts of a TestContext. Such model elements have to be instantiated as global objects.

TestConductor supports BlackBox- and GreyBox testing (cf. sections Black Box Testing and Grey Box Testing on page 39 ff) for both kinds of TestArchitectures.

## TestCase Definition

A *TestCase* represents the smallest element that can be defined and executed by TestConductor. A TestCase describes a sequence of input stimuli and expected behavior, in order to verify a certain functional behavior of a system under test. TestCases can define both, black box and white box behavior.

TestConductor supports several ways to define TestCases:

- Sequence diagrams
- Statecharts
- Flow charts
- Pure code

With the optional add-on Rhapsody® Automatic Test Generation (ATG™) for Rhapsody in C++ TestCases can be generated automatically.

## TestCase Execution

TestConductor is a *TestCase execution engine* and represents the second stage of the Rhapsody Testing Environment. It enhances the testing capabilities by not only executing the automatically generated TestArchitecture, but it also offers a test execution analysis with respect to the expected results. If the TestCase e.g. is implemented by a sequence diagram the expected behavior is expressed by

- The ordering of defined messages
- Parameter values of messages
- Messages from SUT to testing components



- Specified return values on operation calls

## TestResult Representation

TestConductor presents TestExecution results as reports which are maintained as part of the model. Besides an execution report (see section Execution Results on page 67), also reports for various coverage measures are generated and added to the model:

- ModelCoverage (cf. section Computing Model Coverage during Test Execution on page 79)
- RequirementCoverage (cf. section Computing Requirement Coverage on page 82)
- CodeCoverage (cf. section Computing Code Coverage on page 85).

# Rhapsody Testing Profile

---

The *Rhapsody Testing Profile* is based on the official UML Testing Profile. It contains new terms and stereotypes that can be utilized for model testing artifacts in Rhapsody. A couple of elements defined in the UML Testing Profile are presently not part of the Rhapsody Testing Profile. However, the Rhapsody Testing Profile includes supplementary elements that are not part of the UML Testing Profile. Stubbing, for example, is one of these additional elements that are used for test activities which is not not addressed by the UML Testing Profile.

For further information on the Rhapsody Testing Profile please refer to the *TestConductor Tutorial*, where depict examples on the Rhapsody Testing Profile are provided.

The definitions of the Rhapsody Testing Profile, i.e. stereotypes and new terms overriding properties and defining tags, are listed and explained in appendix Definitions of the Rhapsody Testing Profile on page 143 ff.

## Adding the Testing Profile automatically

The first usage of any TestConductor functionality automatically adds the Rhapsody Testing Profile to a model. For example this can be done by choosing the Rhapsody menu entry **Tools > TestConductor**.

In case the model does not yet contain the actual Rhapsody Testing Profile, TestConductor offers to add the missing Rhapsody Testing Profile automatically.

In case the Rhapsody Testing Profile is unloaded, TestConductor ask to load it.

In case a loaded profile already uses the name “TestingProfile” Rhapsody TestConductor advises the user.

Once the Rhapsody Testing Profile has been loaded into a Rhapsody project by starting TestConductor the Rhapsody browser window will contain the above stated testing profile packages and its individual sub-packages as shown in the following picture.

## Adding the Testing Profile manually

It is also possible to add the testing profile manually to a model:

- Open your project in Rhapsody
- Select the menu item **File > Add Profile to Model...**
- Select the following **Data Type**: ‘Profile (\*.sbs)’
- Select in Rhapsody installation folder:  
‘...\\Share\\Profiles\\TestingProfile\\TestingProfile\_rpy\\TestingProfile.sbs’
- Press **Open** to add the Rhapsody Testing Profile to the model.

## Using the Testing Profile

The Rhapsody Testing Profile defines a set of stereotypes and new terms which are automatically applied on model elements by Rhapsody TestConductor in

- TestArchitecture creation,
- TestCase creation, definition and specification,
- automatic preparation for TestCase execution,
- result management,
- coverage measurement and
- reporting.

TestConductor provides a set of context menu helpers applicable on certain model elements. For example, TestArchitecture creation is applicable on model elements Class, Object, Block, Module among others. E.g. when creating a TestArchitecture for a class, TestConductor creates a TestPackage (new term on package) hierarchy, with a <<TestingConfiguration>> stereotyped code generation configuration, a <<TestContext>> new termed class structuring and organizing the relations of SUT and <<TestComponent>> new termed classes forming the environment of the SUT.

By using these stereotypes and new terms, TestConductor adds functionality applicable to stereotyped and new termed model elements to Rhapsody and tailors Rhapsody behavior on model elements to testing purposes.

## Refining Testing Profile Stereotypes

Most model elements in a TestArchitecture created by TestConductor are marked with stereotypes or new terms defined in the Testing Profile. Stereotypes are used for three functions:

- To arrange special elements in the same group in the model browser ('new term' stereotypes, some of them with their own icon);
- As a hook for TestConductor actions (TestConductor actions are only available on certain elements);
- Stereotypes add or modify certain properties/tags of elements of the TestArchitecture.

For example TestCases in a TestArchitecture are basically operations provided with the new term stereotype <<TestCase>>, which sets some property values and leads to grouping all TestCases in the model browser underneath the node TestCases (instead of operations). Also several TestConductor actions (e.g. "Update TestCase") are only possible for <<TestCases>> but not for ordinary operations. As another example the stereotype <<TestingConfiguration>> is used to distinguish standard configurations from TestingConfigurations which are adjusted to the special needs of the TestConductor TestArchitecture. A <<TestingConfiguration>> has additional tags for configuring additional features (like coverage measurement) or fine-tuning the test execution (e.g. rtc\_log\_kind to define the manner of logging).

Users may wish to create their own stereotypes to have a simple and transparent way to induce specific changes to elements in reoccurring scenarios. But if settings or tags are to be modified which are also affected by a coexisting Testing Profile stereotype on the same element – meaning that two stereotypes are trying to modify the same property in the same way – it is not sure which stereotype's modification is actually applied on the element, therefore it is not recommended to have conflicting stereotypes. The option to replace the Testing Profile stereotype with the user stereotype is not advised either, since the Testing Profile stereotypes act as hooks for TestConductor actions, thus disabling TestConductor functionality on that element. The solution is to have the user stereotype inheriting from the Testing Profile stereotype, thus preventing conflicts and preserving TestConductor functionality on that element<sup>2</sup>.

In fact the Testing Profile already provides such a refined stereotype: The stereotype <<TargetTestingConfiguration>> inherits from stereotype <<TestingConfiguration>> and adds additional tags and modifications to properties suitable for test execution on target. Because of the inheritance of the original stereotype <<TestingConfiguration>> all TestConductor actions expecting a TestingConfiguration will accept this <<TargetTestingConfiguration>> as well.

---

<sup>2</sup>Note that changing default values of TestConductor stereotypes may affect the functionality of the TestArchitecture.

# Model-based Unit Test Definition

---

The term *unit test* is often used within the software development, but interpreted quite different. Unit tests are performed on differently large software units like simple functions, simple classes up to complex function libraries. However, the goal of each unit test is in most cases the same. On the one hand the unit is tested for its functional behavior. On the other hand often additionally structural analysis are accomplished, in order to find uncovered (dead) code.

In order to prepare, execute, and assess a unit test several steps are usually performed:

1. A TestArchitecture (or test harness or test frame) must be constructed
2. TestCases must be defined and implemented
3. TestCases must be executed on the host machine
4. TestCases must be executed on the target machine

Each of the four mentioned steps is usually time consuming and difficult to perform. TestConductor makes the preparation, execution, and the assessment of tests much easier by lifting the test process up to the level of UML models, and by offering a high degree of automation for the steps listed above.

TestConductor supports unit testing on model-level by following the UML Testing Profile. Therefore TestConductor automates the time consuming and complex task of test environment creation. The automatic TestArchitecture generation can be used for:

- Simple classes (In SysML: Activities, Blocks, Viewpoint)
- Simple classes with inheritance
- Composite classes
- Composite classes with inheritance
- Objects (In SysML: Parts)
- Files (Modules)

The other complex task of unit testing is the definition of TestCase or TestScenarios, typically done by writing test code in the same language than the unit to be tested. Model-based unit testing with TestConductor combines the advantage of graphical TestCase definition via sequence diagrams or flow charts with the familiar pure code based TestCases. Using the optional add-on Rhapsody Automatic Test Generation (ATG), you have also the possibility to perform automatic TestCase generation.

## TestArchitectures

Testing units of a Rhapsody model using the Rhapsody Testing Profile requires certain preparation steps to be repeatedly performed. Therefore TestConductor provides a powerful feature that creates the complete *TestArchitecture* automatically. Automatic TestArchitecture generation means:

- Creation of a new TestPackage.
- Creation of a new TestContext.
- Depending on the chosen mode for TestArchitecture creation either instantiation of the selected SUT class as part of the TestContext or as a global object in the TestPackage.
- Creation of TestComponents.
- Depending on the chosen mode for TestArchitecture creation either instantiation and 'wiring' of TestComponentInstances as parts of the TestContext or as global TestComponentObjects in the TestPackage.
- Creation of an adequate code generation component and configuration.
- Adding a test configuration (dependency-relation) to the TestContext referring to the created code generation configuration.
- Creation and drawing of a TestContext diagram.

Fundamentally, TestConductor supports two different testing modes: Animation based and assertion based testing mode. TestArchitecture creation will create different resulting TestArchitectures depending on the chosen testing mode:

- animation based testing mode (applicable to C, C++, Java, Ada models): In animation based testing mode, the scheduling and arbitration, i.e., the way TestConductor decides whether a TestCase is passed or failed, is based on animation messages coming from Rhapsody's animation feature. In particular comparison of message observations to the expectations according to the test specification relies on serialization underlying the animation feature. Test execution is based upon running an appropriate test specific observer in the Rhapsody process communicating with the tested application via the Rhapsody animation socket. Hence, animation based testing mode always requires:
  - animation instrumentation (including requirement of appropriate serialization for types, objects, classes, functions, events, e.t.c)
  - socket connection between tested application and Rhapsody application.
- assertion based testing mode (applicable to C and C++ models only, not available for Java and Ada): In contrast to animation based testing mode, in assertion based testing mode both scheduling and arbitration of TestCases is directly controlled by assertions that are compiled into the test executable, i.e., scheduling and arbitration of TestCases is independent from Rhapsody's animation feature. Since in assertion based testing mode the TestCases are part of the application itself, neither animation instrumentation nor socket connection between tested application and Rhapsody application is required, giving way for testing the application without the animation overhead (e.g. enabling testing production code)

as well as testing without the requirement of a runtime connection to the tested application (e.g. enabling testing on target).

**This document only refers to the assertion based testing mode.**

For TestArchitecture creation, so-called replacement TestComponents (cf. Section Replacements on page 24) will be introduced, if inheriting TestComponents don't allow overwriting of behavior according to the needs of test execution. For replacements, it can be preselected whether stubs or wrapper will be created (property `TestConductor.Settings.ReplacementCreationMode = {Wrapper, Stub}`, Wrapper is the default).

TestArchitecture generation can be customized interactively using property `TestConductor::Settings::CreateTestArchitectureMode` (cf TestConductor settings “General Properties”, page 98).

If `CreateTestArchitectureMode` is set to ‘Standard’, then project properties are used in the generated code generation Configuration, the compile environment is set to the project's default environment and animation instrumentation is enabled while ‘Advanced’ opens a dialog that allows selection of an existing configuration from which all overridden properties, settings, and scope settings will be inherited.

When choosing 'Advanced' TestArchitecture creation mode for assertion based TestArchitectures, a dialog will appear, letting the user individually choose the replacement's kind of implementation for each TestComponent.

It may sometimes be necessary to manually adjust the scope of the code generation Component after automatic TestArchitecture creation. In rare cases, all classes of one package may have been replaced by replacements, but types or events of that package still need to be regarded in the scope. In this case, it might be helpful to select a package with right-click instead of left-click. While left-clicking a package in the scope dialog selects the package and its contents, right-click selects only the package and its non-selectable content.

Note that TestConductor can't determine meaningful parameters for non-standard constructors automatically for instances of TestComponents or classes having no default constructor. It might be necessary to manually adjust the constructor calls for TestComponentInstances or for the SUT after TestArchitecture creation w.r.t. constructor arguments.

By default, TestArchitectures are created as '*BlackBox*' TestArchitectures, i.e. the SUT will not be instrumented with testing assertions and thus internals of the SUT such as self-invocation of operations or communications among parts of the SUT can not be considered in sequence diagram TestCases. When setting property `TestConductor::Settings::CreateTestArchitectureTransparency` to '*GreyBox*' (cf TestConductor settings “General Properties”, page 98), TestArchitecture creation will create a copy of the SUT model element in the TestArchitecture. This copy will be used as replacement of the original SUT model element and enables TestConductor to instrument the SUT replacement for testing purposes. Using 'GreyBox' testing, also self messages as well as communication among parts of the SUT can be considered in sequence diagram TestCases.

Using the default TestArchitecture creation, implicit objects can neither be grey box tested, nor can implicit objects or singletons be used in TestComponent position. Stubbing in implicit objects or singletons is not supported by default.

Optionally, TestArchitecture creation can be customized by checking property `TestConductor::Settings::CreateTestArchitectureUsingGlobalObjects` (cf TestConductor settings “General Properties”, page 98). If this property is set, then TestArchitecture creation will instantiate all SUT classes and TestComponent classes as global objects instead of instantiating parts for classes. This construction enables also usage of links to instantiate associations among classes and implicit objects. This would be impossible for parts connected to global objects, since these connections would cross composite class boundaries, which is not supported by Rhapsody code generation in general. Fundamental support of global objects as test artifacts enables also treatment of implicit objects by object replacement copies for grey box testing (on SUT side) as well as for stubbing (in TestComponent place). Using global objects is recommended if implicit objects or singleton objects are involved in modeling in particular in Rhapsody in C models.

## Replacements

It is crucial for assertion based testing mode that TestComponents in general as well as the SUT in grey box testing mode can be instrumented for testing purposes, since all checks as well as all observations in the TestCases are based on assertion instrumentation. Since TestConductor must never modify model elements in the user model, assertion instrumentation must only affect testing artifacts. Thus, whenever possible, TestArchitecture creation introduces TestComponents inheriting from the design model elements for instrumentation. In Rhapsody in C, inheritance is restricted to inheritance from interfaces. For all other model elements, no inheriting TestComponents can be used. For Rhapsody in C++, inheritance is restricted to virtual and abstract operations. A non-virtual operation can not be overridden by an inheriting TestComponent. Thus, for all these cases, inheriting TestComponents can not be used to realize driver operations and stubs or observation instrumentation without affecting the original model elements in the user model.

If inheriting TestComponents can not be used, replacing the original model element in the scope of the code generation with a copy of the model element and instrumenting the copy for testing purposes solves the problem. TestArchitecture creation for assertion based testing mode makes use of replacements whenever inheritance is inappropriate for creation of TestComponents.

All references to a replaced model element in the TestArchitecture, such as instances, links, connectors, etc. refer to the replaced model element and also the TestScenarios of sequence diagrams refer to the original model elements<sup>3</sup>. Identification of replacements for referenced model elements is based on stereotyped dependencies, e.g. a TestComponentInstance referring to model class A is equipped with a `<<use_replacement>>` dependency on replacement class A' (a copy of A in the TestArchitecture). Replacement A' is equipped with a `<<replacement>>` dependency on A.

---

<sup>3</sup>Except for replacements of files (modules). For TestFile – replacement of file – the TestScenarios refer to the TestFile instead of the replaced file.



In the code generation scope, class A is deselected, whereas A' belongs to the scope instead of A. It is important to understand that a replacement always must have the same name as the model element replaced by it.

Creation of replacements depends on property

`TestConductor.Settings.ReplacementCreationMode`. If the property is set to 'Wrapper' (Default) then the replacement will be created as '*identical*' copy of the replaced model element, which means the replacement will preserve most of the behavior of the replaced class. If the property is set to 'Stub' then e.g. operation bodies in the replacement will be emptied and the original behavior of the replaced class will not be preserved.

Copying or cloning, respectively, of classes and objects does not always yield an 'identical' clone of the original class or object:

- Bidirectional associations: since bidirectional associations consist of both ends and cloning one side of the bidirectional association would require the other side to be related the original associated element as well as to the cloned associated element. Thus, on the cloned side of the bidirectional association, the association becomes a directed one. Directed associations are initialized differently from bidirectional associations. To solve this problem, TestConductor adds appropriate functions to the cloned class or object, enabling initialization of the relation as if it was a bidirectional one.
- Composite classes and objects: relation 'Knows parent as' of parts becomes uninitialized in the clone. This mainly affects composite grey box SUT clones, i.e. clones of classes with parts in SUT position for grey box testing. In order to fix this issue, either appropriate initialization code has to be added to the initializer of the TestContext, or in a TestAction in each TestCase for the grey box architecture (cf. GreyBox Architecture in `Samples/CSamples/TestConductor/TestingCookbook/CCompositeCoffeeMachine_wo_ports_objects`, `Samples/CppSamples/TestConductor/CppCompositeCoffeeMachine_wo_ports`).

Since TestArchitecture update does not affect code generation component and configuration, it might be necessary to adapt the scope of the code generation component manually after either manually adding or removing replacements or after TestArchitecture update.

(See also TestingCookbook:

- “How can I create a greybox test architecture with multiple SUT classes?”

)

### Dependencies used for Navigation on Replacements

The following table gives an overview about the dependency stereotypes being used for navigation on replacements.

Dependencies used for navigation on replacements			
ArchitectureUsingGlobalObjects==False			
TestComponent replacement of class A	type of TestComponentInstanc e itsA is A	TestComponentInstanc e itsA has <<use_replacement>> dependency on replacement	Replacement TestComponent A' has <<replacement>> dependency on class A

		TestComponent A'	
TestSUT replacement of class A ( <i>greybox testing</i> )	type of SUT itsA is A	SUT itsA has <<use_greyboxreplacement>> dependency on replacement TestSUT A'	Replacement TestSUT A' has <<greybox_replacement>> dependency on class A
TestFile replacement of file X	type of TestFile X' is implicit	TestContext has <<use_filereplacement>> dependency on TestFile X'	Replacement TestFile X' has <<filereplacement>> dependency on file X
ArchitectureUsingGlobalObjects==True			
TestComponentObject replacement of implicit object O	type of TestComponentObject O' is implicit	TestContext has <<use_instancereplacement>> dependency on TestComponentObject O'	TestComponentObject O' has <<instancereplacement>> dependency on implicit object O
TestSUTObject replacement of implicit object O ( <i>greybox testing</i> )	type of TestSUTObject O' is implicit	TestContext has <<use_greyboxinstancereplacement>> dependency on TestSUTObject O'	TestSUTObject O' has <<greyboxinstancereplacement>> dependency on implicit object O
TestComponent replacement of class A	type of TestComponentObject itsA is A	TestComponentObject itsA has <<use_replacement>> dependency on replacement TestComponent A'	Replacement TestComponent A' has <<replacement>> dependency on class A
TestSUT Replacement of class A ( <i>greybox testing</i> )	type of TestSUTObject itsA is A	TestSUTObject itsA has <<use_greyboxreplacement>> dependency on replacement TestSUT A'	Replacement TestSUT A' has <<greybox_replacement>> dependency on class A
TestFile Replacement of file X	type of TestFile X' is implicit	TestContext has <<use_filereplacement>> dependency on TestFile X'	Replacement TestFile X' has <<filereplacement>> dependency on file X
Table 1: Dependencies used for navigation on replacements			

The dependencies allow TestConductor to instrument A' whenever test artifacts, such as driver operations or stubs, would have to be added to A.

For replacements, it can be preselected whether stubs or wrapper will be created (property `TestConductor.Settings.ReplacementCreationMode = {Wrapper, Stub}`, Wrapper is the default).

## Interfaces

Interfaces are basically abstract classes. Implementations for interfaces have to be provided by inheriting classes. For Rhapsody in C, code generation supports inheritance from interfaces by virtualization tables, for Rhapsody in C++ the code generation is straight forward.

TestArchitecture creation creates inheriting TestComponents for relations of the SUT with interfaces.

Interfaces can not be selected as SUT for TestArchitecture creation.

## Ports

(See also TestingCookbook:

- “How can I create a test architecture for a class using ports?”

)

TestConductor supports TestArchitecture creation for classes and objects using ports. For each port of the SUT, a TestComponent is created with a suitable corresponding port, i.e. providing the required contracts and requiring the provided contracts of the SUT port.

Note, that for each port of the SUT a separate TestComponent will be created which might cause problems when using the TestCase Wizard for creation of TestCases for existing sequence diagrams (cf. section Creating TestCases with the TestCase wizard, page 52 ff.).

The TestArchitecture can be modified manually, s.t. TestComponents for multiple ports are merged from the separated ones and ports are connected according to the capabilities of the merged TestComponents. TestArchitecture update will keep existing links and not introduce separate TestComponents for existing port connections.

TestArchitecture creation 'realizes' provided contracts, i.e. adds reception declarations and operations to the TestComponents created for SUT relations to ports. These realizations are required for message realizations in TestScenarios of SD TestCases. It might sometimes be necessary to add realizations – in particular event receptions – manually to be able to realize messages in TestScenarios. Especially for so-called rapid ports (ports without contracts – aimed at receiving and sending events), TestArchitecture creation and update can not automatically add all desired event receptions to realize the implicit port contract.

## VariationPoints and Variants

(See also TestingCookbook:

- “How can I test models using variation points?”

)

TestConductor supports relations of the SUT to VariationPoints by creating <<Variant>> TestComponents. On creation of the TestComponent, TestConductor adds a <<Static>> generalization relation to the TestComponent and a <<Varies>> dependency on the VariationPoint, s.t. the <<Variant>> TestComponent becomes a valid implementation of the VariationPoint.

Furthermore, TestArchitecture creation adds a VariationPoint mapping to the <<Variant>> TestComponent to the code generation component of the TestArchitecture. Since code generation component and configuration are not affected by TestArchitecture update, it might be necessary to adapt these mappings manually after TestArchitecture update.

Note, that stereotypes <<VariationPoint>> and <<Variant>> are applicable to Rhapsody meta class Class. Hence, they can also be applied on files in Rhapsody in C (modules) and on implicit objects, since files and implicit objects own their implicit class. But Rhapsody does not allow inheritance from implicit objects. Thus, <<Variant>> TestComponentObject or <<Variant>>TestFile can not sufficiently be created automatically.

## Inheritance

General inheritance is not feasible in Rhapsody in C. Inheritance is only supported by the concepts of interfaces, ports and variation points. Therefore, TestArchitecture creation is mainly based on the concept of replacements.

For Rhapsody in C++, inheritance is fully supported. TestArchitecture creation by default creates inheriting TestComponents, whenever the class to be represented by the TestComponent is virtual. If the class is abstract, the representing TestComponent will have to implement the abstract class by inheritance. This is in particular the case for interfaces.

If the class to be represented by a TestComponent is non-virtual, a replacement will be needed in TestComponent place, since otherwise stubbing can not be performed in the TestComponent but would have to be done in the base class.

If the class to be represented by a TestComponent itself inherits non-virtually from another class, then replacements will be created along the entire inheritance hierarchy unless a virtual base class has been reached.

## Templates and Template Instances

Templates and Template Instances are not supported by automatic TestArchitecture creation and update. In order to test models using templates or template instances, TestArchitectures have to be created and maintained manually.

## Automatic TestArchitecture Generation

TestConductor offers automatic creation of TestArchitecture for a selected SUT class, object or file. By default (cf. section TestConductor Settings on page 96 ff), a TestArchitecture is created that instantiates the SUT and its testing environment artifacts as parts of the TestContext. For testing global objects, a different mode of TestArchitecture creation can be chosen, such that global objects and object replacements are instantiated in a TestPackage and the TestContext is equipped with associations to these global objects.

Even though TestArchitecture creation is aimed at fully automatically creating suitable TestArchitectures for unit testing the selected SUT elements, it might be necessary to apply manual modifications to the obtained architecture and to adapt the code generation scope manually, respectively. In principle, a TestArchitecture can be modified and modeled like any other modeling artifacts in Rhapsody. In particular, TestComponents can be used for implementing testing related additional observations, such as 'no other event was sent to the TestComponent', user-defined stub-operations for code TestCases can be implemented. Links can be established in a different way: e.g. TestConductor creates one TestComponent per port of the SUT. Sometimes it may be preferred to have more complex TestComponents serving and driving more than one of the SUT ports. Such modifications can be applied on the TestArchitecture – TestArchitecture update will keep such manual modifications and complete the TestArchitecture only with missing artifacts.

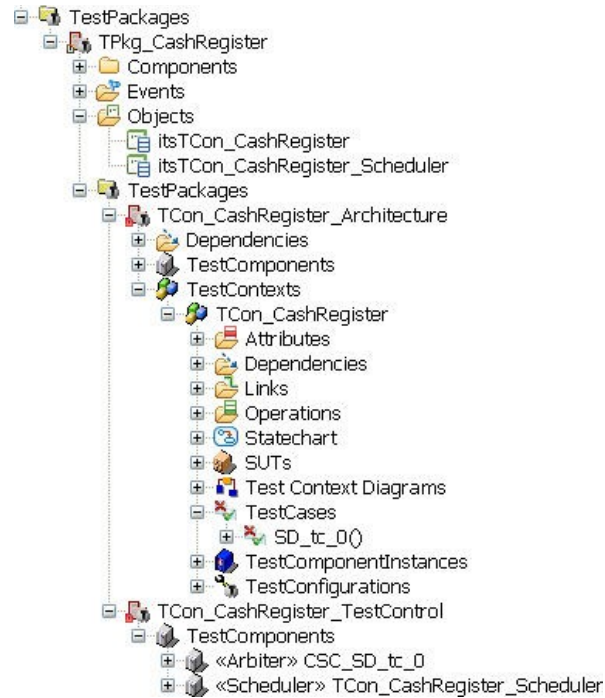
## Context Menu 'Create TestArchitecture'

By default, context menu 'Create TestArchitecture' can be invoked mainly on Class, Object, Block (SysML) and Module (i.e. file model elements) model elements – besides some other kinds of model elements which are derived from the mentioned ones.

For user defined new terms on the basis of these meta classes, the context menu will not be offered by default. By adding the respective new term to the applicableTo<nr> entry for

the corresponding name<nr>=Create TestArchitecture entry in the Rhapsody.ini file, the functionality can be made applicable to also user defined new terms.

1. The created TestPackage contains two sub TestPackages, one architecture sub package that actually contains the TestContext and the TestComponents that are connected to the SUT, and a control TestPackage that contains an auto generated scheduler TestComponent and the auto generated arbiter TestComponents that control the test execution in assertion based testing mode.



*Figure 2: TestArchitecture in Rhapsody Browser*

2. Inside the top level TestPackage, two static objects are defined. One object is an instance of the created TestContext, and one object is an instance of the created scheduler. Since the top level package is part of the scope of the TestingConfiguration that is used to generate and build code for the test executable, always a TestContext instance and a scheduler instance is defined in the test executable.
3. The configuration that is created inside the top level TestPackage is used in order to generate and build the code of the test executable. It is stereotyped with <<TestingConfiguration>>. The stereotype provides several tags that can be used to define several testing options (cf. section Tags of the <<TestingConfiguration>> Stereotype on page 60).
4. TestComponentInstances forming the environment of the SUT are either instances of so called replacement TestComponents or instances of TestComponents inheriting from the original design classes. In C inheritance is only supported from interfaces, i.e. TestComponentInstances for associations to interfaces may be represented by instances of TestComponents realizing the interfaces, whereas all other TestComponentInstances are necessarily instances of replacement TestComponents. Replacement TestComponents are derived by copying from the original design classes and *replace* these in the scope of the TestingConfiguration, i.e. are used for code-generation instead of the original classes.

In C++, usage of inheritance vs. replacement TestComponents is determined by virtuality. If all operations of a design class are virtual, then the TestComponent used for instantiation of this class can be derived by inheritance. If at least one of the member operations isn't a virtual operation, a replacement TestComponent is used for instantiating the respective TestComponentInstance.

The user can influence the way replacements are created using property  
`TestConductor.Settings.ReplacementCreationMode = {Wrapper, Stub}`

A wrapper replacement is created as a fully functional copy of the design class, whereas a stub replacement is created as a copy from which behaviors are removed, e.g. operation bodies are emptied, statecharts are emptied, etc.

If property `TestConductor.Settings.TestArchitectureCreationMode` is set to “Advanced”, the user can specify the kind of each TestComponent individually as either inheriting, wrapper replacement, or stub replacement, respectively.

TestConductor then first analyzes all connections of the SUT with its environment via ports and associations and then opens a dialog using which the user can determine how the TestComponents around the SUT will be created.

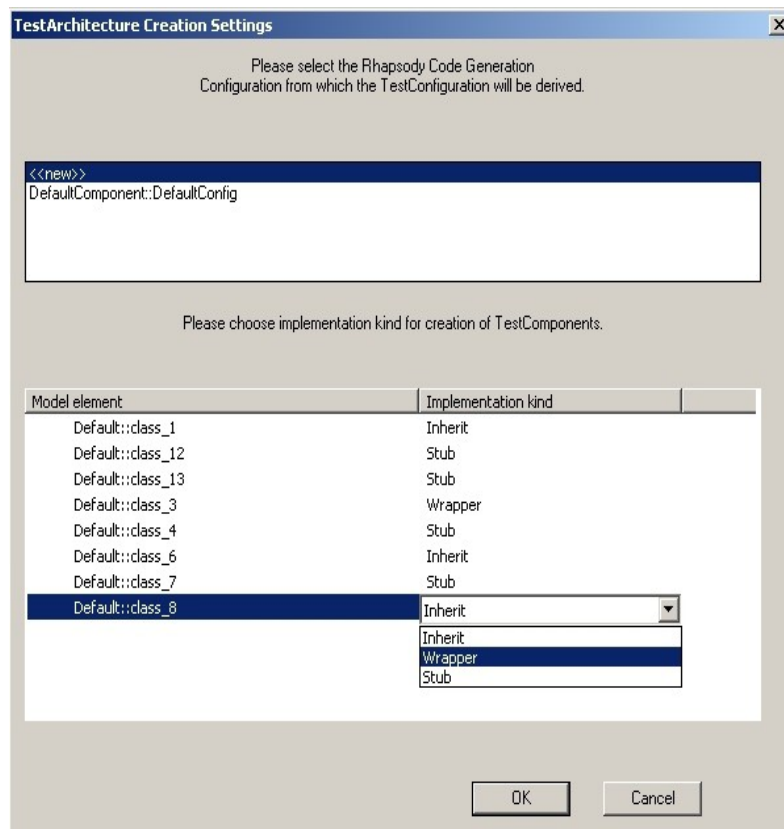
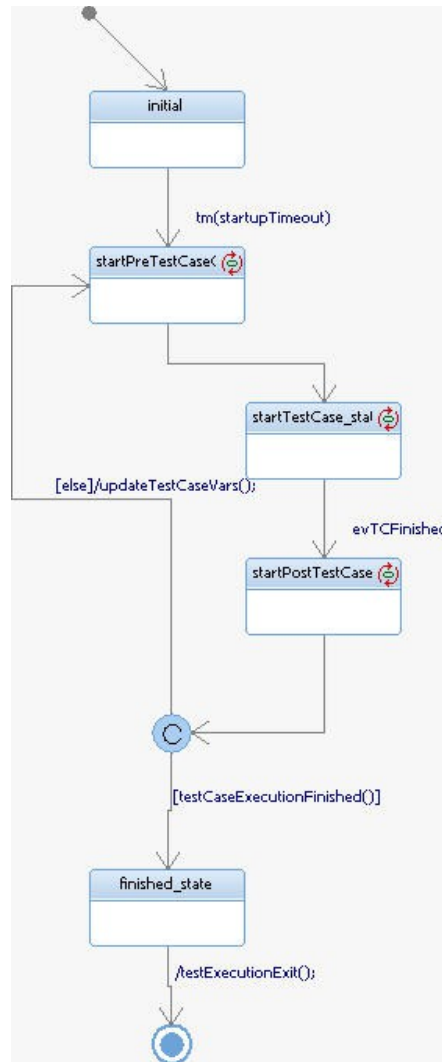


Figure 3: Advanced TestArchitecture Creation Dialog

As already explained above, for some TestComponents inheritance from the original design class is not possible, since an inheriting TestComponent would not allow the necessary stubbing of operations (due to non-virtuality or language limitations). For these TestComponents, the user can only choose between stub and wrapper.

## Test scheduling with <<Scheduler>> TestComponents

As described in the previous section, when creating a TestArchitecture, a scheduler TestComponent is generated that is used to control the starting and stopping of TestCases. The scheduler is part of the test executable. By default, the behavior of the scheduler is defined by the following statechart:



*Figure 4: TestCase Scheduler Statechart*

By default, the scheduler parses the command line when the test executable is started. Based on the specified TestCases that shall be executed, the scheduler starts the selected TestCase(s). This default behavior can be adjusted according to your needs. For instance, if you want to e.g. add an automatic timeout mechanism for all TestCases you can adjust the behavior of the scheduler as it is described in section Execution Timeout on page 77.

The TestCaseScheduler can be customized using properties `TestConductor.TestContext.PreTestCaseOperation` and `TestConductor.TestContext.PostTestCaseOperation` of the TestContext.



These properties can be set to operation-names<sup>4</sup> to be called before starting a TestCase and after termination of a TestCase, respectively.

`TestConductor.TestContext.PreTestCaseOperation` denotes an operation to be called on entering state `startPreTestCaseOperation_state`, whereas `TestConductor.TestContext.PostTestCaseOperation` denotes an operation to be called on entering state `startPostTestCaseOperation_state` – the respective TestCase is started in state `startTestCase_state` (cf. section TestContext Properties on page 101).

## Test arbitration with <<Arbiter>> TestComponents

If you define the behavior of a TestCase by using a sequence diagram, in assertion based testing TestConductor automatically adds a so-called arbiter TestComponent to the control sub package of your TestArchitecture. An arbiter is a TestComponent that contains the stereotype <<Arbiter>>. Besides the arbiter class, TestConductor also adds an instance of the arbiter class to the TestContext that contains the TestCase. During runtime, this instance is used to control the TestCase execution of the TestCase to which the arbiter belongs. The TestCase and its arbiter are connected by a dependency that contains the stereotype <<ControlArbiter>>:

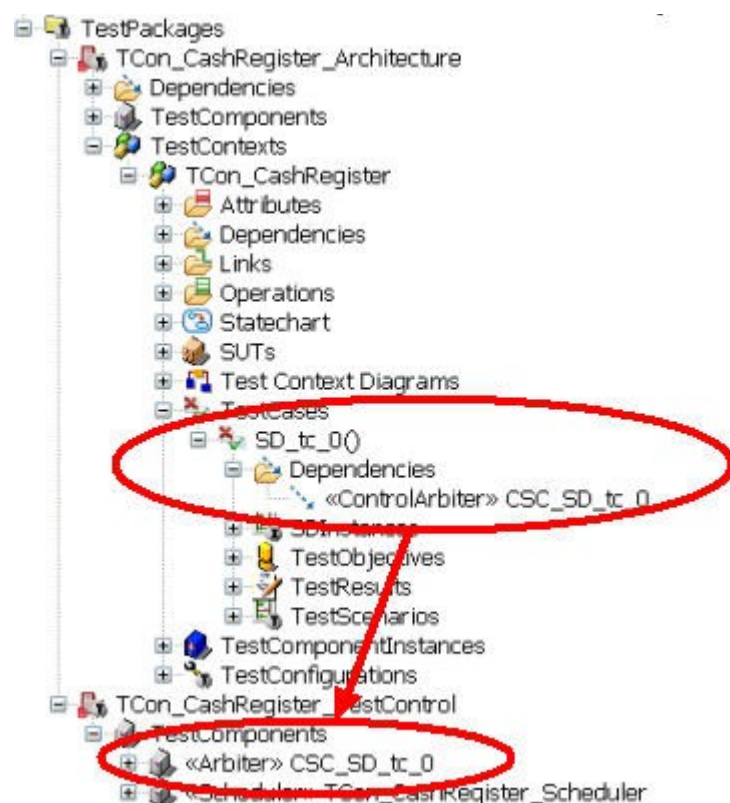


Figure 5: Arbiter of SD TestCase

**Note:** If an arbiter TestComponent has changed after an update of the corresponding sequence diagram TestCase, the TestContext owning the arbiter instance needs to be build. Also, the TestPackage owning the instance of the TestContext needs to be build. The

<sup>4</sup>For using 'void x(void)' member-operation of the TestContext, 'x' has to be specified in the property. Only void operations without arguments are supported using this mechanism.



Makefile generated by Rhapsody does not contain the necessary Makefile rule which forces building the TestPackage after a change in an arbiter TestComponent. In most cases this is not a problem, the TestPackage is build anyway after an update because of changes in other TestComponents. But if only a time interval in a sequence diagram TestCase changes the TestPackage is not build automatically. This can cause the tested application to crash during test execution.

To avoid this, after adding, removing or modifying a time interval the user should explicitly invoke a Code->Clean before building the test application again using 'Build TestCase/TestContext/TestPackage' from the context menu.

## Creating test executables with TestingConfigurations

In order to execute TestCases in assertion based testing mode, always a test executable is needed that actually contains the code for the TestArchitecture, the scheduler and all arbiters. In order to generate the code, a Rhapsody <<TestingConfiguration>> code generation configuration is created. the test executable always contains all the code that is necessary in order to execute TestCases of the TestContext that belongs to the TestingConfiguration. In particular, it is not necessary to have animation turned on for the TestingConfiguration. Both animated and non-animated configurations can be executed the same way.

## Generate and Build the TestContext

After generation of the new *TestContext* you should check whether it is complete and consistent. Therefore you should generate and build the TestContext to get information about potential compile or link warning or errors.

Right-click on the TestContext and select **Build TestContext** from the context menu.

If the generate, compile and link procedure are resulting in an executable you are able to execute it (a TestContext without any TestCase cannot be executed).

## Using Classes (UML) and Blocks (SysML)

(See also TestConductor Tutorial for Rhapsody in C and Rhapsody in C++.

See also TestingCookbook:

- “How can I create a test architecture for a class using ports?”
- “How can I test models using variation points?”)

Creating TestArchitectures for single individual classes (Rhapsody in C and C++ - as well as for blocks in Rhapsody for SysML models) is the standard TestArchitecture creation for unit testing considered throughout this entire section. TestArchitecture creation is tailored to support particular features of classes in a specific manner, such as dedicated support for ports, associations with interfaces and variation points, e.t.c.

Variations of this standard TestArchitecture creation and influences from particular testing strategies are described on the following pages.

## Using Objects

(See also TestingCookbook:

- “How can I create a test architecture using global objects?”
- “How can I create a test architecture for a singleton object?”

)

Creating a TestArchitecture on objects is a similar work flow as for classes, but in order to create a TestArchitecture for testing an object, the object can not be directly instantiated as part of a TestContext. If an object was instantiated as part of a TestContext, the object would be moved into another scope and thus the model would be modified. Hence, in order to provide testing support for objects without modification of the original design, the TestContexts just references the object from the design using directed associations and directed links. Since by default the original (implicit) object is referenced with all its relations to other objects in the model and because TestConductor can't modify these relations without modifying the referenced object or other model elements in its scope, stubbing is not supported in TestArchitectures for objects in animation based testing mode. Stubbing is also not supported in assertion based testing of objects unless

TestArchitectures are created with specific support for global objects (checking property `TestConductor::Settings::CreateTestArchitectureUsingGlobalObjects` – cf TestConductor settings “General Properties”, page 98)

In order to refer to an object, the TestContext is created with a directed association to the selected object, which does not modify the object. This association is stereotyped with the testing profile stereotype `<<instantiated>>`.

Except for TestArchitectures created with global object support (to enable global object support for C or C++ models, check property

`TestConductor::Settings::CreateTestArchitectureUsingGlobalObjects`) `<<instantiated>>` associations are not initialized by links but the TestContext is instrumented with an additional constructor/initializer initializing the association with the address of the global variable representing the object. This constructor/initializer has to take the multiplicity of the object into account.

Except for TestArchitectures created with global object support, the TestArchitecture for objects will not care about ports of the object, since the mapping of these ports to ports of other objects may already be defined in the design. The only way to stimulate an object in a system TestArchitecture is to use the association from the TestContext to the object.

Rhapsody offers an alternative to create a TestArchitecture on a selected object. The user can expose the class of the selected object. For Rhapsody in C++ this alternative will set the user into the position of applying unit tests to the underlying class of the object under test. **For Rhapsody in C, in general, exposing an object's class might not be the best choice, because exposing an object's class massively affects the code representation of the object's functions.**

TestConductor optionally supports to create a TestArchitecture suitable for unit testing of an object (or a class associated with an object), including automatic generation of driver or stub code also in implicit objects. To create such a TestArchitecture, open the TestConductor main dialog (Rhapsody menu Tools->TestConductor) and set option “Architecture using global objects” to True (or check property `TestConductor::Settings::CreateTestArchitectureUsingGlobalObjects` for the project) before creating a TestArchitecture. An example can be found in the TestConductor testing cookbook, section “How can I create a TestArchitecture using global objects?”.

## Using Files (Modules)

(See also TestingCookbook:

- “How can I test an external file?”
- “How can I test an external library?”

)

Creating a TestArchitecture on files(to be more precise: modules) is a similar work flow as shown for objects. Support of modules is useful mostly for Rhapsody in C, since Rhapsody in C++ only allows external files within the scope of a CG component. Since modules provide global declarations and definitions, test support for modules is realized by a TestContext referring the module using a <<Usage>> dependency.

The declaration of external (source and library) files and testing with TestConductor is demonstrated in the TestingCookbook (“How can I test an external file?”).

TestConductor offers special support of <<CInterfaceFile>> stereotype in AssertionBased TestingMode. Interfacing files using the <<CInterfaceFile>> stereotype is briefly explained in chapter Support for interfacing Files in C using <<CInterfaceFile>> Stereotype at page 45.

## Using Parts of composite classes

Only global (i.e. top-level) objects may be tested. There will be no support for testing parts (with implicit class) of composite classes.

If the part is an instance of an explicit class, then a TestArchitecture for that class can be created and the class can be tested using this TestArchitecture.

## GreyBox TestArchitectures for classes and objects

(See also TestingCookbook:

- “How can I create a greybox test architecture with multiple SUT classes?”
- “How can I observe the communication between parts of a greybox SUT?”

)

By default, TestConductor doesn't support observation of self-messages of the SUT or messages among SUTs in assertion based testing mode. In assertion based testing mode, TestConductor instruments the TestComponents and the TestContext before code generation with assertions, checking the values of operation arguments, the return values from operation calls, the order of messages between TestComponents and the SUT e.t.c.

Since the SUT itself is not instrumented, TestConductor does not regard messages from SUT to SUT, i.e. self-messages of the SUT are ignored by default. In order to enable also observation of self-messages, TestConductor can create a special variant of TestArchitectures for so-called GreyBox testing. Observability of SUT internals is restricted to operations and event receptions of the SUT class and its parts. Observability of operations and event receptions of parts of the SUT is limited to one level of

decomposition, i.e. internals of parts of parts are not observable in GreyBox testing (see also section Grey Box Testing on page 40).

When property

'TestConductor.Settings.CreateTestArchitectureTransparency' is set to 'GreyBox' (default is 'BlackBox'), the SUT Class is copied to the TestArchitecture as <<TestSUT>>. This copy is then used for testing instead of the original class (the <<TestSUT>> copy is equipped with a <<greyboxreplacement>> dependency on the original class). TestConductor will instrument the <<TestSUT>> according to the needs of the individual TestCases. The SUT instance is equipped with a <<use\_greyboxreplacement>> dependency on the <<TestSUT>>. The scope of the CG-component of the TestArchitecture is computed s.t. the <<TestSUT>> replaces the original class in the CG scope. If the SUT is decomposed into parts, then classes of the parts will be treated similarly: <<TestSUT>> copies of the part classes will be created in the TestArchitecture for testing purposes and these copies will be equipped with appropriate <<greyboxreplacement>> dependencies on the original classes in the design and the parts of the SUT copy will be enriched with <<user\_greyboxreplacement>> dependencies on the <<TestSUT>> copies of their classes. To distinguish <<TestSUT>> copies of part classes from the <<TestSUT>> copy of the SUT, TestArchitecture creation annotates <<TestSUT>> copies of classes with a tag `decomplevel`. For part classes the tag is assigned value 0 whereas it is assigned a value >0 for the <<TestSUT>> copies of the parent classes, i.e. the SUT classes.

Property

'TestConductor.Settings.CreateTestArchitectureTransparency' can either be set in the features dialog on project level or in TestConductor's main dialog (via menu Tools->TestConductor).

**Note:** Changes in the original design class, such as modified operation bodies e.t.c., will be regarded in the TestArchitecture only after explicit 'Update TestArchitecture'. Update of particular model elements such as statechart, operations, attributes or the entire class can be inhibited by stereotyping that model element by <<Stub>> in the <<TestSUT>> copy. TestArchitecture update will omit updating model elements stereotyped <<Stub>>.

**Note:** GreyBox testing of files is not supported.

## TestArchitectures with multiple SUT classes or objects

(See also TestingCookbook:

- “How can I create a test architecture with multiple SUT classes and/or instances?”
- “How can I create a greybox test architecture with multiple SUT classes?”
- “How can I create a test architecture for a Package with multiple classes?”

)

TestArchitectures with more than one SUT class or object can simply be created by first creating a TestArchitecture for one of the classes or objects to be tested and successively

adding further SUT instances. TestArchitecture Update can be used to automatically complete the TestArchitecture with TestComponents and TestComponentObjects.

Creating TestArchitectures for more than one class or object will in general be an at least partially manual task, since the SUT elements have to be connected accordingly and the code generation scope has to be manually adapted according to the involved model elements.

For black box TestArchitectures an iterative approach of TestArchitecture creation, removal of TestComponents, addition of further SUT elements, appropriate connection of SUT elements and TestArchitecture updates can easily performed using Rhapsody modeling capabilities and the context menu helpers in the Rhapsody browser.

For adding SUT replacements in grey box testing, TestConductor provides the dedicated helper “*Create Greybox SUT*” (see Testing Cookbook for example usage).

The testing cookbook provides examples e.g. answering the questions “How can I create a TestArchitecture with multiple SUT classes and/or instances?”, “How can I create a TestArchitecture for a Package with multiple classes?” and “How can I create a greybox TestArchitecture with multiple SUT classes?”

## Updating TestArchitectures

TestArchitecture creation generates an appropriate test environment for the SUT in its state of development in a particular instant of time. When the model is further developed, functions of the SUT and its environment may change their signature, interfaces and ports may be added or deleted, relations may be added and deleted, etc. Whenever such modifications took place, the TestArchitecture needs to be adapted to the modified model. For existing TestArchitectures, TestConductor provides the possibility to automatically update a TestArchitecture after changes have been made in the model.

'Update TestArchitecture' is offered as context menu entry on TestContexts.

'Update TestArchitecture' follows the same rules as TestArchitecture creation and will complete the existing TestArchitecture with appropriate TestComponents for added relations and update TestComponents w.r.t. modified relations and interfaces of the SUT. Since TestArchitecture avoids deleting model elements that may contain user changes – such as e.g. existing operation bodies. Furthermore, TestArchitecture update will not affect the scope selection in the code generation component. Hence, it might become necessary to manually adapt the scope selection and to manually delete artifacts in the TestArchitecture, which have become superfluous due to modifications of the model. It is in general recommended to update the TestArchitecture after modifications of the SUT in order to keep track of the changes in the TestArchitecture.

## Up-to-date check for TestArchitectures

TestConductor offers a context menu entry on TestContext “Check if TestArchitecture is up-to-date”. Using this context menu item it can be checked whether “Update TestArchitecture” will apply changes to an existing TestArchitecture or if the TestArchitecture is up-to-date.

## TestArchitectures for MicroC Models

TestConductor supports testing of MicroC models with a specifically tailored TestArchitecture generation.

Per default TestConductor restricts code generation component for the generated TestArchitecture such that all design packages but only the TestPackage containing the architecture belong to its scope. Setting property `TestConductor::Settings::CreateTestArchitectureMode` to 'Advanced' allows inheritance of overridden properties from an already existing configuration

Since code generation for MicroC does not regard initialization settings of the configuration, i.e. no initial instance selection, TestConductor explicitly creates an object of the TestContext.

The MicroC profile provides two different initialization modes: 'CompileTime' and 'RunTime'. While 'RunTime' is like normal initialization for C models which requires no specific support by TestConductor, 'CompileTime' influences a set of model elements, such as e.g. accessibility of associations. In particular, this affects the generated initializers of TestContexts for objects (cf. TestArchitecture creation "Using Objects", page 33). Consequently, TestArchitectures generated for initialization mode 'RunTime' are in general not compilable with 'CompileTime' initialization and vice versa. Note, that this also affects the initializer of TestComponents generated for statechart TestCases (cf. TestCase Definition with Statecharts, page 43 ff). It is, hence, strictly recommended to check the initialization mode defined for the project before creation of a TestArchitecture and to check the initialization mode defined for the referenced configuration before creation of the first statechart TestCase.

Assertion based testing mode relies on code instrumentation according to the needs of the individual TestCases. In order to leave the original design elements untouched, TestConductor creates copies of the classes associated with the SUT class. The scope of the CG component is then adjusted, s.t. the copied classes are used instead of the original design classes when building the test application – these copies are called *replacements*. MicroC CG for CompileTime does not properly initialize objects or parts of these replacements. The relation initialization for associations is not generated consistently for links among replacements or between SUT and replacement TestComponentInstances. In particular, outgoing associations of replacement classes are not initialized in MicroC CompileTime TestArchitectures. This affects the ability to drive the SUT with messages from replacement TestComponentInstances, whereas stubbing and invocation of operations of TestComponents by the SUT works as expected.

In order to enable also driving the SUT by replacement TestComponentInstances, it is recommended to set property

`'C.CG.Configuration.AllCategoriesInitializingMode'` to

`'ByCategory'` and property

`'C.CG.Configuration.RelationInitializingMode'` to `'RunTime'`.

For greybox Testing (cf. section Grey Box Testing on page 40) this setting of properties is required.

## TestArchitectures for Code centric Models

(See also TestingCookbook:

- "How can I test applications developed with Rhapsody Architect for Software?"

)

For code centric Rhapsody models, the source code of the SUT is compiled to a library and the executable with the test harness is linking this library. The code of the SUT library is not instrumented with animation code and it is built with the code centric property settings while the test harness contains animation instrumentation.

For the SUT library, it is possible to choose an already existing library of the project or TestConductor can automatically create a new library CG Component.

When invoking “Create TestArchitecture” on a class in a code centric Rhapsody model, a dialog appears with the options to select an existing library CG Configuration or to create a new library CG Component and Configuration for the SUT. If an existing CG Configuration is selected, a TestArchitecture is created with another CG Component and Configuration for the generation and compilation of the test harness. This CG Configuration has some properties enabled which are usually disabled in the code centric profile, for example properties “CG::Relation::AddGenerate” and “CG::Relation::SetGenerate” are enabled and “CG::Configuration::MainGenerationScheme” is set to “Full”. The scope of the newly created CG Component contains only the test harness and it has a “Usage” dependency to the CG Component of the SUT, making sure the needed header files and the library of the SUT can be found.

If creation of a new CG Component for the SUT library is chosen, then TestConductor creates two CG Components in the TestArchitecture: First a library CG Component “libSUT” with the scope set to the SUT class and its associations and the default property settings of the project and second an executable CG Component for the test harness.

After creating the TestArchitecture, the user should revise the settings of the newly created CG Components and Configurations. It might be necessary for example to add more model elements to the scope of the CG Components or to modify the options for the “Additional Sources”, “Include Path” etc. The user has to build the SUT library; for the CG Configuration “RadioLib::RadioDebug” this can be done by executing the shell script “buildLib.sh” (located on the project folder) in a cygwin shell. The executable of the test harness can be built using the TestConductor menu functions “Build TestCase”, “Build TestContext” or “Build TestPackage”.

The TestArchitecture for code centric models can be used the same way as TestArchitectures for non code centric models, with some restrictions because of the not animated SUT (internal communication of the SUT cannot be observed).

## Production Code (Black Box) Testing

Production code or *black box testing* means that the internal behavior of the SUT can not be observed by TestConductor. The objective is to test the interface behavior of a SUT.

**Note:** You can use the same TestCases defined for white box testing. In case of black box testing TestConductor ignores all messages which communicate between SUT objects. Only the input and output messages are observed.

## Black Box Testing

By default, in assertion based testing mode, internals of the SUT such as self-messages aren't observable for TestConductor. TestConductor only instruments TestComponents with assertions and, thus, doesn't regard self-messages of the SUT.



In order to also regard such self-messages for debugging purposes or in early phases of test application, TestConductor supports Grey Box testing with a dedicated TestArchitecture creation (cf. Grey Box Testing on page 40 and GreyBox TestArchitectures for classes and objects on page 35).

For using assertion based testing mode with MicroC models cf. section TestArchitectures for MicroC Models on page 38)

## Grey Box Testing

In assertion based testing mode, observability of messages is established by automatically instrumenting the code with assertions. By default, this instrumentation is limited to the test harness, i.e. TestConductor will not instrument the SUT itself, but only the TestComponents that form the test environment of the SUT. Thus, by default, internals of the SUT are not observable by TestConductor in assertion based testing mode, self-messages of the SUT are ignored, the SUT is treated as black box.

Sometimes, especially for debugging purposes and in early phases of testing, it may be desired, to regard also internal messages of the SUT in the TestCases. For this use case, TestConductor supports a special variant of TestArchitecture creation. If property 'TestConductor.Settings.CreateTestArchitectureTransparency' is set to 'GreyBox' (default 'BlackBox') on project level, the class to be tested is copied to the TestArchitecture and the copy is used for the testing activities instead of the original class. This allows TestConductor to instrument the copy according to the needs of the TestCases, without modification of the original class. Property 'TestConductor.Settings.CreateTestArchitectureTransparency' can either be set in the features dialog on project level or in the TestConductor main dialog (available via Tools->TestConductor).

Observability of self-messages is restricted to operations of the SUT class itself, messages among parts of the SUT are not observable for GreyBox testing.

Note that Grey Box testing tests a copy of the SUT. Hence, the TestArchitecture does not automatically keep track of modifications to the SUT. It is strictly recommended apply 'Update TestArchitecture' (cf. section Updating TestArchitectures) after modifying the SUT in the model.

Testing a copy of the SUT instead of the original classes involves dealing with model representation and code generation subtleties:

- copying a class associated with other classes via bi-directional associations breaks the bi-directionality of the associations of the copy: associations in the copied class to associated elements become directed, since the associated classes still refer to the original class. TestConductor therefore creates a set of functions needed for initialization and cleanup of association instantiation, e.g. by links. Normally, Rhapsody code generation auto generates such functions for bi-directional associations. TestConductor aims at reproducing this functionality but may fail with respect to disregarded properties affecting generate code.
- copying a class disregards/omits dependencies owned by elements outside the class but referring to elements belonging to the original class as source<sup>5</sup>.

---

<sup>5</sup>dependencies on requirements are intentionally removed from the copy in order to not confusing RequirementCoverage measurement and reporting tools. RequirementCoverage measurement instead regards the <<greyboxreplacement>> and <<greyboxinstancereplacement>> dependencies of the copy on the original element in order to consider the dependencies on requirements in the original model elements.



- for composite GreyBox SUTs, code generation does not initialize 'Knows parent as' relations of parts with the composite class, since these relations are valid only among the original classes but are not valid among the replacement copies. Such relations have to be established either in a TestContext initialization function (e.g. initializer/constructor) or in the individual TestCases, for example by invoking an appropriate routine in a TestAction.
- Rhapsody code generation for Rhapsody in C does not always generate all necessary include statements into the code of a composite class instantiating a class with ports - when the instantiated class is replaced by a copy (cf section Replacements on page 24). Normally, for each port of a class an additional file is generated, but the formal classifier of the instance is deselected from the code generation scope, since the classifier has been replaced by a copy of it. In order to avoid compiler warnings (and possible errors), the missing includes can be added to property C\_CG.Class.ImpIncludes or C\_CG.Class.SpecIncludes, respectively.
- since TestConductor has instrument event processing in order to establish observability of event consumption, GreyBox testing can not deal with non-standard event consumption<sup>6</sup>.
- CodeCoverage measurement (cf. section Computing Code Coverage on page 85) is not meaningful for GreyBox testing, since the SUT copies are instrumented for testing purposes and CodeCoverage would also consider instrumentation.

For MicroC and SMXF GreyBox Testing is supported only with limitations: Since compile-time initialized associations can not dynamically be re-initialized during run-time by default Rhapsody code generation (pointer variables in the class-representing struct are declared with `const` modifier), replacement technique does not initialize bi-directional associations sufficiently. For MicroC, this can be workarounded by setting framework-properties, s.t. relations are run-time initialized in the GreyBox TestArchitecture. TestArchitecture Creation and Update issue an adequate warning:

```
WARNING: TestArchitecture uses TestComponent-replacements with associations to
other classes.
It is recommended to set property
'C_CG.Configuration.AllCategoriesInitializingMode' of the CG-configuration to
'ByCategory' and
'C_CG.Configuration.RelationInitializingMode' to 'RunTime'.
Otherwise, the associations will not be initialized properly.
If TestComponent-replacements are aimed at receiving events, then also property
'C_CG.Configuration.FrameworkInitializingMode' should be set to 'RunTime'.
```

Using this workaround, GreyBox testing can also applied for MicroC models.

Since SMXF does not support run-time initialization at all, this workaround does not exist for SMXF. Therefore, TestConductor doesn't fully support GreyBox Testing for SMXF.

---

<sup>6</sup>For RiC++, TestConductor overrides OMReactive::processEvent() locally with an own implementation. At the end of this local implementation OMReactive::processEvent() is invoked. This conflicts with user-defined solutions overriding OMReactive::processEvent() locally.

For RiC including MicroC a similar approach is implemented using the properties C\_CG.Framework.OverrideReactiveConsumeEventOperation and C\_CG.Framework.ReactiveConsumeEventOperationName. This conflicts with user-defined solutions using these properties.

For SMXF the TestConductor approach is based on using property C\_CG.Class.RootStateDispatchEventBeginCode. Also this policy conflicts with user defined modifications of this property.

In any case, it is recommended to compare GreyBox testing results with corresponding BlackBox TestCases for establishing evidence for validity of GreyBox testing results.

Hence, GreyBox testing should be seen as a test development strategy and as a debugging aid, rather than a reliable stand-alone testing policy.

## TestCase Definition

For the generated TestContext, individual TestCases can be defined. TestConductor supports four possible ways to define TestCases:

- TestCase definition with code
- TestCase definition via flow charts
- TestCase definition via statecharts
- TestCase definition via sequence diagrams

### TestCase Definition with Code

Probably the most common way to test units today is writing TestCases in the same language than the application is written.

With Rhapsody and TestConductor it is also possible to write TestCases manually, because TestCases are stereotyped operations of a TestContext.

### Defining a Code TestCase

The creation of a new TestCase is nearly the same than creation of a new operation:

- Right-click on the TestContext and select **Create Code TestCase**
- A Code TestCase is created and its **features** dialog automatically opens.
- The Code TestCase is created with a predefined body consisting of a comment and an initial assertion. The body can be edited in the implementation tab of the features dialog.

**Note:** TestConductor provides several `RTC_ASSERT` macro types, which can be used to define assertions within TestCases. A detailed description of these macros can be found in the chapter TestConductor Assert Macro on page 156.

### Testing reactive behavior with Code TestCases

Since code TestCases are basically operations of a TestContext, testing reactive behavior, i.e. reaction to events, can not be done without modifications to the TestContext. Operations can't wait on events so the *TextContext* has to be made an *active object* and thus execute in a separate thread. Now, the thread executing the TestContext can be delayed unless the SUT has reacted to an event.

- Example code in C++:  
`itsClass_0.GEN(evX());`

```

OXFTDelay(1000);
RTC_ASSERT_NAME("reaction", itsClass_0.IS_IN(reaction_state))
;

```

- Example code in C:  

```

RiCGEN(&(me->itsClass_0), evX());
RiCOXFDelay(1000);
RiCIS_IN(&(me->itsClass_0), reaction_state);

```

## TestCase Definition with Flow Charts

A graphical way to describe TestCases is by using flow charts. Since TestCases are special operations of a TestContext you can use flow charts. Flow charts can be used to define the behavior of operations with Rhapsody.

### Defining a Flow Chart TestCase

- Right-click on the TestContext `r` and select **Create FlowChart TestCase**
- The FlowChart TestCase is created and the graphical FlowChart editor opens with the automatically predefined FlowChart specification of the newly created TestCase.

### Testing reactive behavior with Flow Chart TestCases

Since flow chart TestCases are basically operations of a TestContext, testing reactive behavior, i.e. reaction to events, can be done with the same techniques as applied for Code TestCases (cf. page 42).

## TestCase Definition with Statecharts

TestCases can also be defined using statecharts. Due to the ability of statecharts to wait on timeouts, statechart TestCases are particularly suited for testing reactive behavior.

In order to separate TestCase behavior from possible reactive behavior of the TestContext, statechart TestCases are defined using specialized TestComponents, which are then dynamically instantiated for test execution.

Note that the statechart TestCase can be used easily to stimulate reactive behavior in the SUT, but that in general the SUT will react but not respond to the stimuli, i.e. since the SUT has in general no relation with the stimulating TestComponent, the SUT will not send events to this TestComponent. Observation of reactions on stimuli thus may often be only achieved indirectly or require manual modifications of the TestArchitecture.

Statechart TestCases are comprised of the following model elements:

- a TestCase, i.e. basically an operation of the TestContext.
- a TestComponent owning the statechart defining the TestCase behavior.
- a dependency of the TestCase on the TestComponent. This dependency is stereotyped `<<StatechartTestCase>>`.

### Defining a Statechart TestCase

- Right-click on the TestContext and select **Create Statechart TestCase**

- A TestCase operation for starting the statechart TestCase is created in the TestContext
- A new <<SCArbiter>>TestComponent with a predefined statechart is created and the graphical statechart editor is opened with the predefined statechart.

Creation of a statechart TestCase adds an operation ('new termed' TestCase) to the TestContext. This TestCase has a dependency on a newly created <<SCArbiter>>TestComponent owning the statechart. The TestComponent has a directed association to the TestContext, which can be used to refer to parts, variables and operations of the TestContext. Upon 'Update TestCase', a <<SCTCInstance>>TestComponentInstance instantiating the TestComponent is added to the TestContext. 'Update TestCase' also instruments the TestCase operation with initialization of required associations and with sending an evTCStart event to the TestComponentInstance to start statechart execution, i.e. trigger its initial transition..

Furthermore, the TestContext needs to be populated with a *rtc\_init()* and a *rtc\_exit()* operation which are invoked by the statechart. This population is initiated by “Update TestCase”, “Update TestContext”, and “Update TestPackage”, respectively.

## TestCase Definition with Sequence Diagrams

Another option to define TestCases is by using sequence diagrams. In the context of the Rhapsody Testing Profile such sequence diagrams are stereotyped (*new termed*) <<TestScenario>>. TestScenarios play a dominant role in the TestConductor test process. They are the graphical means of specifying and defining the tests, and enable TestConductor to visualize design flaws.

Detailed information regarding the usage of the powerful features of sequence diagram TestCases are described in chapter Specifying Requirements with Sequence Diagrams on page 108 ff.

Detailed information on generation of DriverOperations and StubOperations can be found in section Model Population – Create Driver Operations and StubOperations on page 48 and in section Influencing DriverOperation and StubOperation Generation on page 119).

By default, TestConductor ignores self-messages of the SUT specified in TestScenarios. In order to test also self-messages of the SUT, Grey Box Testing, cf. page 40, can be applied.

## Defining a Sequence Diagram TestCase

- Right-click on the TestContext and select **Create SD TestCase**
- An operation ('new termed' TestCase) is added to the TestContext.
- A TestScenario is created below the TestCase. The TestScenario is predefined with life-lines for all SUTs and TestComponentInstances belonging to the TestContext (and accordingly for all TestSUTObjects and TestComponentObjects for TestArchitectures using global objects).
- The graphical sequence diagram editor opens with the predefined TestScenario specification of the newly created TestCase.

Specification using TestScenarios is discussed in detail on pages 108 ff.

## Failure Analysis in Sequence Diagram TestCases

When the test is executed, the results of the individual assertions as well as the overall execution result is shown in the execution window. After execution, a witness TestScenario for the TestCase execution can be generated and inspected via 'Show As SD' from the context-menu of the execution window.

Further information about test execution and the related results is described in chapter Test Execution on page 59 (in particular: Interpretation of witness diagrams on page 73).

Further information about failure analysis can be found in chapter Failure Analysis on page 132.

## TestConductor.h, TestConductor\_C.h and TestConductor\_C.c

Since Rhapsody 7.1 the testing profile require the TestContext, TestComponents, and TestComponentInstances to include the TestConductor header file by setting property `CPP_CG.Class.ImpInclude` to `TestingConductor.h`. Additionally, TestConductor adds the path '`$(OMROOT) / ../TestConductor`' to the include-path of the code-generation component when creating a TestArchitecture.

To provide an adequate assertion support for Rhapsody in C, a similar header file is provided and the testing profile was extended, such that TestContext, TestComponents, and TestComponent instances automatically include an appropriate `TestConductor_C.h` header by setting property `C_CG.Class.ImpInclude` to `TestConductor_C.h`. In contrast to the Rhapsody in C++ solution, for Rhapsody in C also an C-Implementation file was provided, which must be linked only once. Therefore, the `TestConductor_C.c` file is included by the code generation configuration Main file (cf. property `C_CG.Configuration.ImplementationProlog` of the `<<TestingConfiguration>>` code generation configuration).

## Support for interfacing Files in C using <<CInterfaceFile>> Stereotype

Rhapsody predefines a stereotype `<<CInterfaceFile>>` in package `PredefinedTypesC`. Applying this stereotype to a file causes the code generation to just generate the declarations of the functions without implementing them. For `<<CInterfaceFile>>` *afile*, all functions are declared as *afile\_\$op*, where *\$op* is the basic name of the function. In order to use a `<<CInterfaceFile>>` file interface, a file can refer to the interface using a generalization. The inheriting file should have property `C_CG.Operation.PublicName` set to "*<afile>\_\$op*", where *<afile>* is the name of the `<<CInterfaceFile>>`. Furthermore, `<<CInterfaceFile>>` *afile* as well as the inheriting file should override `C_CG.Operation.UseProtectedNameAndPublicNameInFile` by checking the property. Now, the inheriting file defines the implementation of the functions declared by the `<<CInterfaceFile>>` *afile*. Other files that are desired to use these implementations only have to refer to the `<<CInterfaceFile>>`. This ways, a notion of interfaces can be used with files in C, declaration and implementation of functionality

can be handled separately in the model.

TestConductor offers specific support for <<CInterfaceFile>> interfaces, by stubbing the implementations if a file to be tested as SUT refers to <<CInterfaceFile>> interfaces.

## TestConductor Support for Testing Private Operations in Rhapsody in C

(see also TestingCookbook:

- “How can I access file-static (private) variables in C files?”

)

Rhapsody in C allows to set the visibility of attributes to 'private' or 'public'. For 'private' operations, code generation generates file static operations, which aren't accessible from outside the generated implementation file. In particular, such operations can not be referred to in TestCases. Hence, normally such operations can only be tested indirectly, by testing operations internally using them.

To set the tester into the position to test also private operations explicitly, TestConductor optionally creates wrapper functions for private operations in the SUT. Such wrapper function is a public operation created with the same signature as the private operation and belonging to the same unit, i.e. class, object or file.

These wrapper functions are generated to an advanced header file with name `publicwrap_<unitname>.h` and an include statement at the end of `<unitname>.c`

Example:

for some private operation `int f(int x)` of `class_A`, Rhapsody generates

```
static int class_A_f(class_A* me, int x) {  
    ...  
    return ...;  
}
```

in file `class_A.c`.

TestConductor then generates a file `publicwrap_class_A.h` containing:

```
int class_A_callprivate_f(class_A* me, int x)  
#ifdef WRAPPERIMPL  
{  
    return class_A_f(me, x);  
}  
#endif
```

To file `class_A.c` an include is appended:

```
#define WRAPPERIMPL

#include "publicwrap_class_A.h"
```

such that the public wrapper operations become part of the implementation file. Code, flow chart and statechart TestCases can now make use of these public wrapper operations to invoke the referenced static, i.e. private, operations. The public wrapper operations are not available in sequence diagram TestCases, since they are not part of the model.

**Note:** do not apply roundtripping when prompted by code generation.

The generation of public wrapper operations regards the respective properties for arguments, argument types, argument code pattern, implementation name, public and protected name, as well as the singleton stereotype suppressing me-pointer generation on singleton objects.

Although a lot of properties is regarded for generation of public wrappers for private operations, the feature may produce not compilable code under some circumstances. In such cases, stereotype <<no public wrapper>> can be applied on individual functions of the class, object or file to be tested, in order to omit public wrapper generation for the respective function.

Testing support for private operations in Rhapsody in C can be turned on using configuration's tag `PrepareForTestingPrivateOps`.

## TestConductor Support for Testing Private and Protected Operations in Rhapsody in C++

Very similar to support for testing private operations in Rhapsody in C, TestConductor also supports testing private and protected operations for Rhapsody in C++.

For C support, basically only suitable wrapper functions have to be generated into the same file<sup>7</sup> as the static operations and of course also an appropriate header has to be offered with the declarations. For C++ the class declaration itself has to be extended for providing suitable public call wrappers for private operations, since private operations can only be called by member operations of the class itself. This extension requires identification of a correct position in the class declaration in the respective header. TestConductor does neither modify the class in the model nor uses a simplifier to achieve this goal, since original model elements are never modified for testing purposes and using a simplifier for this goal would cause conflicts with other simplifier applications. Thus, class extension is simply performed on file system level and relies on certain assumptions regarding the structure of the generated code.

TestConductor aims at inserting an `#include` directive including a file containing public wrapper function declarations and definitions right before the closing curly bracket `}` of the class declaration. If TestConductor fails to identify this position in the class declaration file, comment `//TestPrivateOpsInstrumentation` can be used to force TestConductor to add the `#include` directive below this comment<sup>8</sup>.

<sup>7</sup>for which purpose TestConductor makes use of the C-Preprocessor by providing an external file and instrumenting the class implementation with an appropriate `#include` directive.

<sup>8</sup>This comment can be added to the generated file by e.g. round tripping or via `CPP_CG.WriterTemplates.ClassSpec` property.

Although a lot of code generation related properties is regarded for generation of public wrappers for private and protected operations, the feature may produce not compilable code under some circumstances. In such cases, stereotype `<<noPublicWrapper>>` can be applied on individual functions of the class or object to be tested, in order to omit public wrapper generation for the respective function.

Testing support for private operations in Rhapsody in C++ can be turned on using configuration's tag `PrepareForTestingPrivateOps`.

## Support for Rhapsody Action Language

(See also Testing Cookbook:

- “How can I test an action language model?”

)

TestConductor also supports testing of SysML models using Rhapsody Action Language.

To be able to Action Language expressions also in user defined assertions in TestCases a function `RTC_ASSERT_NAME_RAL(String name, Boolean exp)` must be provided by the user – e.g. by importing the package with the macro from the Testing Cookbook sample model using Rhapsody's Add-To-Model function (Model: `Samples/CppSamples/TestConductor/TestingCookbook/CppCompositeCoffeeMachine_RAL`).

## Model Population – Create Driver Operations and StubOperations

TestComponents are used to drive input messages of the SUT or to return specified values for operation calls. Therefore TestComponents have to be equipped with appropriate driver operations and stubs.

By using sequence diagram TestCases TestConductor automates the generation of driver operations and stubs. Automatic generation of driver operations and stubs according to the needs of the TestCases is invoked using the context menu 'Update TestCase' on TestCase, or 'Update TestContext' on TestContext, or 'Update TestPackage' on TestPackage, respectively. These menu entries start a “model population” process, analyzing all affected TestCases and generating Driver and stubs for the TestComponents according to the TestScenario specifications of the TestCases (cf. section Influencing DriverOperation and StubOperation Generation on page 119).

Besides driver operation and stub generation, “model population” populates the TestArchitecture also with other artifacts required for test execution, such as generation of `<<Arbiter>>` TestComponents, instantiation of arbiters, setting properties in various model elements in the TestArchitecture, updating operations in Scheduler and TestContext, e.t.c.

## Driver Operations

*DriverOperations* are created for messages from a TestComponent to the SUT, except for messages with checked the tag `RTC_Monitor` in stereotype `<<RTC_MsgInfo>>`, or messages starting at a life-line with the tag `RTC_Monitor` in stereotype `<<RTC_InstInfo>>`. In this case TestConductor will not drive the message.



The name of the driver operation is the concatenation of the name of the TestCase, “\_”, the name of the original operation, “\_” and a number to create a unique name. A comment is generated into the code of the driver operation that contains the *message\_id* and the name of the TestCase for which the driver operation was generated. This allows the user to identify the correct driver operation if he wants to edit it.

*message\_id* is a unique identifier that is assigned to the message by “model population” due to 'Update TestCase/TestContext/TestPackage'. The *message\_id* is stored in tag *RTC\_MsgId* of the <<RTC\_MsgInfo>> stereotype. The <<RTC\_MsgInfo>> stereotype is also applied on the message by “model population”.

The *message\_id* is also referred to by TestConductor assertions that are populated into e.g. DriverOperations and StubOperations.

The visibility of the driver operation is public to make sure this operation can be invoked by TestConductor.

The body of the driver operation consists of a call of the original operation on the SUT either directly on the destination instance via association or via port, depending on the structure of the TestArchitecture.

The values of any input argument for the driven operation call is derived from the specification in the TestScenario. The specified return-value(if specified) and the specified output argument values are stored in local variables for checking after the call.

If the sequence diagram specifies that the returned value has to be checked, the macro *RTC\_ASSERT\_SD\_NAME* is used to check if the returned value adheres to the specified return value. The same macro is used also to check out or in/out argument values (cf. 116) according to the specification TestScenario. If any of these checks fails, the entire TestCase will fail.

For information on how to customize the driver operation, see section User Defined DriverOperation on page 119.

## StubOperations

*StubOperations* are used to return specific return values for operation-calls according to the needs of TestCases as well as for checking argument values.

StubOperations are created for every message referring to an operation from SUT life-line to a life-line representing a TestComponent (except for messages with checked tag *RTC\_Monitor* in stereotype <<RTC\_MsgInfo>> or messages to a TestComponent life-line with checked tag *RTC\_Monitor* in stereotype <<RTC\_InstInfo>>).

For operations, invocation of the StubOperations for different calls (of different TestCases) are controlled by one dedicated StubbedOperation – replacing the original operation or default operation in the TestComponent.

For event messages all required stubs have to be in one operation that is invoked in event processing. For dataflows messages the setter of the associated attribute is instrumented for stubbing.

TestConductor has to determine and control the value returned by the operation for the specified messages in the TestScenario. On the other hand there might be calls of the same operation without a specified return value or the operation is called by e.g. another class on a TestComponent. Therefore, TestConductor has to generate an implementation of the

operation which provides stub returns for specified invocations but it must remain possible to call the original operation.

To ensure this, for replacement TestComponents TestConductor creates a copy of the original operation with the prefix `original_` followed by the operations name, having the same signature as the original operation. This copy of the original operation is stereotyped (new termed) `<<DefaultOperation>>`, whereas the original operation is stereotyped (new termed) `<<StubbedOperation>>`.

For inheriting TestComponents, TestConductor creates a new `<<DefaultOperation>>` in the inheriting TestComponent calling the inherited operation non-virtually.

For each occurrence of the operation in the TestScenario which has to be stubbed, a new operation is added to the TestComponent with the same signature as the original operation.

These operations are stereotyped (new termed) `<<StubOperation>>`. Each of the *StubOperations* returns the particular specified return value, out and in/out arguments for one message in the TestScenario. The name of the StubOperation is the concatenation of the name of the TestCase, the string “`_stub_`”, the name of the original operation and a number in order to make it unique.

The body of the `<<StubbedOperation>>` operation is emptied completely – on cleaning TestComponent or TestPackage (cf. sections Clean TestComponent and section Clean TestPackage on page 52 ), the body can be restored using the *DefaultOperation* - and a new *StubbedOperation* implementation is generated this way: The *StubbedOperation* determines which *StubOperation* (or the *DefaultOperation*) has to be called according to the number of the actual TestCase and according to a counter for the number of invocations of the operation.

## SD TestCases – Stubbing and Observing Operations

SUT A invoking operation f() in  
TestComponent B – operation f ()  
is stubbed:

- (1) f () is cloned to original\_f (),  
original\_f () is stereotyped  
<<DefaultOperation>>
- (2) stubbed behavior of f () is implemented  
in f () 's body and is stereotyped  
<<StubbedOperation>>
- (3) for each occurrence of f () in the  
TestScenario, a dedicated  
<<StubOperation>> is  
created in B, that checks the call  
arguments according to the  
particular occurrence and provides  
the desired return value(s).

StubbedOperation B : : f () :

```
if(current_testcase==1) {
    if(CNT_F==1) {
        return TestCase1_CNT1_f();
    }
    ...
    if(CNT_F==2) {
        error("Unexpected additional f()");
    }
} else if(current_testcase==2) {
    ...
}
...
return original_f();
```

StubOperation B : : TestCase1\_CNT1\_f () :

```
if(!<check_arguments>) {
    error("Wrong Argument");
}

<set CNT_F to next occurrence or -1 if done>

<optional: set out args according to TestScenario>

<return either specified stub value or return of
original_f() if return isn't specified>
```

Figure 6: SD TestCases - Stubbing and Observing Operations

For all messages from a SUT life line to a TestComponent life line, TestConductor creates <<StubbedOperation>> operations and *DefaultOperations* – also if no specific return values or out values of arguments have to be stubbed. Since without mandatory animation instrumentation observability can only be established using assertion instrumentation, these 'observation'-stubs are used to track messages during TestCase execution.

If

- the tag TestConductor::RTC\_MsgInfo::RTCMonitor for the sequence diagram message is set to true, or
- the tag TestConductor::RTC\_InstInfo::RTCMonitor of stereotype on the receiver life-line in the TestScenario is checked (Note that <<RTCInstInfo>> is not applied to life-lines by default)

TestConductor will not provide *StubOperations* returning specified return values or output argument values. In this case, the generated *StubbedOperation* will only serve observation purposes and the concerned messages will be monitored but not be stubbed. The *StubbedOperations* will always call their *DefaultOperations* in this case.

For further information how to customize the *StubOperation* please read the chapter Fehler: Referenz nicht gefunden at page Fehler: Referenz nicht gefunden.

## Clean TestComponent

*DriverOperations*, *StubbedOperations* and *StubOperations* can be deleted manually, but TestConductor provides the functionality to delete the automatically generated operations of a TestComponent at once. To clean a TestComponent select the TestComponent and choose from the context menu the item **Clean TestComponent**.

## Clean TestPackage

**Clean TestPackage** also deletes all results and coverage results as well as the arbiter TestComponents from the TestPackage.

To clean a TestPackage select the TestPackage and choose from the context menu the item **Clean TestPackage**.

Note, that invoking Clean TestPackage will yield slightly different results when invoked on the TPkg\_<element> TestPackage or on the contained TCon\_<element>\_Architecture package. The latter cleans only all artifacts that are referenced by elements of the TCon\_<element>\_Architecture TestPackage, whereas Clean TestPackage on the TPkg\_<element> TestPackage also removes unreferenced obsolete Testartifacts, such as <<Arbiter>> TestComponents of deleted TestCases, which will remain in the model if Clean TestPackage is invoked only on the TCon\_<element>\_Architecture TestPackage. It is therefore recommended to invoke Clean TestPackage always on the TPkg\_<element> TestPackage.

## Specifying a TestScenario

Using TestScenarios as TestCase specifications is a key concept of TestConductor.

Sequence diagrams are an intuitive graphical formalism aimed at capturing communications among communicating objects. Sequence diagrams may consist of a wide range of graphical elements, such as messages, conditions, actions, timers, Interaction Operators, e.t.c. pp.

TestConductor supports a well defined subset of these graphical elements, some of them as specialized (stereotyped) variants. In order to distinguish between the rather informal character of sequence diagrams and the formal interpretation of TestCase specifications by sequence diagrams, sequence diagrams supported as TestCase specifications are stereotyped ('new termed') <<TestScenario>>.

Specification of TestScenarios is discussed in detail in section Specifying Requirements with Sequence Diagrams on pages 108 ff.

## Creating TestCases with the TestCase wizard

As an alternative to manually create TestCases, one can also automatically create TestCases with the TestCase wizard. The TestCase wizard allows to automatically create TestCases based on existing

- Sequence Diagrams
- Operations and Event Receptions

- Requirements

The TestCase wizard automatically add a TestObjective to the newly created TestCase referring to the element for which the TestCase has been created (cf. section Using the Test Requirement Coverage Report on page 103).

A SD TestCase based on an existing Sequence Diagram can be created with the following steps:

1. Right click on a sequence diagram in the browser or in sequence diagram editor and selection of “Create TestCase...” from the context menu.  
This opens the TestCase wizard dialog:
2. In the TestCase wizard dialog, all TestArchitectures (i.e., all TestContexts) that are suitable to map the life lines of the existing sequence diagram to the life lines that are available in the TestArchitecture (i.e., the life lines of the SUT instances and the life lines of the TestComponentInstances) are listed. The list only shows the short names of the TestContexts – if the mouse hovers over the name, a 'bubble' opens and reveals the full path name of the respective TestContext. A TestArchitecture is suitable, if
  - All life lines of the existing sequence diagram can be mapped to life lines of SUT instances or TestComponentInstances s.t. all specified messages can occur also between the remapped life lines of the TestArchitecture.
  - At least one life line of the existing sequence diagram must belong to the same class (or file/object) as one of the SUT instances of the TestArchitecture. This rule can be turned on/off by setting the property `“TestConductor.Settings.MapSDToTestArchitectureMode”` to “weak”. By setting this property to “weak”, no existence of a life line that has the same class as one of the SUT classes is required any more. Only the specified messages must be possible in the remapped life lines of the TestArchitecture. This mode allows to remap an existing sequence diagram also to TestArchitectures that contain completely disjoint classes but which have at least interfaces that are compatible. The default value for this property is “strict”.
3. If no suitable TestArchitecture can be found, the list will contain only the entry <<new>>. On selecting <<new>> a new dialog will open that lists all classes of all life lines of the selected sequence diagram. In this dialog the SUT class has to be selected for which the TestArchitecture will be created. Pressing ok will invoke TestArchitecture creation for the selected SUT.
4. As a result, a new sequence diagram TestCase will be created that contains the same messages as the original sequence diagram, mapped to life lines referring to suitable instances in the TestArchitecture.

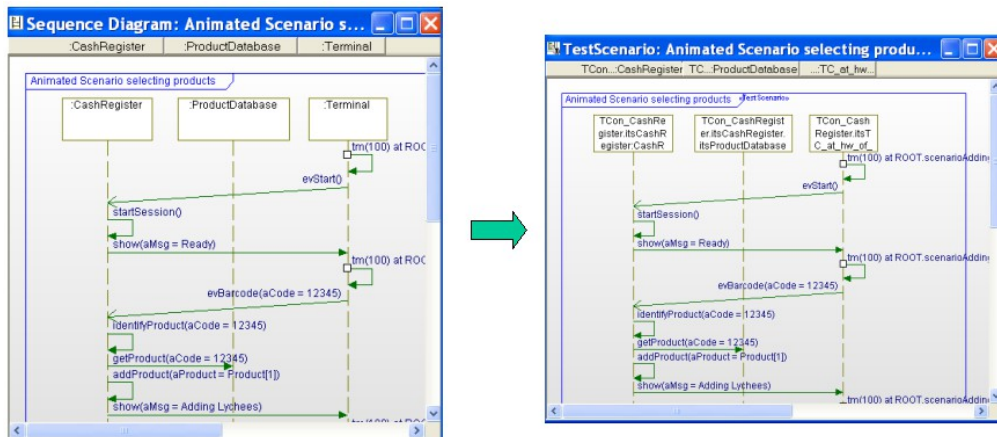


Figure 7: SD TestCase mapping using TestCase Wizard

Note that the long names of the life-lines are navigation expressions referring to suitable instances of the respective classifiers in the target TestArchitecture. TestConductor needs these navigation expressions e.g. for identifying suitable links when generating driver operations.

In order to create a TestCases based on an operation or an event reception, do the following:

1. In the browser, select one of the operations or event receptions of a class (or file/object) and select "Create TestCase..." from the context menu.
2. In the TestCase wizard dialog, all TestArchitectures (i.e., all TestContexts) that contains a SUT instance of the class (or file/object) of the selected operation/event reception are listed. Additionally, the element <<new>> is listed. In the lower left of the dialog a drop down box can be used to select the kind of TestCase to be created. Depending of the selection of the TestArchitecture and the TestCase kind, a new TestCase is created and added to the selected TestArchitecture. If <<new>> is selected, a new TestArchitecture for the class (or file/object) of the selected operation will be created.
3. The created TestCase already contains a call to the selected operation with default arguments. Additionally, a dummy assertion is created that can be refined in order to check out values of the called operation.

In order to create a TestCases based on a requirement, do the following:

1. In the browser, select a requirement and select "Create TestCase..." from the context menu.
2. In the TestCase wizard dialog, all TestArchitectures (i.e., all TestContexts) of the model are listed. Additionally, the element <<new>> is listed. Furthermore, a drop down box can be used to select the kind of TestCase one wants to create. Depending of the selection of the TestArchitecture and the TestCase kind, a new TestCase is created and added to the selected TestArchitecture. When <<new>> is selected, a new TestArchitecture (a subsequent dialogs asks for the class for which a new TestArchitecture should be created) is created. Furthermore, the original requirement for which the new TestCase has been created is linked as a test objective to the TestCase.

## Creating Sequence Diagram TestCases from existing Scenarios using an explicit instance mapping

(see also TestingCookbook:

- “How can I use an existing sequence diagram for tests?”

)

Creating Sequence Diagram TestCases from existing Scenarios can be done either fully automated using the TestCase Wizard (cf. section Creating Sequence Diagram TestCases from existing Scenarios using an explicit instance mapping, page 52 ff) or by explicitly selecting the TestArchitecture in which a TestCase shall be created for the source scenario. Optionally, a mapping of the classifiers of the source scenario to classifiers in the TestArchitecture for which the TestCase will be created can be provided.

When attempting to create a sequence diagram test using the case wizard, the TestCase wizard first analyzes all existing TestArchitectures for being suitable candidates for TestCase creation and offers the suiting TestArchitectures for selection as target for TestCase creation. Hence, if instances or messages in the source scenario have no possible realization according to the automatic mapping algorithm, the respective TestArchitecture is not offered for selection. The algorithm provides no information why certain TestArchitectures aren't considered suitable for the particular scenario.

The heuristics of the mapping algorithm maps classifiers of the source scenario to 'compatible' –according to the chosen mapping strategy (weak or strict, cf. pages General Properties, page 98 ff.) – classifiers in the selected TestArchitecture. The heuristics work pretty well for classifiers with port contracts. In particular for classifiers engaged in only few communications or without port contracts, the heuristics may produce not optimal results.

The TestCase wizard is only capable of an instance to instance mapping. Merging or splitting life-lines – e.g. according to composite and part relations – is not supported by the TestCase wizard.

To overcome the drawbacks described above, creating sequence diagram TestCases from existing scenarios – optionally using an explicit instance mapping – has been introduced as alternative to the TestCase wizard.

A sequence diagram TestCase from an existing scenario can be created by invoking 'Create TestCase from Scenario' on the scenario.

For a user defined mapping and a determined TestArchitecture, the TestCase is created in any case and a detailed report provides feedback about the individual actions the algorithm performed for TestCase creation and scenario mapping. If no mapping is active on invocation of 'Create TestCase from Scenario' the resulting TestCase resembles the result of invoking the TestCase wizard with the major difference that the TestCase is created in the target TestArchitecture even though the TestCase wizard considers the TestArchitecture not suitable. The 'MappingReport' comment in the created TestCase will contain detailed information regarding the successful steps and problems during creation and mapping.

Mappings can define

- simple mappings of individual classifiers to classifiers,

- splitting life-lines of classifiers into a set of life-lines of particular classifiers – as needed e.g. for mapping a composite to its parts,
- merging life-lines of a set of classifiers to one life-line of a particular classifier – as e.g. used in mapping parts to its parent composite.

Once created mappings are part of the model (TestingProfile model element `SDMapping`) and can be shared for further TestCase creations. Definition of mappings is described on page 56.

Mappings refer to the classifiers of life-lines. Mapping of individual messages is currently not supported.

The work flow of sequence diagram TestCase creation for an existing scenario consists of the following steps:

1. Activation of the desired `SDMapping`. An `SDMapping` is activated by setting stereotype `<<ActiveSDMapping>>` on the `SDMapping`. At most one `SDMapping` must be stereotyped at a time.  
A dedicated helper 'Set as Active `SDMapping`' unsets the stereotype from all currently stereotyped `SDMappings` and activates the selected one.
2. The target `TestArchitecture` is determined by setting one of its code generation configurations active. The active code generation configuration must be stereotyped `<<TestingConfiguration>>` or `<<AnimationBasedTestingConfiguration>>` or by a stereotype inheriting from one of them.
3. Invocation of 'Create TestCase from Scenario' on a sequence diagram or a `TestScenario`.

## Definition of mappings for sequence diagram TestCase creation from existing scenarios

Testing profile model elements

- `SDMapping`,
- `SDInstanceRealizationMapPair`,
- `SDInstanceRealizationSplit`,
  - `SDInstanceRealizationSplitTarget`,
- `SDInstanceRealizationMerge`,
  - `SDInstanceRealizationMergeOrigin`

have been introduced for defining mappings for sequence diagram TestCase creation from scenarios.

These model elements have – depending on their meaning to the mapping – tags 'Origin' and 'Target' of type `ModelElement`<sup>9</sup>.

---

<sup>9</sup>Classifier would be more appropriate, but for classifier, the selection dialog doesn't offer files and implicit objects. Thus, to be able to pick also files and objects from the selection dialog for tags, Classifier is too



The top level element of each mapping is an `SDMapping`

`SDMappings` can consist of

- `SDInstanceRealizationMapPair` – simple mappings of individual classifiers to classifiers, `SDInstanceRealizationMapPair` has two tags 'Origin' and 'Target' of type `ModelElement`. life-lines referring to 'Origin' shall be mapped to 'Target'.
- `SDInstanceRealizationSplit` – splitting life-lines of into a set of life-lines of particular classifiers. `SDInstanceRealizationSplit` has tag 'Origin' for defining, which Classifier shall be split and
  - arbitrary many `SDInstanceRealizationSplitTarget` elements, each with a tag 'Target'. The set of `SDInstanceRealizationSplitTarget` elements belonging to a `SDInstanceRealizationSplit` define the set of classifiers to which the life-lines referring to 'Origin' classifier shall be split.
- `SDInstanceRealizationMerge` - merging life-lines of a set of classifiers to one life-line of a particular classifier. `SDInstanceRealizationMerge` has a tag 'Target' denoting the classifier for which the origins will be merged and
  - arbitrary many `SDInstanceRealizationMergeOrigin` elements, each with a tag 'Origin'. The set of `SDInstanceRealizationMergeOrigin` elements belonging to a `SDInstanceRealizationMerge` define the set of elements for which the referring life-lines shall be merged to an life-line referring to 'Target' classifier.

`SDMappings` can be created in any package or `TestPackage` in the model, but it is recommended to create `SDMappings` in the target `TestArchitecture` *to which* the `SDMapping` maps classifiers of scenarios.

`SDMappings` can be created using the context menu item “Add New->TestingProfile->SDMapping” on a package or `TestPackage`. According to the hierarchy of mapping elements, `SDInstanceRealizationMapPair`, `SDInstanceRealizationSplit`, `SDInstanceRealizationMerge` can be added to a `SDMapping` with the context menu item “Add New->TestingProfile-> SDInstanceRealizationMapPair”, etc. on a `SDMapping`.

Similarly, `SDInstanceRealizationSplitTarget` and `SDInstanceRealizationMergeOrigin` can be added accordingly to `SDInstanceRealizationSplit` and `SDInstanceRealizationMerge`, respectively.

The 'Origin' and 'Target' tags of the mapping elements can be set in the tags-tab of the features dialog of the respective element: on clicking into the value entry field of the tag, a '...' button appears on the right side of the entry field. Pressing that '...' button opens a 'Select Value' dialog, which is basically a mini model browser.

---

restrictive. Instead of restricting the selection, the defined `SDMapping` is strictly checked on application of the mapping.

Unfortunately, the tag value is displayed only with its short name in the browser and in the entry field in the features dialog – and the selected model element is not preselected when opening the selection dialog again for a defined tag. This makes it difficult to verify correctness of an existing mapping or even understand its meaning with only the information provided in the browser and in the features dialog. In order to obtain information about the model paths of the selected classifiers in the mapping tags, context menu item 'Update Description' can be invoked on `SDMapping`. This helper will generate an information report for the mapping using model path names of the tag values and write the report to the description of the `SDMapping`.

## SDMappings for Replacements

`TestComponents` are often realized by *replacements*, i.e. copies of the original classes, objects or files in the environment of the SUT – replacing the original model elements in the code generation scope for testing purposes (cf. section Replacements on page 24). It is important to note that even though replacements are used for instrumentation, the `TestScenarios` of sequence diagram `TestCases` refer to the replaced original model elements<sup>10</sup>. Thus, for mappings also the original model elements have to be selected – not the replacements.

---

<sup>10</sup>Except for `TestFiles` replacing files.

# Test Execution

---

During test execution, TestConductor drives events, operation calls, and dataflows sent from the TestComponents or TestContext to SUT objects, and monitors all messages from SUT objects to TestComponents as specified in the TestCases. TestConductor automatically checks and reports whether the order of messages corresponds to the real order in the running application. In addition, TestConductor monitors the arguments of messages.

Before Rhapsody 7.6, TestConductor only supports so-called animation based testing mode. In animation based testing mode, the scheduling and arbitration, i.e., the way TestConductor decides whether a TestCase is passed or failed, is based on animation messages coming from Rhapsody's animation feature. Starting from Rhapsody 7.6, TestConductor also supports so-called assertion based testing mode. In contrast to animation based testing mode, in assertion based testing mode both scheduling and arbitration of TestCases is directly controlled by assertions that are compiled into the test executable, i.e., scheduling and arbitration of TestCases is independent from Rhapsody's animation feature. Since in assertion based testing mode the TestCases are part of the application itself, observation of messages or behavior in the initialization of the application is limited. The TestCase arbitration and scheduling is not initialized before other parts of the application. Hence, for testing system setup using the assertion based testing mode, it is recommended to provide the model with an initial trigger for starting system setup.

**This document only refers to the assertion based testing mode.**

## Overview

TestConductor supports execution of

- individual TestCases
- TestContext
- TestPackage

The test execution is visualized with an execution dialog. Depending on the type of TestCases the view and interaction possibilities of the execution dialog slightly differ.

## Testing Configuration

Prerequisite for each execution of an application is a defined Rhapsody code generation configuration. This configuration must be compileable and linkable.

The code generation configuration used for updating, building and executing must have the stereotype <<TestingConfiguration>>. Each TestArchitecture is

automatically equipped with a suiting `<<TestingConfiguration>>` code generation configuration upon TestArchitecture creation using the context-menu 'Create TestArchitecture' TestConductor.

## Tags of the `<<TestingConfiguration>>` Stereotype

The stereotype `<<TestingConfiguration>>` contains several tags that can be used in order to control how the test executable is created, and which test execution options should be applied when executing TestCases using that configuration:

- **CodeCoverageOptionsFileName**  
In this tag a file name of a code coverage options file can be specified. In the options file, one can specify compiler specific options for controlling the source code annotation tools that annotate the code of the SUT in order to compute code coverage achieved by the executed TestCases.
- **ComputeCodeCoverage:**  
If this option is turned on, when executing TestCases TestConductor computes which parts of the code generated for the SUT are covered to what extend. TestConductor computes statement coverage, decision coverage, decision/condition coverage and modified condition/decision coverage (MC/DC). Which parts of the SUT are considered for code coverage can be controlled by the tag “CoverageKind”. **Note:** Code coverage is restricted to C and C++.  
(Default: false)
- **ComputeModelCoverage:**  
If this option is turned on, when executing TestCases TestConductor computes which model elements of the SUT are covered. TestConductor computes which states, transitions and operations are executed by the TestCases. Which parts of the SUT are considered for code coverage can be controlled by the tag “CoverageKind”.  
**Note:** Model coverage is restricted to animated configurations.  
(Default: false)
- **ComputeRequirementCoverage:**  
If this option is turned on (requires also ComputeModelCoverage to be turned on), TestConductor measures dynamical requirement coverage on the basis of model elements linked to requirements using stereotyped dependencies (cf. page 82 ff. for details on the coverage measure and its configuration options).  
**Note:** Requirement coverage is restricted to animated configurations.  
(Default: false)
- **CoverageKind:**  
This tag controls which parts of the TestArchitecture is considered by model coverage and code coverage. The possible values are
  - **SUT\_flat:** Only the SUT itself is considered.
  - **SUT\_hierarchical:** The SUT and its parts are considered.
  - **TestContext\_flat:** The SUT and all TestComponents are considered.

- **TestContext\_hierarchical:** The SUT and its parts, and all TestComponents with all their parts are considered.

(Default: SUT\_flat)

Despite choosing the coverage kind in general, the coverage scope can be fine-tuned by adding `<<considerForCoverage>>` and

`<<considerNotForCoverage>>` dependencies on the respective model elements (classes, objects and files) to the TestContext.

- **CreateWitnessScenariosForFailedSDTestCase:**

This tag controls whether TestConductor should automatically create a so called witness scenario if an SD based TestCase has failed.

(Default: false)

- **EnableOverloading:**

For Rhapsody in C++, TestConductor supports overloaded operations in TestComponents by default, i.e. operations with identical names but differing formal argument lists - differing in argument types or of different lengths. Normally, there is no reason to disable support for overloaded operations.

(Default: true)

- **NoConsoleApp:**

This tag controls whether TestConductor should hide console windows (cmd windows) when executing test applications and helper tools on Windows.

(Default: false)

- **PopulateCompileCommandForCodeCoverage :**

If this option is turned on, the property “<lang>.<Env>.CCompileCommand” will be automatically populated by TestConductor in order to call the code instrumentation tools of TestConductor that are needed when computing code coverage of TestCases.

If there are problems with the automatic population of this property, turn off this option and adjust the property “<lang>.<Env>.CCompileCommand” manually.

(Default: true)

- **PopulateInvokeExecutableProperty:**

If this option is turned on, when executing TestCases from within Rhapsody, TestConductor automatically overwrites the property

“<lang>.<Env>.InvokeExecutable” with the content of the tag “rtc\_testexecution\_script\_filename”.

(Default: true)

- **PrepareForTestingPrivateOps:**

Rhapsody in C generates file static operations for private model operations, which can not be seen or invoked from outside the generated implementation file.

In order of being able to test also private operations in the SUT, TestConductor will generate additional operations to the SUT code, providing 'public' wrapper operations calling the private operations. These wrapper operations can be used in code , flow chart , and statechart TestCase to test private operations (see section TestConductor Support for Testing Private Operations in Rhapsody in C on page 46 for details).

Rhapsody in C++ uses C++ declaration modifiers public, private and protected according to the visibility settings in the features dialog of operations. For C++, support for testing private and protected operations similar to the support for testing private operations in C is available using this tag (see section TestConductor Support for Testing Private and Protected Operations in Rhapsody in C++ on page 47 for details).

(Default: false)

- **RTC\_MAX\_ASSERT:**

The value of this tag defines how much memory TestConductor reserves for storing the results of executed assertions. The memory for storing the results of assertions is always defined statically in order to allow test execution on targets that don't support dynamic memory allocation. If during test execution the assertion memory exceeds its limits, TestConductor stops test execution and logs an error message.

(Default: 200)

- **ResultVerification:**

TestCases can be defined by either sequence diagrams, flowcharts, statecharts or plain code. Based on the behavior specification of the TestCase, TestConductor populates the model with operations and statecharts that implement the behavior of the TestCase as specified e.g. by a sequence diagram. After model population, TestConductor uses Rhapsody's code generator in order to generate code from the populated model. Now, if Rhapsody's code generator contains an error, a TestCase execution could yield the wrong result since TestConductor has used Rhapsody's code generator to generate the testing code. In order to prevent such situations, TestConductor can perform a so-called result verification. Result Verification is a technique that checks the consistency of a test execution with the TestCase behavior specification in Rhapsody. If result verification is turned on, TestConductor will detect potential errors in Rhapsody's code generator, thus making sure that the TestCase result TestConductor computes is correct even if code generation errors occurred in the testing code.

(Default: true)

- **ShowActualValueInWitnessScenario:**

This tag controls if TestConductor should automatically show the observed parameter or return value for failed messages when creating a witness scenario. Showing the actual value is supported for basic types (like int, long, double) but not for typedef of enum types. This option is not supported in combination with option rtc\_assert\_handling set to by\_id.

(Default: true)

- **rtc\_adapter\_content:**

This tag allows for defining adapter code, that can be used to realize the transfer of results from the target to the host. For example, a target debugger script can be provided in this tag, that reads out the assertion array and dumps the content of the array to a file on the host.

(Default: empty)

- **rtc\_adapter\_filename:**

If tag rtc\_adapter\_content is not empty, then rtc\_adapter\_content is written to the denoted file for use in e.g. a target debugger.

(Default: \$CONFIGDIR/rtcadapt.txt)

- **rtc\_assert\_dumpfile:**  
If turned on, then the contents of the assertion array will be dumped to the file denoted by tag `rtc_assert_resultfilename`.  
The tag must be turned off if the target does not support files.

(Default: true)

- **rtc\_assert\_dumpfile\_kind:**  
This tag controls when the collected assertions are dumped into the result file. Possible values are
  - **at\_exit:** assertions are dumped when the test executable exits.
  - **after\_testcase:** assertions are dumped after one TestCase execution.
  - **immediately:** assertions are dumped immediately when they are executed.

(Default: immediately)

- **rtc\_assert\_handling:**  
This tag controls how much information is stored with the outcome of each assertion. Possible values are:
  - **by\_string:** With this value it is possible to add a string with additional information to each assertion, providing more information when doing show assertion or show as SD. This allows easier analyzing of test outcomes but increases the memory consumption of the tested application.
  - **by\_id:** With this value only a unique number and the result of each assertion. This reduces the memory consumption of the tested application.

(Default: by\_string)

- **rtc\_assert\_mem\_code:**  
This tag allows for customization of the `rtc_assert_id` function. Function `'void rtc_assert_id(int e, int ln, int nr)'` is defined in `TestConductor_C.c` (for C) and `TestConductor.h` (for C++), respectively. If **rtc\_assert\_mem\_code** is empty, the original implementation as provided by TestConductor is used.  
The function takes 3 arguments:
  - `int e`: the value of the assertion expression
  - `int ln`: the line number of the assertion in the source code
  - `int nr`: the number of the implementation file according to a TestConductor-internal numbering of generated files.

TestConductor expects a result file on the host with the following syntax:

$$\begin{array}{lcl} \mathbf{Lines} ::= & \varepsilon & \\ & | \mathbf{Lines Line} & \\ \mathbf{Line} ::= & \text{ASSERTION} = \text{nr,ln,e} & \end{array}$$

Where  $\epsilon$  means the empty word, 'ASSERTION', '=', and ',' are token and nr, ln, e are integer values according to the arguments of *rtc\_assert\_id*. (in reversed order).

For simplicity, arbitrary text lines not starting with 'ASSERTION' may be contained in the result file but are ignored.

Using **rtc\_assert\_mem\_code**, the implementation of *rtc\_assert\_id* can be customized in any way that produces a result file in correct syntax on the host, e.g. sending the values via serial connection to a serial port server application on the host that creates the result file.

(Default: empty)

- **rtc\_exit\_kind:**

This tag controls how the test executable shall be exited. Possible values are:

- **by\_system\_exit:** The test executable exits by calling "exit".
- **User\_defined:** The test executable exits by executing the content of the tag "rtc\_exit\_user\_definition".

(Default: by\_system\_exit)

- **rtc\_exit\_user\_definition:**

In this tag you can specify a code sequence that shall be executed when the test executable exits. This can be useful e.g. for targets that need a special way for correctly terminating executables.

(Default: empty)

- **rtc\_info\_filename:**

This tag specifies the name of the so-called info file that is used by TestConductor in order to generate some TestCase related information into a file, e.g. name and id of TestCases. The info file is used by the reporting tool repgen in order to generate execution reports.

(Default: \$CONFIGDIR/rtcinfo.txt)

- **rtc\_log\_autogenerate**

If this tag is turned on, TestConductor automatically adds log messages to the test executable. The log messages give information e.g. which TestCase is currently executed. Based on the value of the tag "rtc\_log\_kind", the generated log messages are either printed to the console or to a log file or both.

(Default: true)

- **rtc\_log\_filename**

This tag specifies the name of the log file that can be generated by the test executable. If the file is generated or not during test execution depends on the value of the tag "rtc\_log\_kind".

(Default: \$CONFIGDIR/rtclog.txt)

- **rtc\_log\_kind**

This tag specifies how log messages should be treated inside the test executable. The possible values are

- **to\_console:** log messages are printed to the console



- `to_file`: log messages are printed to the file specified in the tag `"rtc_log_filename"`.
- `to_console_and_file`: log messages are printed to the console and are logged into the file specified in the tag `"rtc_log_filename"`
- `user_defined`: when log messages are executed, the code entered in the tag `"rtc_log_user_definition"` is executed.

(Default: `to_console`)

- **`rtc_log_user_definition`:**

In this tag you can specify a code sequence that is executed in the test executable when a log message is specified. The specified code sequence will be executed if the value of the tag `"rtc_log_kind"` is set to `"user_defined"`.

(Default: empty)

- **`rtc_report_dir`**

This tag specifies to which directory TestConductor generates the execution reports after TestCase execution.

(Default: `$CONFIGDIR`)

- **`rtc_result_filename`:**

This tag denotes the file from which TestConductor will read the result of TestCase execution. If tag **`rtc_assert_dumptofile`** is set to true, then the results will automatically be written into this file.

(Default: `$CONFIGDIR/rtcresult.txt`)

- **`rtc_result_handling`:**

This tag specifies how test execution results are treated in the test executable. Possible values are

- `automatic`: if set to automatic, TestConductor automatically reads in test results after test execution.
- `Manual`: if set to manual, TestConductor does not automatically reads in test results after TestCase execution.

- **`rtc_testexecution_script_content`:**

This tag specifies the content of the script file that is used by TestConductor to call the test executable. The tag contains the options for the test executable that e.g. are used to select the TestCase that shall be executed.

(Default: `"$executable" -resultfile "$rtc_resultfile" -logfile "$rtc_logfile" -tcontext $tcontext -tcase $tcase`)

**Note:** When testing with Cygwin environment a trailing `"$OMROOT/etc/cygwinrun.bat"` (with `$OMROOT` expanded) is added to the script content to make sure the tested application can load the necessary Cygwin dlls and start correctly.

This can be disabled by adding the tag `"DisableUseCygwinrun"` with the value `"True"` to the code generation configuration.

- rtc\_testexecution\_script\_filename:**  
 This tag specifies the name of the script file that is used in order to call the test executable.

(Default: \$CONFIGDIR/tc\_run.bat)
- rtc\_testexecution\_script\_populate:**  
 This tag specifies whether the content of the file specified in the tag "rtc\_testexecution\_script\_filename" is populated with the content specified in the tag "rtc\_testexecution\_script\_content".

(Default: true)
- rtc\_testexecution\_update\_check:**  
 By default, TestConductor checks on every invocation of 'Build TestCase/TestContext/TestPackage' and on every invocation of 'Execute TestCase/TestContext/TestPackage' whether an update of the TestCase, TestContext or TestPackage, respectively is needed, since test definitions have been modified and test artifacts have to be updated accordingly. The update check may take reasonable time on large TestContexts and TestPackages if e.g. many TestCases are involved. Tag `rtc_testexecution_update_check` turns off the update check and leaves it to the user to ensure that all TestCases have been updated appropriately before invoking build and execute.  
 Main use case is to turn off update check for applying manual modifications after update before build and execute, since update check inhibits build and execute for not appropriately updated TestCases.  
**Note:** be careful disabling update check.

(Default: false)
- rtc\_testreport\_script\_content\_tcase:**  
 This tag specifies the content of the script file that is used by TestConductor to generate html execution reports for TestCases from the test results computed by the test executable. The tag contains the options for the repgen tool that are used in order to generate the html reports for TestCases

(Default: "\$RTCINSTALLDIR/repgen" -infofile "\$infofile" -resultfile "\$resultfile" -outdirectory "\$RTCREFDIR" -tcontext \$fulltcontext -tcase \$fulltcase)
- rtc\_testreport\_script\_content\_tcontext:**  
 This tag specifies the content of the script file that is used by TestConductor to generate html execution reports for TestContexts from the test results computed by the test executable. The tag contains the options for the repgen tool that are used in order to generate the html reports for TestContexts.

(Default: "\$RTCINSTALLDIR/repgen" -infofile "\$infofile" -resultfile "\$resultfile" -outdirectory "\$RTCREFDIR" -tcontext \$fulltcontext)
- rtc\_testreport\_script\_filename:**  
 This tag specifies the name of the script file that is used by TestConductor in order to generate html reports based on the execution results computed by the test executable.

(Default: \$CONFIGDIR/tc\_rep.bat)

- **rtc\_testreport\_script\_populate:**  
If this tag is turned on, the content of the file specified in the tag “rtc\_testreport\_script\_filename” will be populated with the content of the tag “rtc\_testreport\_script\_content\_tcase”, if a TestCase is executed, and with the content of the tag “rtc\_testreport\_script\_content\_tcontext”, if a TestContext is executed.  
  
(Default: true)

## TestConfiguration Dependency

Most TestConductor functions depend on properties and definitions in code generation configurations. E.g. instrumentation during 'Updates TestCase/TestContext/TestPackage' as well as TestArchitecture Update compute their results according to certain properties to support test execution against different code generation configurations. After creation there is a <<TestConfiguration>> dependency targeting a Rhapsody <<TestingConfiguration>> code generation configuration, located underneath the TestContext.

The algorithm TestConductor uses to choose the appropriate configuration is as following:

- If the currently active configuration is located in the same component as the configuration targeted by the <<TestConfiguration>> dependency of the TestContext is a <<TestingConfiguration>>, the currently active configuration will be used.
- Otherwise the configuration targeted by the <<TestConfiguration>> dependency (Default TestingConfiguration) of the TestContext will be used.

One can switch between the code generation configurations by switching the active Rhapsody configuration from those configurations in the same component as the default TestingConfiguration.

If no <<TestConfiguration>> dependency exists, an error message is issued if the active code generation configuration is not stereotyped <<TestingConfiguration>>. If the active code generation configuration is stereotyped <<TestingConfiguration>>, TestConductor will try to perform the desired action with the active code generation configuration without further checking if the active configuration is suitable for the TestArchitecture.

## Execution Results

After the test executable has been built, individual TestCases, entire TestContexts or TestPackages can be executed. When invoking a TestCase from within Rhapsody, TestConductor calls the script specified in the <<TestingConfiguration>> tag “rtc\_testexecution\_script\_filename” that actually calls the test executable with the parameters that select the TestCase that shall be executed. The chosen TestCase is executed, and after termination the results are dumped into the result file specified in the <<TestingConfiguration>> tag “rtc\_result\_filename”. However, this result file only contains the raw results, i.e., the outcome of the assertions that have been executed

during test execution. In order to generate a complete test execution report based on these raw results, TestConductor uses the tool “repgen”. After test execution, when the raw results have been computed by the test executable, TestConductor calls the script that is specified in the tag “rtc\_testreport\_script\_filename”. This script actually calls repgen with the correct parameters in order to generate both a xml report and a html report that shows the detailed test results. The generated xml report is only used internally by TestConductor in order to present the execution results in the test execution GUI when working within Rhapsody. In summary, in assertion based testing, test execution and test reporting is a process separated into 2 steps:

- TestCases are executed by calling the test executable with the correct parameters. The test executable computes raw test results.
- Based on the raw test results, a call of the repgen tool with the correct parameters generates readable html reports based on these raw results.

Both of these steps can either be done from within Rhapsody (the same way as for animation based testing) or outside of Rhapsody.

## Performing result verification for TestCase execution

When operating in assertion based testing mode, TestConductor provides the option to perform a so-called result verification after TestCase execution. This feature is turned on if the tag “ResultVerification” of the TestingConfiguration is turned on. When result verification is turned on, after TestCase execution TestConductor checks if the raw results written to the result file by the test executable is consistent with the graphical behavior description in Rhapsody (either as sequence diagram, statechart, or flowchart). For a behavior description provided as plain code no result verification is performed. For graphical behavior description provided as a sequence diagram, TestConductor populates the model with a statechart that represents the possible allowed execution sequences specified in the sequence diagram. The result verification check made by TestConductor is independent from Rhapsody’s code generator, and can be used in order to detect defects of Rhapsody’s code generator that may influence the TestCase execution results. By using result verification, TestConductor makes sure that the test execution results computed by TestConductor are ALWAYS correct, even in case of errors in Rhapsody’s code generator that may affect the correctness of the testing source code that is used to build the test executable. The result verification is able to detect e.g. the following potential code generation problems that may influence the test execution result:

- The code generator wrongly ignores transitions or states in a statechart
- The code generator wrongly takes additional transitions in a statechart
- The code generator fires statechart transitions in wrong order
- The code generator wrongly ignores transitions or actions in a flowchart
- The code generator wrongly takes additional transitions in a flowcharts
- The code generator fires flowchart transitions in wrong order

When result verification is turned on (by default), the generated html test execution result always contains the information if result verification was enabled or not, and if it was successful or not. In case result verification was enabled and it was not successful, the TestCase status is automatically set to “Error”.

## TestCase Result

TestCase: SD\_with\_alt

Friday, May 13, 2011 09:00:51

Environment Information	
Test executed on machine:	TSV
Test executed by user:	User
Used operating system version:	Windows 2000 / Windows XP
Used Rhapsody version:	7.6, build 2063218
Used TestConductor version:	2.4.4, build 2481

Tested Project	
Project:	CModelCodeCoverage
Active Code Generation Component:	TPkg_Calc_Comp
Active Code Generation Configuration:	CodeCoverageConfig

SequenceDiagram used in TestCase	
TPkg_Calc::TCon_Calc_Architecture::TCon_Calc.SD_with_alt::SDTestScenario_0	

Results	Summary: PASSED
SDInstance 'SD_tc_0'	
Status:	PASSED
Progress:	100% (7/7)

Result Verification	
Result verification successful	

Figure 8: Test Result Report with Result Verification Information

## TestCase Execution

In order to execute a TestCase, the TestCase has to be updated (for details of the update cf. section Model Population – Create Driver Operations and StubOperations on page 48) and build:

- The context menu on TestCase, TestContext and TestPackage offers 'Update TestCase', 'Update TestContext' and 'Update TestPackage', respectively.

Definition of this functionality is hierarchical: 'Update TestCase' instruments the TestContext according to the needs of the individual TestCase. 'Update TestContext' is based on updating all TestCases belonging to the TestContext, while 'Update TestPackage' is based on updating all TestContexts and TestPackages contained in the TestPackage to be updated.

TestConductor may issue warnings and errors during update. Warnings and errors will appear in the 'Log'-tab off the output window.

- The context menu offers 'Build TestCase', 'Build TestContext' and 'Build TestPackage' on TestCase, TestContext and TestPackage, respectively.

By default, building TestCase/TestContext/TestPackage performs an up-to-dateness check before building and prompts for ok if a an update is considered necessary before building. The up-to-dateness check can be turned off by unchecking tag `rtc_testexecution_uptodate_check` of the concerned <<TestingConfiguration>> code generation configuration.

Note that invoking the build from the context-menu is not the same as invoking regenerate and build for the active code generation configuration in the Rhapsody user interface or 'Code' menu: TestConductor requires some additional information for result verification and a configuration header

TestConductorControl.h configuring basic TestConductor functionality. This required information will only be generated and written when using the context-menu build commands.

- The context menu offers 'Execute TestCase', 'Execute TestContext' and 'Execute TestPackage' on TestCase, TestContext and TestPackage, respectively.  
Again, this functionality is hierarchically based on executing a single TestCase very similar to update and build).

## Test Execution Dialog for code, flow chart, startechart based tests

Flow chart, code, and statechart TestCases are merely code based TestCases, because TestConductor uses the code generation capabilities of Rhapsody's code generator. The execution dialog enables you to activate the actual test execution and displays the test results.

If you have modified your SUT or your TestContext, you must rebuild the code of the TestContext before you start actual test execution.

Execute any TestCase by using the context menu entry 'Execute TestCase'. The TestConductor execution dialog will open, and the TestCase execution will be started.

### Test Execution Dialog

TestConductor displays the assertions defined in a code, flow chart, or statechart TestCase at run-time of the TestCase. During test execution new assertions are listed as soon as they are reached and checked by TestConductor. Each line in the dialog displays information about one particular assertion including the final results, as shown in the following figure.

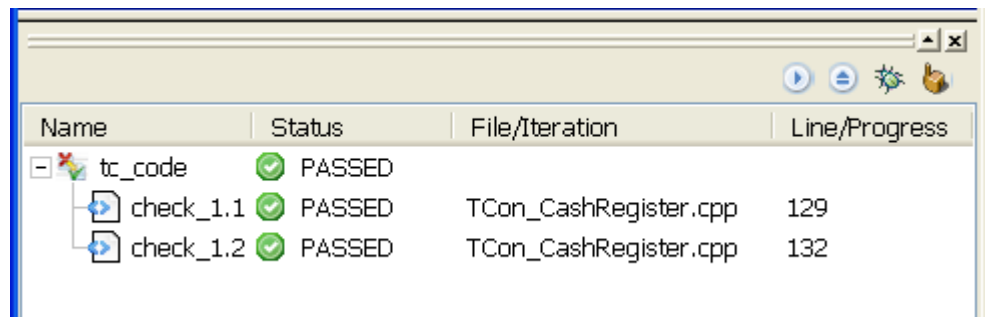


Figure 9: Test Execution Dialog for Code-, Flowchart- and Statechart TestCases

After the TestCase execution has been terminated you can analyze the results of executed assertions.

## Test Information

TestConductor displays information to analyze the test results. The information columns are as follows:

- **Name**—Displays the name of the assertion checked by TestConductor during test execution.
- **File/Iteration**—Shows information about the source file name in which the TestConductor assertion is specified. If a SD TestCase is executed, it shows the iteration number of the SDInstance.
- **Line/Progress**—Shows information about the code line within the file in which the assertion is specified. If a SD TestCase is executed, it shows the progress of the SD instance.
- **Result**—Shows the result of the assertion. The possible values are *PASSED* and *FAILED*.

Double clicking an individual assertion or invoking 'Show Assertion' on an assertion in the execution dialog will highlight the assertion in their model context.

## Controlling TestCase execution

The TestCase execution dialog provides several functions that can be used to control the TestCase execution. The functions are available by pressing one of the icons in the top right corner of the execution dialog.

## Test Execution Dialog for sequence diagram based tests

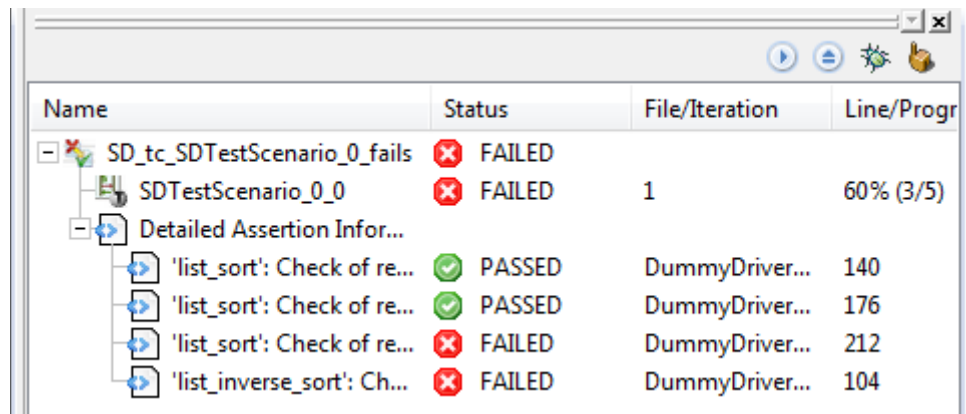
The execution dialog enables you to activate the actual test execution and displays the test results. You can use test results in order to generate sequence diagrams for further regression testing or in order to prepare documentation.

If you have modified your SUT or your TestContext, you must rebuild the code of the TestContext before you start test execution.

Context menu entry 'Execute TestCase' of a selected TestCase opens the execution dialog. For a sequence diagram that is exclusively referenced by only one TestCase, the execution dialog can alternatively be opened using the context menu entry 'Execute TestCase of TestScenario' of the selected sequence diagram. After selecting 'Execute TestCase', the execution dialog opens and the TestCase execution starts.

## Test Execution Dialog

During TestCase execution, the test execution information is displayed in the test execution dialog.



Name	Status	File/Iteration	Line/Progress
SD_tc_SDTestScenario_0_fails	FAILED		
SDTestScenario_0_0	FAILED	1	60% (3/5)
Detailed Assertion Infor...			
'list_sort': Check of re...	PASSED	DummyDriver...	140
'list_sort': Check of re...	PASSED	DummyDriver...	176
'list_sort': Check of re...	FAILED	DummyDriver...	212
'list_inverse_sort': Ch...	FAILED	DummyDriver...	104

Figure 10: Test Execution Dialog for SD TestCases

## Test Information

In the execution window, TestConductor displays information to analyze the test results. The information columns are as follows:

- **Name**—Shows the list of all run-time instances in the order of their appearance in the test. You can activate sequence diagram instances sequentially (one after another) or in parallel (independently).
- **Status**—Shows the current states of run-time instances during test execution. The possible values are “*NOT ACTIVE*”, “*ACTIVE*”, “*PASSED*”, and “*FAILED*”. In the example, the entire test executes automatically, until it eventually shows the final result “(Status – *FAILED*)”, because TestConductor found an error.
- **File/Iteration**—In assertion based testing mode, TestCases can't be iterated. Hence Iteration is always 1 in assertion based testing mode. For individual assertions as in the Detailed Assertion Information of SD TestCases or the user defined assertions of Code TestCases, column File/Iteration shows the file in which the respective assertion is located.
- **Line/Progress**—Shows the percentage of message actions that passed successfully through the tested sequence diagram instance during test execution. A message action is one of the following:
  - Event sending
  - Internal event consumption
  - Operation call
  - Condition mark validation

For example, every event arrow in a sequence diagram potentially specifies two ordered message actions. It depends on the source and origin of a message (SUT or TestComponent) whether sending or reception of the message can be instrumented or not (cf. section Influencing DriverOperation and StubOperation Generation on page 119 ff). TestConductor only counts the instrumented message ends for the progress counter. TestConductor displays the progress as “percentage X/Y”. The X stands for the number of instrumented actions that passed; Y stands for all the instrumented actions belonging to the sequence diagram.



## Displaying Test Results by witness scenarios

After execution of a TestScenario based TestCase, 'Show As SD' can be invoked from the execution window to generate and show a witness TestScenario of the execution.

On invoking “Show as SD”, TestConductor automatically adds a color coded TestScenario in the model to the failed TestCase.

The resulting witness can be used for failure analysis.

See section Failure Analysis on page 132 ff for more information about failure analysis.

## Automatically adding witness scenarios to the model for failed SDInstances

Sometimes it is useful that SDs showing failed SDInstances are added automatically to the model after TestCase execution, e.g. for documentation purposes or if TestCases are executed in batch mode and failed TestCases are analyzed later.

In order to do this in *animation based testing mode*, property

“TestConductor.TestCase.CreateSDForFailedSDInstance” can be switched on. *Assertion* based testing mode ignores this property. Instead, tag

CreateWitnessScenarioForFailedSDTestCase of the

<<TestingConfiguration>> can be used to create witness scenarios for failed SD TestCases executed for this configuration.

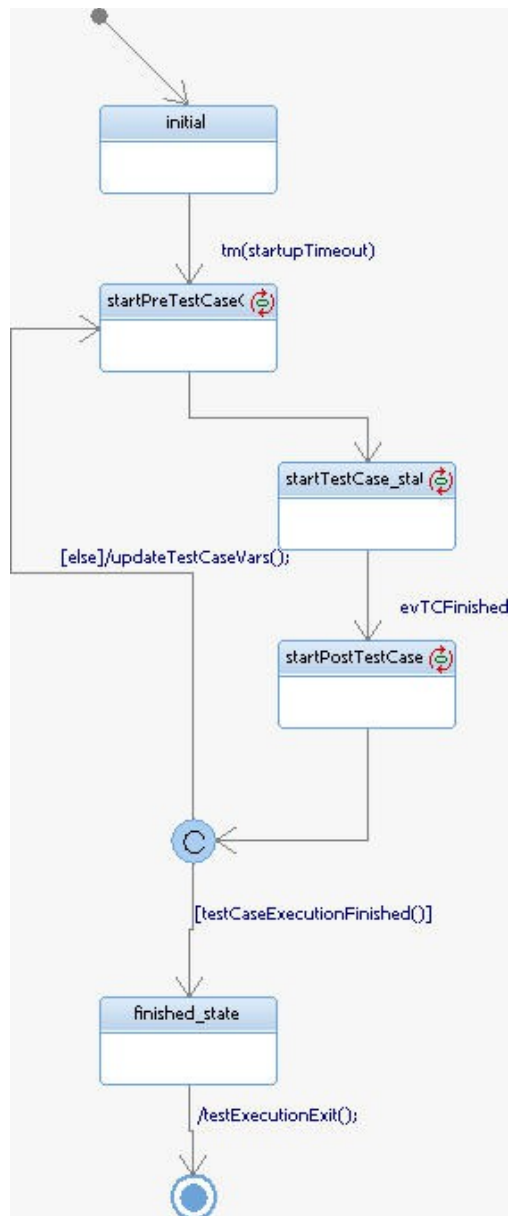
Now, after executing a TestCase with the tag switched on in the active code generation configuration, TestConductor automatically adds a scenario to the TestCase showing the reason of the TestCase failure.

## Abort Test Execution

In order *to abort a running test* either click the **stop icon** in the Rhapsody tool bar or click the **abort icon** in the test execution window.

## Execution Timeout

In assertion based testing mode, in order to define a timeout for TestCases, the scheduler that actually starts and stops the TestCase execution can be modified. By default, a standard scheduler that is auto generated for a TestArchitecture has the following structure:



*Figure 11: Unmodified TestCase Scheduler Statechart*

In order to define a TestCase timeout that works for all executed TestCases, add the following transition to the scheduler with the timeout value you want to have for your TestCases. In the depicted sample, we choose a timeout value of 3 seconds:

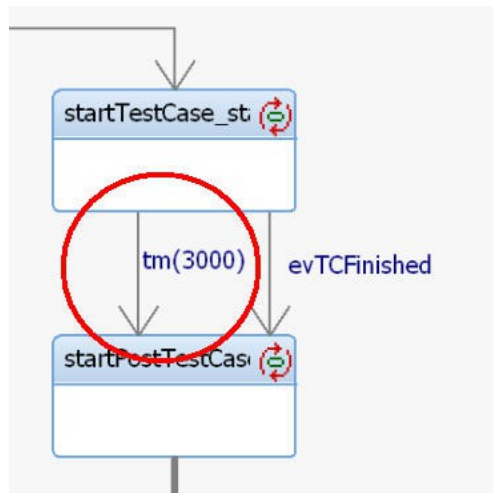


Figure 12: Introducing TestCase Timeout Transition

Alternatively, also property

`TestConductor.TestCase.ExecutionIdleTimeout` (Default 600 (seconds)) can be used to define a timeout for an individual TestCase. Note, that this is the only `TestConductor.TestCase` - property on TestCases that is regarded by assertion based testing mode.

## Test Execution Report

After the execution of a TestCase has finished and the execution dialog has closed, an execution report is written into a *HTML file*. This file is added to the TestCase as a controlled file.<sup>11</sup> If a report file already exists it is overwritten, only the report of the last execution is stored in the model. If a TestCase is executed for multiple code generation configurations, for each configuration a separate test execution report is stored in the model. This way the test results with different settings (debug, release) or from different execution environments (host, target) can be compared.

TestConductor also stores a tag *Verdict* below the linked report file, which stores the result of the TestCase execution.

Possible values are: "*Passed*", "*Failed*", "*Aborted*", "*Timeout*" and "*Undefined*" and "*Error*".

A double click on the test result in the browser opens the linked HTML test report<sup>12</sup>.

<sup>11</sup>Note that with the property `TestConductor.Settings.ReportLocation` (see page 99) a user can specify a dedicated report location)

<sup>12</sup>Open policy for html documents can be influenced by properties `Model.ControlledFile.OpenPolicy` and `Model.ControlledFile.OpenCommand`

## Debugging TestCases

When a TestCase fails one can use TestConductor's debugging capabilities to analyze the reason for the fail. "Debugging mode" in the TestCase execution window can be turned on by pressing the button displaying a bug:

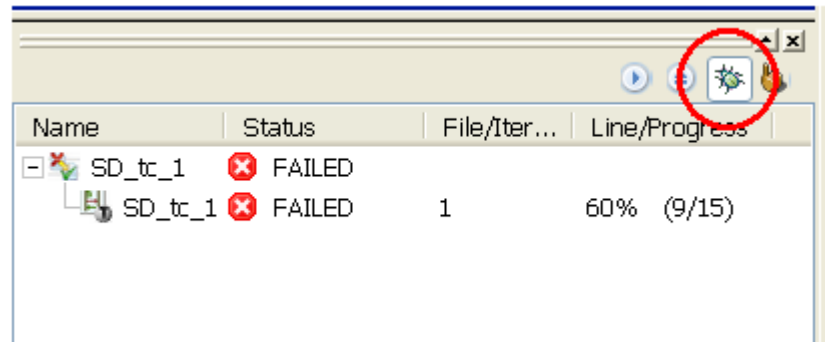


Figure 13: Debugging Button in Test Execution Window

After turning on debugging mode, one can restart the TestCase, e.g. by pressing the "Start" icon in the execution window. In contrast to normal test execution mode, in debugging mode the test execution does not progress automatically but can be controlled by using Rhapsody's animation toolbar. TestCase execution can then be controlled "Go Step", "Go", "Go Idle", "Go Event" and "Go Action" commands in the animation toolbar. In the execution window, one can see the current progress of the TestCase, and in parallel Rhapsody's animation features can be used (e.g. animated sequence diagrams or animated statecharts) to inspect the model during execution of the TestCase. Besides the Go-commands, also all other animation commands are available, e.g. one can add tracer commands to the command prompt as for example "trace #all all" to trace almost everything during execution.

Debugging TestCases is available only if the test application was built with animation instrumentation.

Debugging a TestCase is possible only when executing a single TestCase. When executing a TestContext or TestPackage the Debug button is disabled (and switched off).

The TestConductor Tutorials provide exercises on debugging TestCases.

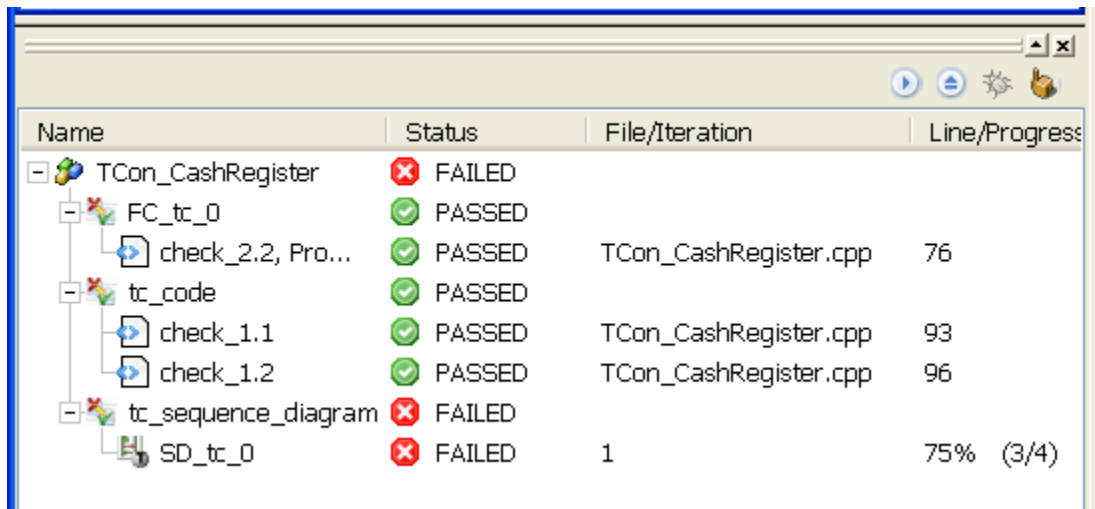
## TestContext Execution

### Starting Test Execution

One kind of batch execution is the execution of a complete TestContext. It will then execute all TestCases belonging to a TestContext.

- Right-click on the TestContext and select 'Update TestContext' from the context menu. This updates all necessary driver and StubOperations derived from the defined sequence diagram TestCases within the TestContext.
- Right-click on the TestContext and select 'Update TestContext' from the context menu. This re-generates the necessary code for all elements of the TestArchitecture and starts the compile and link process for the TestArchitecture.
- Right-click on the TestContext and select 'Execute TestContext' from the context menu. This starts the batch execution for all defined TestCases within the TestContext.

- If the user selects a TestContext and invokes its execution, all TestCases of this TestContext are executed in a sequence.



Name	Status	File/Iteration	Line/Progress
TCon_CashRegister	FAILED		
FC_tc_0	PASSED		
check_2.2, Pro...	PASSED	TCon_CashRegister.cpp	76
tc_code	PASSED		
check_1.1	PASSED	TCon_CashRegister.cpp	93
check_1.2	PASSED	TCon_CashRegister.cpp	96
tc_sequence_diagram	FAILED		
SD_tc_0	FAILED	1	75% (3/4)

Figure 14: Test Execution Window for TestContext Execution

## Stopping Test Execution

To terminate the execution of a TestContext or a TestPackage, press the **abort icon** in the test execution window.

## Execution Timeout

The testing profile defines a global timeout, which can be overwritten for every TestPackage, TestContext and TestCase. This default value is 600 seconds.

You may define a timeout for this batch mode execution of TestCases individually per TestCase. This can be done via the property

```
TestConductor::TestCase::ExecutionIdleTimeout
```

If a timeout is defined and the application doesn't show any activity for <value of timeout> seconds the execution of this TestCase is interrupted and the next TestCase is started. In this case, this TestCase will be marked as “*timeout*” in the result report.

## Ordering of TestCases

The *order of the TestCases* inside the TestContext (similar to the “*Edit Operations Order*” in the Rhapsody browser) can be changed. In this way you can influence the execution order of the TestCases.

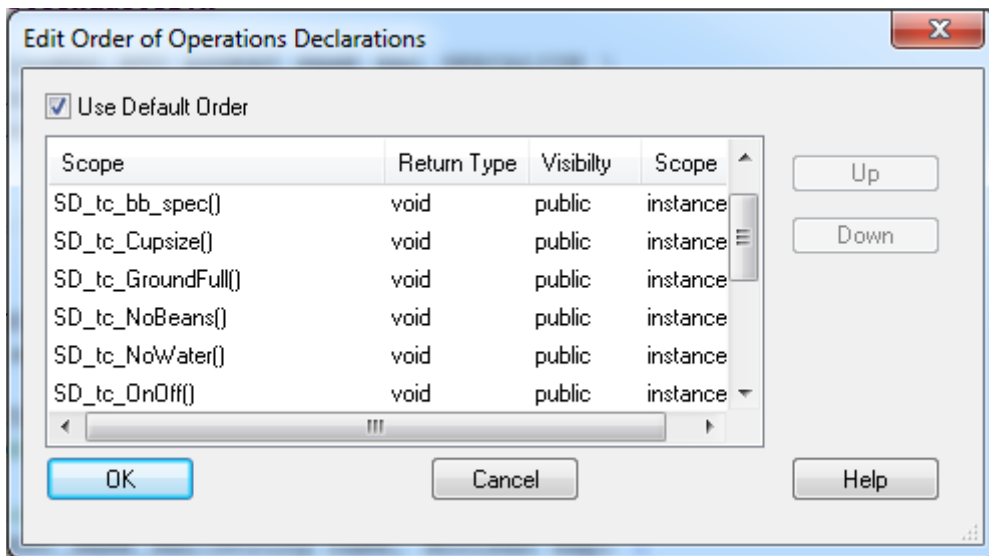


Figure 15: Ordering of TestCases

Per default the TestCases are sorted and executed in alphabetical order.

## Test Execution Report for TestContext

After execution of each TestCase its result *HTML report* is written. The file is added to the TestCase as controlled file.<sup>13</sup>

After execution of all TestCases an execution report of the whole TestContext is written into an HTML file. The file is added to the TestContext as controlled file.

If a report file already exists it is overwritten, only the report of the last execution is stored in the model. If a TestCase or TestContext is executed for multiple code generation configurations, for each configuration a separate test execution report is stored in the model. This way the test results with different settings (debug, release) or from different execution environments (host, target) can be compared.

## TestPackage Execution

### Starting Test Execution

One kind of batch execution is the execution of a complete TestPackage. It will then execute all TestCases underneath all TestContexts belonging to a TestPackage.

- Right-click on the TestPackage and select 'Update TestPackage' from the context menu. This updates all necessary driver and StubOperations derived from the defined sequence diagram TestCases within the TestPackage.

<sup>13</sup>Note that with the property `TestConductor.Settings.ReportLocation` (see General Properties on page 99) a user can specify a dedicated report location)

- Right-click on the TestPackage and select 'Build TestPackage' from the context menu. This re-generates the necessary code for all elements of the TestArchitectures and starts the compile and link process of all TestArchitectures.
- Right-click on the TestPackage and select 'Execute TestPackage' from the context menu. This starts the execution of all defined TestCases within the TestPackage.

Executing a TestPackage is almost like the execution of a TestContext, except the following difference:

- If one TestContext cannot be executed, this TestContext is skipped, the reason for the problem is written to the result report, and the next TestContext is executed.

## Stopping Execution

The **abort icon** in the test execution window aborts the execution of a TestContext or a TestPackage..

## Execution Timeout

The testing profile defines a global timeout, which can be overwritten for every TestPackage, TestContext and TestCase. This default value is 600 seconds.

A timeout for execution of TestCases can be defined individually per TestCase via the property

```
TestConductor::TestCase::ExecutionIdleTimeout
```

If a timeout is defined and the application doesn't show any activity for <value of timeout> seconds the execution of this TestCase will be interrupted and the next TestCase will be started. In this case, this TestCase will be marked as “inconclusive” in the result report.

## Test Execution Report for TestPackage

After the execution of all TestCases, the execution report is written into an HTML file. This file is added to the TestPackage as a controlled file.<sup>14</sup> A report for each TestContext and TestCase that has been executed was also created during execution.

If a report file already exists it is overwritten, only the report of the last execution is stored in the model. If a TestCase or TestContext or TestPackage is executed for multiple code generation configurations, for each configuration a separate test execution report is stored in the model. This way the test results with different settings (debug, release) or from different execution environments (host, target) can be compared.

## Computing Model Coverage during Test Execution

(see also Sample-models:

---

<sup>14</sup>Note that with the property `TestConductor.Settings.ReportLocation` (see General Properties on page 99) a user can specify a dedicated report location)

- CSamples/TestConductor/CModelCodeCoverage
- CppSamples/TestConductor/CppModelCodeCoverage

)

When executing TestCases, i.e., either individual TestCases, a TestContext or a TestPackage, TestConductor provides the possibility to compute which model parts of the SUT are executed during the execution of the TestCases. This information is provided by an HTML report that is created and added to the model after the execution of the TestCases. The report contains information about accumulated coverage of states, transitions, events and operations (except constructors and destructors) of all SUT classes used in the TestArchitecture.

## Computing Model Coverage for single TestCases

The <<TestingConfiguration>> stereotype provides tag “ComputeModelCoverage” for switching on model coverage measurement.

If the tag is switched on for a <<TestingConfiguration>> code generation configuration, then each TestCase, TestContext and TestPackage execution for this configuration will generate a model coverage report in addition to the execution report. The model coverage report will be added to the model in the same location as the execution report<sup>15</sup>. If a TestCase is executed for multiple code generation configurations, for each configuration with enabled model coverage measurement a separate model coverage report will be stored in the model.

## Coverage Items

Model elements which are subject to the coverage are the *operations*, *event receptions* and elements in *behavior specifications* (statecharts or activities) of the classes for which coverage is measured (for the selection of classes for the coverage measurement see section 'Choosing the Coverage Kind for Model Coverage'). If an operation is specified by a behavior diagram<sup>16</sup>, this behavior is considered as well. Of a behavior all vertexes and transitions contained in the behavior are considered. If a coverage item is marked as 'covered' this means that the corresponding code generated for the model element has been traversed during the execution of the test, e.g. an operation has been called or a state in a statechart has been reached<sup>17</sup>. The coverage information is from the model view, there is no information about how much of the user code has been traversed, but only that the model element was used. For a code view with detailed information about the coverage of the generated and the user code you need to use code coverage.

Limitations:

- Overridden operations can not be distinguished
- Overloaded operations can not be distinguished

<sup>15</sup>Note that with the property `TestConductor.Settings.ReportLocation` (see section General Properties on page 99) a user can specify a dedicated report location)

<sup>16</sup>For operations only token oriented activities are allowed.

<sup>17</sup>For some statechart and activity elements which are directly dependent of other elements Rhapsody does not generate animation messages which are used by TestConductor to measure the coverage. For these elements TestConductor applies a set of dependency rules to derive the coverage.



- Model elements for which animation is switched off appear as 'not covered' even if they were used in the test execution.

## Choosing the Coverage Kind for Model Coverage

TestConductor supports four different kinds of coverage measures, which can be chosen using tag “CoverageKind” of the <<TestingConfiguration>> code generation configuration.

- SUT flat (Default): Only coverage of the top level class of the SUT is measured, i.e. states, transitions, and operations of parts of the SUT are not considered. Coverage of model elements of TestComponents is also not measured.
- SUT hierarchical: Coverage of the SUT is measured in a hierarchical manner, i.e. also states, transitions, and operations of parts of the SUT are hierarchically regarded for coverage measure. Coverage of model elements of TestComponents is again not measured.
- TestContext flat: Coverage is measured in terms of all states, transitions, and operations defined at the first decomposition level of the TestContext, i.e. all states, transitions, and operations of the direct parts of the TestContext are considered.
- TestContext hierarchical: all states, transitions, and operations in the hierarchical structure of the TestContext are considered in coverage measure.

Despite choosing the coverage kind in general, the coverage scope can be fine-tuned by adding <<considerForCoverage>> and <<considerNotForCoverage>> dependencies on the respective model elements (classes, objects and files) to the TestContext.

## Model Coverage Measurement and Animation Instrumentation

TestConductor uses the Rhapsody animation to determine the coverage of model elements, therefore usage of model coverage requires the 'Instrumentation Mode' of the configuration set to 'Animation'. With this setting the Rhapsody code generation instruments the code with additional animation code, TestConductor listens at runtime to animation messages sent by the application and uses these messages to determine the model coverage. There are some elements for which the Rhapsody code generation does not generate explicit animation messages because the code is included in a block of an element with animation message (e.g. in transition chains with junction connectors only the first transition is annotated with animation code, the code of the other transitions is included in the code block of the first transition). For these scenarios TestConductor applies a set of dependency rules to derive the coverage of these elements from the coverage of elements with animation message.

## Traceability of Coverage Items

The html report contains links for the navigation from the report to the Rhapsody model: When clicking on the link of an operation, event, state or transition, the corresponding model element is highlighted in the Rhapsody browser. Highlighting model elements will work only if JavaScript is enabled in the browser and no pop up blocker is active. For Internet Explorer 7 and up, protected mode has to be disabled (Tools->Internet Options->Security).

To highlight the model element, a JavaScript script is used which sends a command to the running Rhapsody application using a TCP/IP port. Per default, port number 50001 is used for this communication. If this port is not available or when running different instances of Rhapsody on the same machine, the port number can be changed so each running instance

of Rhapsody can communicate with the individual model coverage report. To do this, open the TestConductor main dialog by Rhapsody menu Tools->Test Conductor, and change the “Port number for coverage reports” and click OK. After this, double click the ModelCoverageResult in the Rhapsody model to open the report with the modified port number. Allowed port numbers are between 1024 and 65535.

To change the port number when the report is already opened in the browser, change the port in the TestConductor main dialog and also in the edit field in the html report to the same number.

A different default port number can be defined using the environment variable PORTSNOOPERPORT: Set this variable to the new default number before starting Rhapsody.

## Computing Requirement Coverage

(see also TestingCookbook:

- “How can I compute requirement coverage?”

)

## Computing Requirement Coverage for TestCases and TestContexts

Beyond measuring and reporting model element coverage for executed TestCases and TestContexts, TestConductor offers also the measurement of the dynamic requirement coverage for the executed TestCases and TestContexts.

Precondition for measuring requirements coverage by individual TestCases and TestContexts is the linkage of operations, states and transitions with requirements in the Rhapsody model. Stereotyped dependencies targeting requirements can be added to model elements in order to establish e.g. traceability or to express that certain model elements contribute to establishing a particular requirement.

Requirement coverage measurement is enabled by setting both tags “ComputeModelCoverage” and “ComputeRequirementCoverage” on the <<TestingConfiguration>> code generation configuration.

TestConductor optionally regards such dependencies in order to calculate requirement coverage based upon model coverage information. The user can define the stereotypes to be considered in requirement coverage calculation using property `ModelBasedTesting.Settings.StereotypesForDependenciesToRequirements` of the code generation configuration. Consideration of multiple stereotypes can be achieved by listing the stereotypes in a comma separated list. Per default, stereotypes `trace` and `satisfy` are regarded.

TestConductor provides also two properties for the user in order to configure the requirement coverage scope for TestConductor. So the user can specify the packages (and their sub-packages), whose requirements shall be regarded at the requirement coverage calculation within the property `ModelBasedTesting.Settings.RequirementCoverageRequirementsScope` of the code generation configuration. The setting of multiple packages (and their sub-packages) can be archived via a comma separated list of the fully qualified package paths, e.g.

`"RequirementsAnalysisPkg::Requi`

```
requirementsPkg::SecSysReqs, TestPkg::RequirementsPkg::SecSysTestReqs".
```

The second property 'ModelBasedTesting.Settings.RequirementCoverageRegardedTags' of the same code generation configuration, specifies via a "requirement tag with name and value" those requirements within the pre-selected packages, who shall be considered at the requirements coverage calculation. Again the setting of multiple "requirement tags with name and value" can be archived via a comma separated list, e.g.

```
"RequirementType=functional, RequirementType= additional".
```

TestConductor provides additionally two properties for the user in order to configure the model elements scope for the TestConductor requirement coverage calculation. So the user can specify the packages (and their sub-packages), the classes (blocks) or actors, whose model elements shall be regarded at the requirement coverage calculation within the property 'ModelBasedTesting.Settings.RequirementCoverageModelElementsScope' of the code generation configuration. The setting of multiple packages (and their sub-packages), classes (blocks) or actors can be archived via a comma separated list of the fully qualified package, classes (blocks) or actor paths, e.g.

```
"DesignSynthesisPkg::SecSysControllerPkg::SecSysController, ActorPkg::CardReaderEntry".
```

The second property 'ModelBasedTesting.Settings.RequirementCoverageExcludedMetaClasses' of the same code generation configuration, specifies via an "excluded meta classes tag with name and value" those meta classes within the pre-selected packages, classes (blocks) or actors, who shall be excluded from (not considered at) the requirements coverage calculation. Again the setting of multiple "excluded meta classes tags with name and value" can be archived via a comma separated list, e.g.

```
"Attribute, Class, Event".
```

TestConductor distinguishes two kinds of requirement coverage by TestCases:

- **full coverage**

All model elements depending on a particular requirement (w.r.t. specified dependency stereotypes) are covered by a TestCase or TestContext. The TestCase or TestContext then fully covers the requirement – a dependency stereotyped <<fully>> on the requirement is added to the Requirement Coverage Result Report of the TestCase or TestContext.

- **partial coverage**

Not all model elements depending on a particular requirement (w.r.t. specified dependency stereotypes) are covered by a TestCase or a TestContext. The TestCase or TestContext then only partially covers the requirement – a dependency stereotyped <<partially>> on the requirement is added to the Requirement Coverage Result Report of the TestCase or TestContext.

## Transitivity of Dependencies (Refinement of model elements and requirements)

Via the TestConductor property "ModelBasedTesting.Settings.RequirementCoverageTransitivityOfDependencies" the support for the refinement of

model elements and the refinement of requirements (for the TestConductor requirement coverage calculation) can be switched on or off.

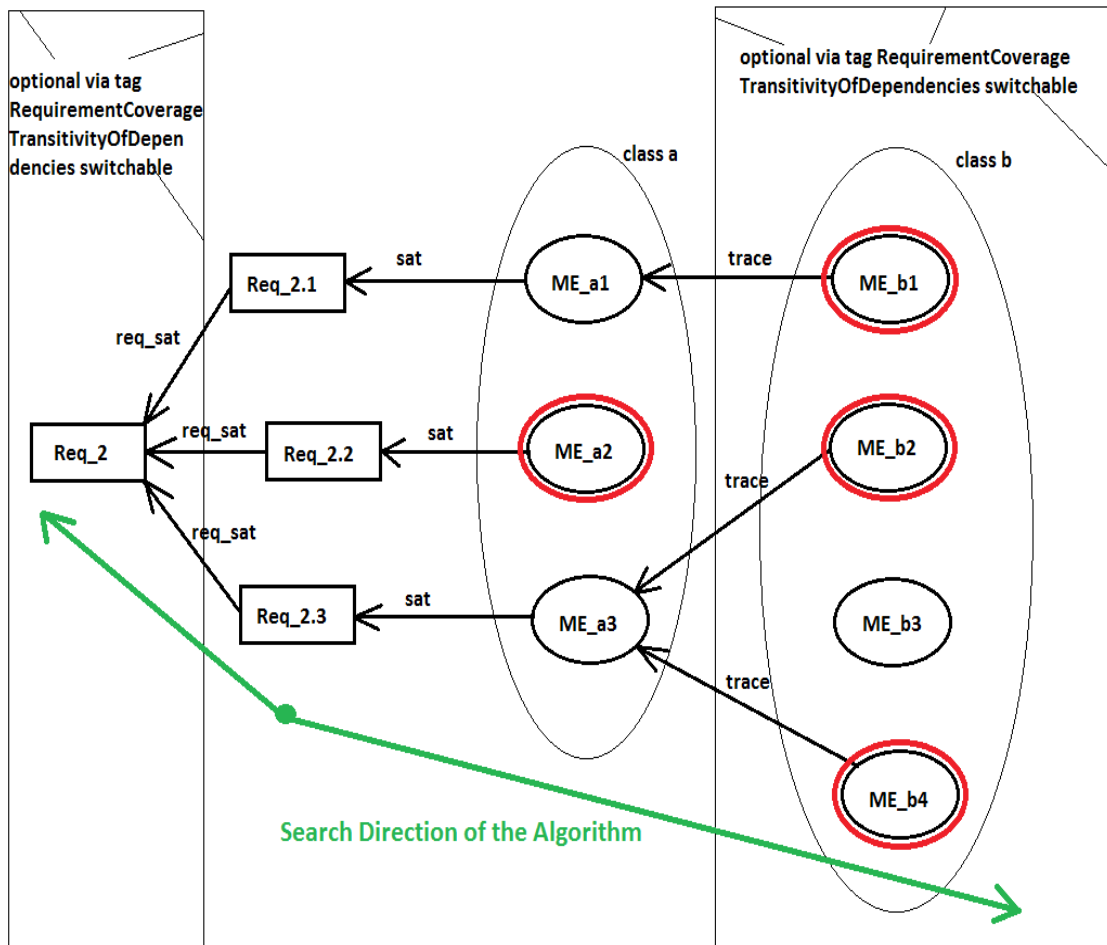


Figure 16: Transitivity of Dependencies (Refinement of model elements and requirements)

The figure above shows an application for the refinement of requirements and model elements.

If transitivity of dependencies is switched off, ME\_a1 is connected to Req\_1.2, ME\_a2 is connected to Req\_2.2 and ME\_a3 is connected to Req\_2.3. But if transitivity of dependencies is switched on, the refinements of ME\_a1 by ME\_b1 and of ME\_a3 by ME\_b2 and ME\_b4 are considered during the requirement coverage calculation. This means, the requirement Req\_2.3 for example is only fully covered by a TestCase or a TestContext if both model elements ME\_b2 and also ME\_b4 are covered by this TestCase or TestContext (if class B is within the model element scope).

If transitivity of dependencies is switched off the connections between the requirements Req\_2.1, Req\_2.2 and Req\_2.3 to the requirement Req\_2 are not considered. But if transitivity of dependencies is switched on the requirement Req\_2 is refined by the requirements Req\_2.1, Req\_2.2 and Req\_2.3 and these refinements are considered at the requirement coverage calculation. This means, the requirement Req\_2 is only fully

covered by a TestCase or a TestContext, if the requirements Req\_2.1, Req\_2.2 and Req\_2.3 are as well fully covered by this TestCase or TestContext (if Req\_2 is within the requirements scope).

An example explaining the transitivity of dependencies related to the handling of refined model elements: A model element "A1" (class A) has a satisfy dependency to a requirement "req\_17". And there is a refinement of the model element "A1", as the two model elements "B1" and "B2" (class B) have both a trace dependency to the model element "A1". And in the same way model element "B1" is refined by the model elements "C1" and "C2" (class C) and model element "B2" is refined by the model elements "C3" and "C4" (class C). If the property "RequirementCoverageTransitivityOfDependencies" is set and "RequirementCoverageModelElementScope" is only set to "class B", then the requirement "req\_17" is fully covered by a TestCase, if the TestCase covers all of the model elements "B1" and "B2". But if the TestCase covers only one of the model elements "B1" and "B2", then the requirement "req\_17" is only partially covered.

An example explaining the transitivity of dependencies related to the handling of refined requirements: A model element "A1" (class A) has a satisfy dependency to a low level requirement "req\_LL\_11". And this low level requirement "req\_LL\_11" has on his part again a satisfy dependency to a high level requirement "req\_HL\_01". A model element "A2" (class A) has a satisfy dependency to a low level requirement "req\_LL\_22". And this low level requirement "req\_LL\_22" has on his part again a satisfy dependency to a high level requirement "req\_HL\_01". If the property "RequirementCoverageTransitivityOfDependencies" is set and the high level requirement "req\_HL\_01" is within the requirement scope, then this high level requirement "req\_HL\_01" is fully covered by a TestCase, if this TestCase covers both the low level requirements "req\_LL\_11" and also "req\_LL\_22" fully. But if a TestCase covers either the low level requirement "req\_LL\_11" or "req\_LL\_22" only partially, then this high level requirement "req\_HL\_01" is also only partially covered by this TestCase.

## Computing Code Coverage

(see also Sample-models:

- CSamples/TestConductor/CModelCodeCoverage
- CppSamples/TestConductor/CppModelCodeCoverage

)

Besides computing model coverage of TestCases, TestConductor can also compute the achieved code coverage of TestCases (for C and C++ only). In order to turn on code coverage, the tag "ComputeCodeCoverage" of the TestingConfiguration must be turned on'.

If the tag is checked, when building TestCases TestConductor instruments the test executable s.t. during test execution code coverage information is computed. After TestCase execution, the computed results are added as an html report to the model. The result report both contains summary information (e.g. percentage of statement coverage, decision/condition coverage, modified condition/decision coverage (MC/DC)) as well as detailed information about each source line. If a TestCase is executed for multiple code generation configurations, for each configuration a separate code coverage report is stored in the model.

Please note, only implementation files are instrumented for computation of code coverage. For code in specification files, for example C++ inline functions, no coverage information is generated and the coverage report does not contain information if the code in specification files has been covered by the tests or not. The *Source Code* section of the code coverage report contains a list of not instrumented functions in specification files. For C++, state\_IN methods per default are generated inline into the specification files. To be able to compute coverage information for state\_IN methods, the property `CPP_CG::Class::IsInOperation` can be set to virtual to generate these methods into the implementation file.

Similar to model coverage, four different kinds of coverage measures are supported and can be chosen by setting the tag “CoverageKind” of the TestingConfiguration. For details, see previous section Choosing the Coverage Kind for Model Coverage on page 81.

Additional options for code coverage can be specified using a xml file. The location of the file has to be entered in the tag “CodeCoverageOptionsFileName” of the TestingConfiguration. In this tag, either the full path name of the options file or its path relative to the location of the Rhapsody project file can be specified.

The options file can be used to

- Define additional implementation files which shall be instrumented for code coverage. Either the path of the file or the model element can be defined:
  - The files can be defined by the absolute path or by the path relative to the code generation main folder (location of the Makefile).  
Note: Supported are only files generated for model elements.
  - Model elements can be defined by their full model path.
- Specify include paths.
- Specify defined macros.
- Specify details of the used compiler and compile environment.

A template of the options file showing the supported options is located in the TestConductor installation folder: File “TCCodeAnnotationOptions.xml”.

## TestConductor code coverage criteria

TestConductor reports code coverage according to different kinds of coverage criteria. Depending on the coverage criteria, one coverage item comprises of one or more individual coverage goals. TestConductor reports if a coverage item is not covered, partially covered (at least one of its coverage goals has been covered) or completely covered (all of its coverage goals have been covered).

- **Statement Coverage**  
A statement is a statement in the sense of the C/C++ definition of this term. A statement is considered as covered if it has been called during execution of the tests.
- **Function Coverage**  
A function is a function or operation in the generated code. A function is considered as covered if it has been called during execution of the tests. To fulfill this coverage criteria, it is not necessary the function or operation has returned.

- **Condition Coverage**

In order to define what conditions and decisions are, a notion of atomic Boolean expressions is needed. Those atomic Boolean expressions are built using relational operators such as < (less than), > (greater than), == (equality), and so on. Examples for atomic Boolean expressions are  $x > 7$ ,  $(z + 1) \neq y$ , and also  $x < y < z$ .

TestConductor treats atomic Boolean expressions and negations of these expressions as conditions. Therefore, not only expressions like  $x > 7$ ,  $z + 1 \neq y$  are conditions but also expressions like  $!(x > 7)$  or  $!b$ . Note that, in general, conditions can not contain other conditions. This particularly means that expressions like  $!(a < 7)$  or  $(x < y < z)$  induce single conditions. As a special rule, TestConductor ignores constant expressions. For instance, expression  $1 < 5$  will not be reported as separate condition, since it consists of literals only and the value of the relational operator is constantly true.

If a condition appears more than once in a Boolean expression, each occurrence is a distinct condition. Hence, in an expression " $x == 5 \ \&\& \ y == z \ || \ x == 5 \ \&\& \ y < 0$ ", there are four conditions, namely " $x == 5$ ", " $y == z$ ", " $x == 5$ " and " $y < 0$ ".

Each condition comprises of two coverage goals, a condition is completely covered by the tests only if both coverage goals are covered:

1. The condition is true
2. The conditions is false

- **Decision Coverage**

Intuitively, decisions are built on conditions, this means, decisions consist of one or more conditions. These conditions are connected using the Boolean connectives in C++ and C, namely, && (Boolean and), || (Boolean or), ! (Boolean negation) and, as a special rule, ^ (bitwise-xor). So, for example, the following Boolean expressions are decisions:

$x < 7 \ \&\& \ a == b$   
 $x \wedge y \ || \ !(x == 7 \ \&\& \ a + 7 == 25)$

Normally, in C++ or C programs, decisions are used as control expressions for choice points such as if- or while-statements. TestConductor analyzes the following statements for the occurrence of decisions:

- Control expression of if-statements
- Control expressions of iteration-statements while, do, and for
- The first operand of a conditional operator ( $cond ? x : y$ )
- Maximal Boolean expressions occurring in other statements such as assignments, function call etc.

Note that if an expression meets both the characteristics of a decision and a condition, for instance in a term  $if(a > b) \dots$ , then TestConductor reports the expression as decision only. From the above definitions for conditions and decisions, we obtain the following special examples:

```
x = a ^ b           /* a ^ b is a decision */
f(i && (ul1 || ul2)) /* i && (ul1 || ul2) is a
                    decision, i, ul1, ul2
                    are conditions */
```



Each decision comprises of two coverage goals, a decision is completely covered by the tests only if both coverage goals are covered:

1. The decision is true
2. The decision is false

- **Condition/Decision Coverage (C/DC)**

In TestConductor, Condition / Decision Coverage (C/DC) is defined as follows.

- Every decision has taken all possible outcomes (true, false) at least once.
- Every condition appearing in a decision has taken all possible outcomes (true, false) at least once.

This is a standard definition for C/DC. In the C++ and C language, expressions are calculated short-circuit. This particularly means that Boolean expressions are evaluated from left to right, and an evaluation terminates when the outcome of an expression is determined. For example, evaluation of Boolean expression

$x > y \ \&\& \ z == c$

terminates after traversing  $x > y$  if  $x$  is less or equal  $y$ , as the outcome of the entire conjunction is obviously false. In this example, condition  $z == c$  is not evaluated at all.

For coverage of conditions, TestConductor takes such short-circuit calculations into account. This means that conditions can only be covered if evaluation reached their respective position in the surrounding expression, this means. there is no short-circuit until that position. In the above example, this means that condition  $z == c$  can only be covered in states where  $x$  is greater than  $y$ .

For decision evaluation, short-circuit calculation also plays a fundamental role. As the decision's conditions are evaluated from left to right and calculation terminates whenever the outcome of the decision is determined, one obtains a very important relation between decisions and their enclosed conditions. As a condition is traversed only if the outcome of its decision is not yet determined, the condition's evaluation can independently affect the decision's outcome.

From the above explanations, two additional properties for the C/DC definition of TestConductor can be stated, which are inherited from the short-circuit evaluation policy:

- Conditions in a decision can only be covered if the decision's outcome is not yet determined from the short-circuit left-to-right calculation.
- From the short-circuit evaluation setting, a condition can independently affect the decision's outcome if the left-to-right calculation reaches that condition.

Each C/DC coverage item comprises of one or more individual coverage goals, a C/DC coverage item is completely covered only if all its coverage goals are covered.

- **Modified Condition/Decision Coverage (MC/DC)**

Modified Condition / Decision Coverage (MC/DC) was originally defined for non-short-circuit evaluation languages such as ADA. In such languages, neither a calculation order nor an early termination property was defined. Therefore, the additional property – compared with C/DC – that “every condition in a Boolean expression in the program has been shown to independently affect that expression's outcome” required dedicated definitions suitable for non-short-circuit



evaluation languages. In particular, to show independence of the entire expression's outcome, it was defined to hold all conditions but the one of interested fixed while toggling the relevant one. This was done to preclude effects of other conditions to the expression's outcome, masking the one of the relevant. As described above, the evaluation-semantics for Boolean expressions in C++ and C is short-circuit. This imposes a calculation order on such expressions and defines an early termination property. In particular, as defined in TestConductor conditions in a decision can only be covered if the decision's outcome is not yet determined from the short circuit left-to-right calculation. Obviously, this relaxes the requirement to keep conditions fixed while focusing on the relevant one. In other words, the short-circuit evaluation property ensures that conditions independently affect the decision's outcome.

As a conclusion, in a short-circuit setting as defined in TestConductor C/DC and MC/DC induce the same tests to be performed in order to fulfill the respective requirements. There are no additional test vectors needed when improving from C/DC to MC/DC. This is why TestConductor does not offer a separate report section for MC/DC, as this would be the same as for C/DC.

- **Relational Operator**

Consider the expression  $(i > 5)$  which may be erroneously implemented as  $(i \geq 5)$ . In this case, the wrong relational operator "greater or equal than" was chosen in the code. In order to detect such faults it is not sufficient just to test the cases where the relational operation became true and false, it is necessary to check the "boundaries" of the relational operator. To detect the problem for the above implementation, the following valuations for  $i$  need to be tested:

<b>i</b>	<b>Specification <math>i &gt; 5</math></b>	<b>Implementation <math>i \geq 5</math></b>
4	False	False
5	False	True
6	True	True

This table shows that a test run assigning value 5 to  $i$  would reveal the wrong implementation. TestConductor checks that all needed tests needed for full testing of relational operators have been run. This requires a boundary check for each relational operator within the C++ or C code. The general form of this problem is expressed by  $\langle \text{expr1} \rangle \langle \text{relop} \rangle \langle \text{expr2} \rangle$  where  $\langle \text{expr1} \rangle$  and  $\langle \text{expr2} \rangle$  are arbitrary expressions and  $\langle \text{relop} \rangle$  is a relational operator ( $<$ ,  $\leq$ ,  $=$ ,  $\neq$ ,  $\geq$ ,  $>$ ).

An optimal boundary check is done by executing a set of stimuli vectors which lead to evaluations of the relevant relational operator while covering all of the following properties:

- P1:  $(\text{expr1}) - (\text{expr2}) > 1$  (operation became true)
- P2:  $(\text{expr1}) - (\text{expr2}) < -1$  (operation became false)
- P3:  $(\text{expr1}) - (\text{expr2}) = -1$  (boundary check)
- P4:  $(\text{expr1}) - (\text{expr2}) = 0$  (boundary check)
- P5:  $(\text{expr1}) - (\text{expr2}) = 1$  (boundary check)

For each relational operator, TestConductor reports if the tests which have been run perform an optimal boundary check. Each relational operator comprises of

five coverage goals, a relational operator is completely covered by the tests only if all five coverage goals are covered:

1. The operation is true
2. The operation is false
3. (left-right) == -1
4. (left-right) == 0
5. (left-right) == 1

For an example that shows how to use code coverage for C, please try sample “CModelCodeCoverage” in the folder <Samples/CSamples/TestConductor/CModelCodeCoverage>.

For an example that shows how to use code coverage for C++, please try sample “CppModelCodeCoverage” in the folder <Samples/CppSamples/TestConductor/CppModelCodeCoverage>.

Restrictions regarding applicability of code coverage computation can be found in the document <Rhapsody install>/Doc/pdf\_docs/CodeCoverage\_Limitations.pdf.

**Note:** Computation of code coverage for Grey Box TestArchitectures does not filter the instrumentation of the <<TestSUT>> for Grey Box testing. Code coverage is measured for the executed code, i.e. the code that has been instrumented for Grey Box testing purposes.

## Command Line Execution

TestConductor can update, build, and execute TestCases, TestContexts or TestPackages from the command line. Command line execution can either be performed by using the command line feature of rhapsody.exe or by using rhapsodycl.exe.

### Command Line Syntax for rhapsody.exe

You can use following syntax to execute tests from the command line:

- “<Rhapsody executable>” -cmd=open <model file>  
-cmd=call "rtc TC\_COMMAND TC\_ELEMENT" -cmd=save -  
cmd=exit

where TC\_COMMAND is one of the following TestConductor commands

- update\_build\_execute  
performs an update, then a build, and then an execute on the specified test element.
- update\_build  
performs a build, and then an execute on the specified test element.
- update

performs an update on the specified test element.

- `checkUpdateRequired`

queries if an update of TC\_ELEMENT is required. If an update is required, the result TRUE is written to the log file cl.log (see below), otherwise FALSE.

- `build_execute`

performs a build and then an execute on the specified test element

- `build`

performs a build on the specified test element.

- `execute`

performs an execute on the specified test element.

- `clean_update_build_execute`

performs a clean, then an update, then a build, and then an execute on the specified test element.

- `clean_update_build`

performs a clean, then an update and then a build on the specified test element.

- `clean_update`

performs a clean and then an update on the specified test element.

- `clean`

performs a clean on the specified test element.

and TC\_ELEMENT is either “all” or the full path name of a TestCase, a TestContext or a TestPackage.

TestConductor logs in the file “cl.log” in the project folder the command line actions together with the result (SUCCEEDED or FAILED for actions, TRUE or FALSE for queries).

**Note:** `-cmd=save` needs to be defined in order to permanently update the link to the HTML test result report (controlled file) and the Verdict tag under it. At this time older test result files will not be overwritten, but a new file with an incremented number will be created. In case the model will not be saved before exiting, still the old or none result file will be referenced.

Examples:

- `"<full Rhapsody path>\rhapsody.exe" -cmd=open D:\CppCashRegister_rpy\CppCashRegister.rpy -cmd=call "rtc update_build_execute TPkg_CashRegister::TCon_CashRegister::tc_SimpleStart"`

-cmd=save

updates, builds, and then executes the TestCase "tc\_SimpleStart" of the model CashRegister.

- "<full Rhapsody path>\rhapsody.exe" -cmd=open D:\CppCashRegister\_rpy\CppCashRegister.rpy -cmd=call "execute TPkg\_CashRegister::TCon\_CashRegister" -cmd=save executes the TestContext TCon\_CashRegister of the model CashRegister.
- "<full Rhapsody path>\rhapsody.exe" -cmd=open D:\CppCashRegister\_rpy\CppCashRegister.rpy -cmd=call "rtc build\_execute TPkg\_CashRegister" -cmd=save builds and executes the TestPackage TPkg\_CashRegister of the model CashRegister.

## Command Line Syntax for rhapsodycl.exe

If you run the command line version of rhapsody, rhapsodycl.exe, you can execute the same TestConductor commands as for rhapsody.exe. In rhapsodycl.exe, the TestConductor commands are invoked by specifying

- -cmd=call "rtc TC\_COMMAND TC\_ELEMENT"

in the command line prompt of rhapsodycl.exe (or in a file containing the list of commands for rhapsodycl.exe). TC\_COMMAND can be one of the following TestConductor commands:

- update\_build\_execute  
performs an update, then a build, and then an execute on the specified test element.
- update\_build  
performs a build, and then an execute on the specified test element.
- update  
performs an update on the specified test element.
- checkUpdateRequired  
queries if an update of TC\_ELEMENT is required. If an update is required, the result TRUE is written to the log file cl.log (see below), otherwise FALSE.
- build\_execute  
performs a build and then an execute on the specified test element
- build  
performs a build on the specified test element.
- execute

performs an execute on the specified test element.

- `clean_update_build_execute`

performs a clean, then an update, then a build, and then an execute on the specified test element.

- `clean_update_build`

performs a clean, then an update and then a build on the specified test element.

- `clean_update`

performs a clean and then an update on the specified test element.

- `clean`

performs a clean on the specified test element.

and `TC_ELEMENT` is either “all” or the full path name of a `TestCase`, a `TestContext` or a `TestPackage`.

`TestConductor` logs in the file “`cl.log`” in the project folder the command line actions together with the result (SUCCEEDED or FAILED for actions, TRUE or FALSE for queries).

Examples (we assume that `rhapsodycl.exe` is already started):

- `"> -cmd=call "rtc update_build_execute  
TPkg_CashRegister::TCon_CashRegister::tc_SimpleStart"`  
updates, builds, and then executes the `TestCase` “`tc_SimpleStart`” of the model `CashRegister`.
- `"> -cmd=call "execute  
TPkg_CashRegister::TCon_CashRegister"`  
executes the `TestContext` `TCon_CashRegister` of the model `CashRegister`

**Note:** `TestConductor` does not support `rhapsodycl.exe` on Linux.

## Test Execution Report

After test execution all test reports are written in the same manner as described under “`TestCase Execution`”, “`TestContext Execution`” and “`TestPackage Execution`”.

## TestCase Execution on Targets

In addition to executing `TestCases` on the host environment, `TestCases` can also be executed on the target environment. The necessary steps are target environment specific and are further described in the following documents:

- [Testing\\_with\\_RTC\\_on\\_a\\_Linux\\_Target.pdf](#) (Linux)
- [Testing\\_with\\_RTC\\_on\\_a\\_VxWorks\\_Target.pdf](#) (VxWorks)

- Testing with TestConductor on an Integrity Target.pdf (Integrity)
- Testing with TestConductor on a small target.pdf (generic environment)

# Test Management

---

TestConductor is a fully integrated add-on solution for Rhapsody. All relevant test data like the TestArchitecture, TestCases and their TestScenarios, test configurations and test results are stored in the model. Navigation to all the elements can be done via the usual capabilities of the Rhapsody browser.

## Managing Test Data

With this tight integration you have all the possibilities you already know from other elements like classes, package and so on, e.g.:

- Copy, paste, delete
- Create units for TestComponents, TestContext, SUT and TestComponentInstances
- Load / unload TestPackages, TestComponents, TestContext, SUT and TestComponentInstances
- Requirements management
- Configuration management
- Documentation

## Linking TestCase to Requirements

(see also TestConductor Tutorials:

- TestConductor\_Tutorial\_C.pdf
- TestConductor\_Tutorial\_Cpp.pdf

)

TestCases can be linked to their requirements which are defined in the model. This can be done by using *TestObjective* to link model elements to the related requirements.

TestObjective is a new term on dependency.

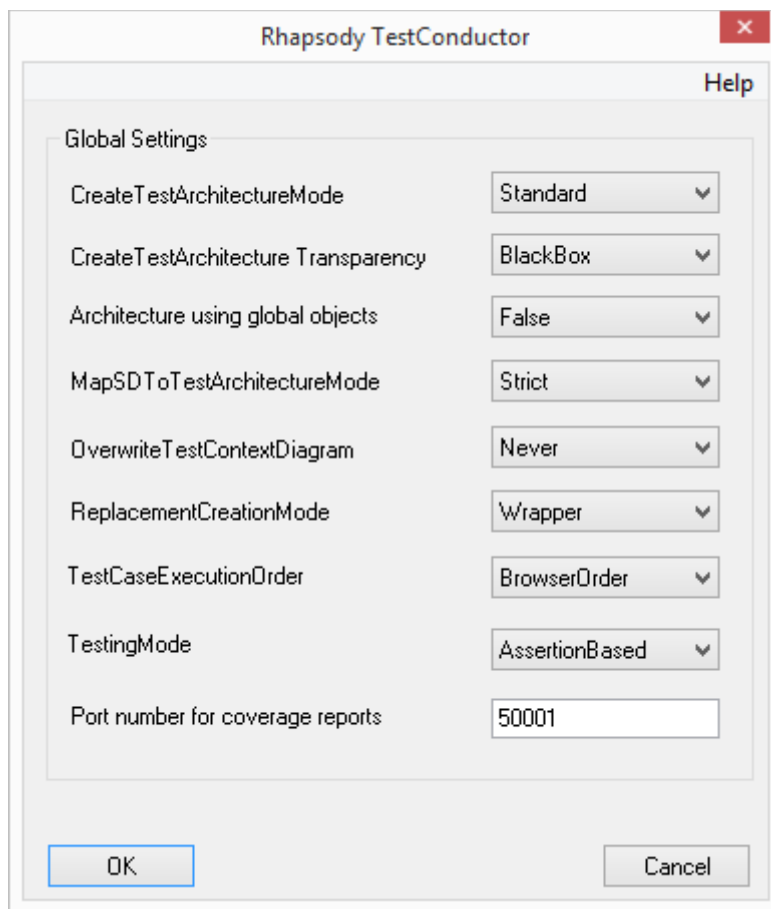
A TestObjective can be added to a TestCase by right-clicking the TestCase in the browser and choosing 'Add New->TestingProfile->TestObjective' from the context menu. A dependency selection dialog will open and the respective requirement can be selected as 'Depends on' of the TestObjective.

The Rhapsody Testing Profile offers matrix layout 'TestRequirementCoverage' with appropriate new term 'TestRequirementMatrix' on matrix view for documenting requirement coverage by TestCases according to TestObjective relations.

Furthermore, the TestConductor addon provides templates for ReporterPLUS (see section Generating Test Reports with Rhapsody ReporterPLUS on page 102) as well as for Rhapsody Publishing Engine (see section Generating Test Reports with Rational Publishing Engine on page 104) for report generation regarding requirement coverage by TestCases according to TestObjective relations.

## TestConductor Dialog

The TestConductor main dialog provides some global TestConductor settings and help functions by selecting **Tools > TestConductor** from the Rhapsody tools menu:



*Figure 17: TestConductor Main Dialog*

The dialog offers the possibility to set some global TestConductor settings and to open TestConductor's tutorial by selecting **Help > Tutorial**. The global settings that can be changed in this dialog are explained in the next section **TestConductor Settings**.

## TestConductor Settings

TestConductor provides a range of global and also TestCase specific settings. The settings are in most cases stored as properties in the model.

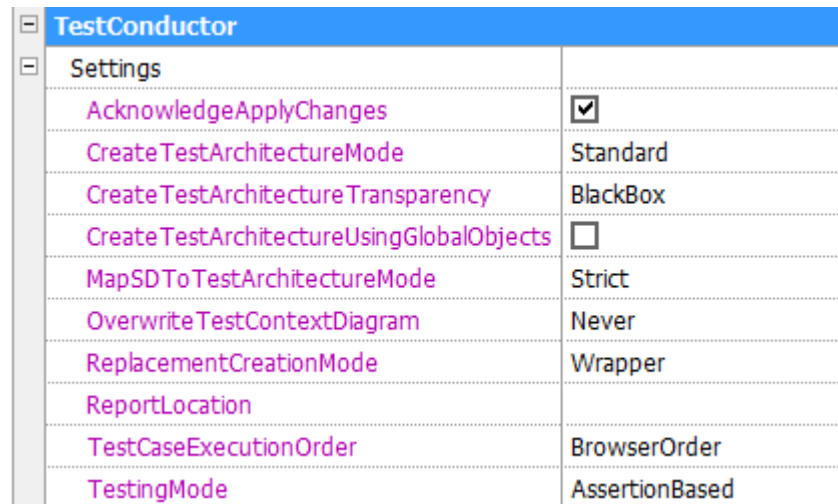


[-] TestConductor		
[-] SDInstance		
ExecutionIterations	1	
ExecutionMode	Monitor	
ExecutionOrder	Linear	
ParameterValues		
[-] Settings		
AcknowledgeApplyChanges	<input checked="" type="checkbox"/>	
CreateTestArchitectureMode	Standard	
CreateTestArchitectureTransparency	GreyBox	
CreateTestArchitectureUsingGlobalObjects	<input checked="" type="checkbox"/>	
MapSDToTestArchitectureMode	Strict	
OverwriteTestContextDiagram	Never	
ReplacementCreationMode	Wrapper	
ReportLocation		
TestCaseExecutionOrder	BrowserOrder	
TestingMode	AssertionBased	
[-] TestCase		
AnimatedSUT	Automatic	
ATGTestCase	<input type="checkbox"/>	
CallOperationsOnlyWhenCallstackEmpty	<input checked="" type="checkbox"/>	
ComputeCoverage	<input type="checkbox"/>	
CoverageKind	SUT flat	
CreateSDForFailedSDInstance	<input type="checkbox"/>	
DriveMessagesToTestComponents	<input type="checkbox"/>	
ExecuteTestWithTracer	<input type="checkbox"/>	
ExecutionAnimationStartedTimeout	20	
ExecutionAnimationStoppedTimeout	20	
ExecutionFirstIdleTimeout	20	
ExecutionIdleTimeout	600	
MultipleConditionCheck	<input type="checkbox"/>	
ResetAppBeforeStartTest	<input checked="" type="checkbox"/>	
TerminateAppOnQuitTest	<input checked="" type="checkbox"/>	
Tolerances		
UseOM_RETURN	<input type="checkbox"/>	
WriteTestExecutionLogFile	<input type="checkbox"/>	
[-] TestContext		
TestContextExecution_PostTestCaseOperation		
TestContextExecution_PreTestCaseOperation		
TestContextExecution_RestartExecutable	<input checked="" type="checkbox"/>	

Figure 18: Properties - TestConductor

## General Properties

TestConductor provides some general settings that change the general behavior of TestConductor. These settings have to be done via properties on TestPackage level. Open the **Feature** dialog of a TestPackage, select the **Properties** tab, switch in the dropdown combo box **View** to *All* and navigate to the metaclass `TestConductor::Settings`



TestConductor	
Settings	
AcknowledgeApplyChanges	<input checked="" type="checkbox"/>
CreateTestArchitectureMode	Standard
CreateTestArchitectureTransparency	BlackBox
CreateTestArchitectureUsingGlobalObjects	<input type="checkbox"/>
MapSDToTestArchitectureMode	Strict
OverwriteTestContextDiagram	Never
ReplacementCreationMode	Wrapper
ReportLocation	
TestCaseExecutionOrder	BrowserOrder
TestingMode	AssertionBased

Figure 19: Properties `TestConductor.Settings`

`TestConductor::Settings::AcknowledgeApplyChanges`

If this property is switched on, TestConductor requires an explicit acknowledge from the user each time a `SDInstance` has been changed. If the property is switched off, changes of `SDInstances` are acknowledged implicitly.

This property is irrelevant in assertion based testing mode.

`TestConductor::Settings::CreateTestArchitectureMode`

This property controls the behavior of the TestConductor function “Create TestArchitecture”. If this property is set to “Standard”, each time “Create TestArchitecture” is performed TestConductor creates a component and a configuration for the newly created TestArchitecture using the default property settings for components and configurations. If the property is set to “Advanced”, each time “Create TestArchitecture” is performed TestConductor opens a dialog which allows to specify from which of the existing components/configurations the property values of the newly created component/configuration shall be derived. Furthermore, if the property is set to “Advanced” and `TestConductor::Settings::TestingMode` is “AssertionBased”, TestConductor offers the user a possibility to define the kind of each TestComponent in the TestArchitecture to be created.

By default this property has the value “Standard”.

`TestConductor::Settings::CreateTestArchitectureTransparency`

By default, TestArchitectures are created as 'BlackBox' architectures, i.e. the SUT is only external communication of the SUT is observable for testing. Internal communication such as self invocation of operations, communication among parts of the SUT is not considered in sequence diagram TestCases.

If CreateTestArchitectureTransparency is set to 'GreyBox', then a copy of the selected SUT will be created in the TestArchitecture that can be instrumented for testing purposes. Testing such a grey box <<TestSUT>> replacement instead of the original SUT model element enables TestConductor to instrument also the SUT model elements with assertions, s.t. self messages and communication among parts of the SUT can be considered in TestCases.

TestConductor::Settings::CreateTestArchitectureUsingGlobalObjects

Since Rhapsody 8.1.4, TestArchitecture creation can optionally use global objects instead of parts for SUT classes and TestComponentInstances. Fundamental support for global instantiation outside the TestContext gives way for grey box testing of implicit objects and stubbing of implicit objects and in particular also <<Singleton>> objects. Note, that parts of class can't be associated with global objects – at least, such associations can't be instantiated using links, since such links would cross class boundaries of the composite parent class of the involved parts. On the other hand 'classical' TestArchitectures using part instantiation, can't deal with implicit objects and singleton objects in TestComponent roles. Thus, it is recommended to use global object instantiation if implicit objects or singleton objects are involved in the testing process.

TestConductor::Settings::MapSDToTestArchitectureMode

This property controls the behavior of the TestCase wizard when a TestCase is created for an existing sequence diagram. If the value of this property is set to “Strict”, only those TestArchitectures are considered to be suitable for the new TestCase that contain at least on SUT instance of one of the classes of the life lines of the original sequence diagram. If the value of this property is set to “Weak”, also all TestArchitectures are considered to be suitable that does not contain a SUT instance of one of the classes of the life lines of the original sequence diagram, but for which the same message exchange is possible as in the original sequence diagram.

TestConductor::Settings::overwriteTestContextDiagram

This property controls the creation of TestContextDiagrams when performing an “Update TestArchitecture” on a TestContext. If this property is set to “Never”, each time “Update TestArchitecture” is performed a new TestContextDiagram is added to the existing TestContextDiagrams, i.e., existing TestContextDiagrams are not overwritten. If this property is set to “askUser”, each time “Update TestArchitecture” is performed TestConductor asks if an existing TestContextDiagram shall be replaced with a new one. If this property is set to “Always”, each time “Update TestArchitecture” is performed TestConductor replaces an existing TestContextDiagram with a new one.

By default this property has the value “Never”.

TestConductor::Settings::ReportLocation

With this property<sup>18</sup> TestConductor can be instructed to store test reports and results not in the default location directly underneath the test element (TestPackage, TestContext, TestCase) but at a location chosen by the user. The location has to be a (test-) package, which will be created if not existing yet. For nested packages the qualified name has to be specified using the delimiter '::' (e.g. “MyResults::Results\_MR1”).

Affected by this property are Test Execution Results, Model Coverage Results, Requirement Coverage Results and Code Coverage Results. Underneath the test element a hyperlink will be created<sup>19</sup> targeting the actual result. If the property expression can not be parsed or the specified package could not be created, the results will be saved at the default location underneath the test element.

Beside fixed package names TestConductor provides the following keywords which will be substituted with the appropriate names of the execution context:<sup>20</sup>

`$TESTPACKAGENAME`: Will be substituted by the name of the TestPackage<sup>21</sup> of the executed element.

`$TESTCONTEXTNAME`: Will be substituted by the name of the TestContext of the executed element. Will be ignored for TestPackage results.

`$TESTCASENAME`: Will be substituted by the name of the executed TestCase. Will be ignored for TestPackage and TestContext results.

`$CONFIGURATIONNAME`: Will be substituted by the name of the TestingConfiguration which was active at test execution. Will be ignored for TestPackage results.

`TestConductor::Settings::TestCaseExecutionOrder`

This property controls the execution order of TestCases when executing a TestContext. Possible values are “BrowserOrder” and “DeclarationOrder”, where “BrowserOrder” defines that TestCases are executed in the same order as they are displayed in the browser. “DeclarationOrder” defines execution in a user defined order. The declaration order can be specified by right-clicking “TestCases” and selecting “Edit TestCases Order” from the context menu (cf. section Ordering of TestCases on page 77).

By default this property has the value “BrowserOrder”.

`TestConductor::Settings::TestingMode`

By default, new TestArchitectures created with Rhapsody 7.6 or higher are created with testing mode set to assertion based testing, i.e., the property “TestConductor.Settings.TestingMode” is set to “AssertionBased”.

To create a new TestArchitecture for animation based testing, open the TestConductor main dialog by choosing “TestConductor” from the tools menu. In the upcoming dialog, select the testing mode you want TestConductor to apply for a newly created TestArchitecture. This setting does not affect any existing TestArchitecture.

---

<sup>18</sup>Property will be evaluated not only on project but also also on package level.

<sup>19</sup>Hyperlink will be created only for test elements which can be written.

<sup>20</sup>Note that the keywords may only be used to specify a complete package name, keywords may not be modified (e.g. correct: “Results::\$TESTPACKAGENAME”, incorrect: “Results:\$TESTPACKAGENAME\_1”)

<sup>21</sup>The outer TestPackage in assertion based mode

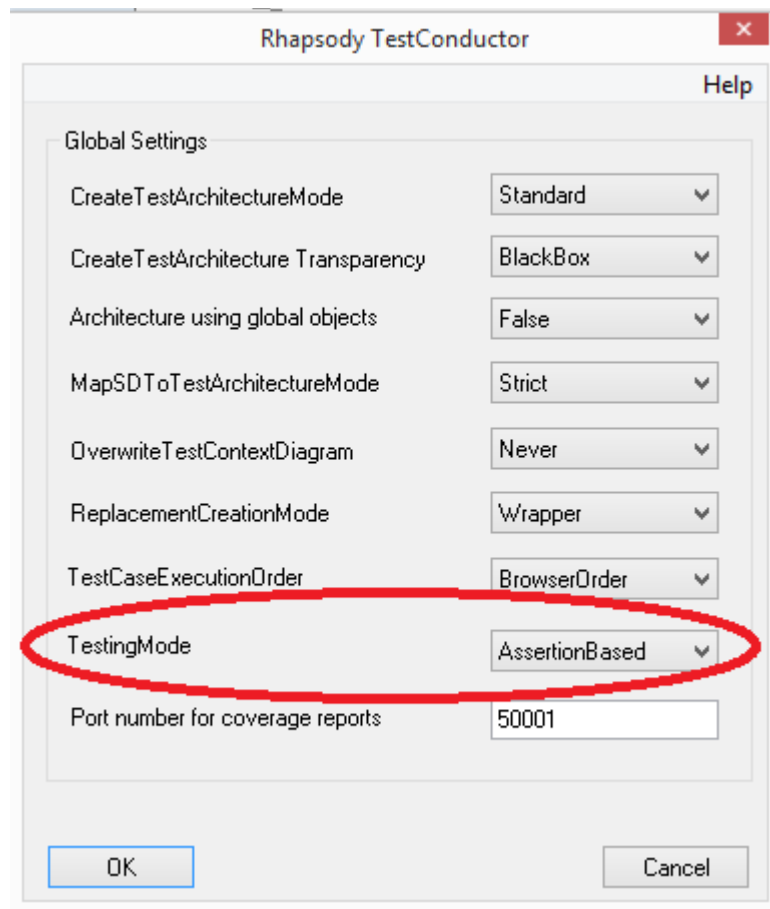


Figure 20: Setting TestingMode

## TestContext Properties

Also some properties for TestContexts can be set by the user. In order to change these properties, open the **Feature** dialog of a TestContext, select the **Properties** tab, switch in the dropdown combo box **View** to *All* and navigate to the metaclass

TestConductor::TestContext

[-] TestConductor	
[-] TestContext	
TestContextExecution_PostTestCaseOperation	
TestContextExecution_PreTestCaseOperation	
TestContextExecution_RestartExecutable	<input checked="" type="checkbox"/>

Figure 21: Properties TestConductor.TestContext

TestConductor.TestContext.TestContextExecution\_RestartExecutable

If this property is checked (true), for each TestCase during execution of the TestContext, the executable of the TestContext is restarted. If the property is not checked (false), the

TestCases are executed without restarting the executable after the previous TestCase has finished its execution.

`TestConductor.TestContext.TestContextExecution_PreTestCaseOperation`

If this property contains a name of an operation of the TestContext, for each TestCase during execution of the TestContext, **before** a TestCase is executed the operation specified in this property is called automatically. In the operation specified in this property, one can initialize or reset some variables that are needed in the subsequent TestCase execution.

`TestConductor.TestContext.TestContextExecution_PostTestCaseOperation`

If this property contains a name of an operation of the TestContext, for each TestCase during execution of the TestContext, **after** a TestCase is executed the operation specified in this property is called automatically. In the operation specified in this property, one can reset some variables that are needed in the subsequent TestCase execution.

## Generating Test Reports with Rhapsody ReporterPLUS

Rhapsody ReporterPLUS is a reporting engine. The user is able to customize the content and style of a Rhapsody ReporterPLUS report by specifying a template. Rhapsody TestConductor delivers the test report template (`TestReport.tpl`) and the test requirement coverage report template (`TestRequirementCoverage.tpl`), which will be installed in the folder “reporterplus\Template” in your Rhapsody installation.

### Executing the ReporterPLUS with the Test Report Template

To execute the test report template on the model containing test data:

- For creating a report only for a selected TestPackage and the containing TestPackages, select a TestPackage in the Rhapsody browser and choose from the menu **Tools > ReporterPLUS > Report on selected package...**
- For creating a report for all TestPackages in the model choose from the menu **Tools > ReporterPLUS > Report on all model elements...**
- In the Rhapsody ReporterPLUS wizard **Select Task** specify the export file format your report shall be displayed in and click **Next>**.
- In the Rhapsody ReporterPLUS wizard **Select Template** check the currently active template. In case the template “`TestReport.tpl`” is not active click on “”, open it from the folder “reporterplus\Templates” in your Rhapsody installation folder and click **Next>**.
- The Rhapsody ReporterPLUS wizard **Confirmation** gives an overview about the selected options. Click the button **<Back** to change the options. Click **Generate** to start the execution of the Rhapsody ReporterPLUS report generation.

- In the dialog **Generate Document** specify a path and a name for the document to generate and click the button **Generate**.
- Rhapsody ReporterPLUS will show the progress during creating the document and start the corresponding application to show the test report.

## Using the HTML Test Report

The created HTML test report is divided into two sections, the table of Contents on the left side and the content section on right side. Dependent of the selected item on the left side, the corresponding section of the report will be shown on the right side.

**Note:** The HTML report will only be displayed correct in the internet browsers and versions, which are shown at report startup.

**Note:** The table of contents will only be shown in a HTML report. To display the table of contents Java must be installed. In case these requirements are not fulfilled, please select another export file format like Microsoft Word.

The first page gives an overview about the loaded model and the contained text contexts. This page is reachable from the highest entry of the table of contents.

Conceptual this report lists all TestContexts of the specified TestPackage(s) during creation. For each TestContext you will find information about

- the system under test
- the TestComponentInstances
- the TestContext diagrams
- the TestCases and their execution status

Each TestContext and the sub-items are reachable by clicking on the corresponding item in the table of content. Click on the plus to extend the tree structure.

## Using the Test Requirement Coverage Report

Execute the test requirement coverage template (`TestRequirementCoverage.tpl`) to get a statement about the relation between a requirement and the corresponding TestCases, which cover a requirement in the model. The testing profile defines the stereotype `<<TestObjective>>` which shall be used to setup a relation between a TestCase and a requirement, which it covers. In general a test objective is a stereotyped dependency, which can link on every element in the model.

This requirement coverage report focus especially on the dependency between a requirement and a TestCase. The test requirement coverage report gives another view on the model.

In opposite to the view “**All Requirements**”, the report also shows a table with “**All TestCases**” of the model. The “**All TestCases**” view is assistant to check, whether a TestCase has a test objective.

Some items in HTML report e.g. requirements, TestCases test results etc. are linked, so the user can easily browse to more detailed information pages.

## Customizing the Test Report

The test report template is customizable to fit specific users requirements. Follow the Rhapsody ReporterPLUS documentation how to adapt it to your needs.

## Generating Test Reports with Rational Publishing Engine

(see also TestConductor Tutorials:

- TestConductor\_Tutorial\_C.pdf
- TestConductor\_Tutorial\_Cpp.pdf

)

Rational Publishing Engine (RPE) is a tool that can be used to automate the generation of documents. The user is able to customize the content and style of a RPE report by specifying a template. Rhapsody TestConductor currently delivers a test requirement coverage report template (`TestRequirementCoverage.dta`), which will be installed in the folder “Share\RPE\Templates\TestConductor” in your Rhapsody installation.

## Creating the Test Report

- Choose from the menu **Tools > Rational Publishing Engine > Generate report...**
- Select the RPE template which should be used for report generation. The template “`TestRequirementCoverage.dta`” must be selected to create a requirement coverage report.
- Specify which types of output files should be created and where they should be saved.
- Then RPE automatically creates the selected reports.

## Test Requirement Coverage Report

A test requirement coverage report gives an overview about the requirements and TestCases specified in the model and how the requirements are covered by TestCases.

The testing profile defines the stereotype <<TestObjective>> which shall be used to setup a relation between a TestCase and a requirement.

All requirements specified in the model are listed and it is shown which requirement is covered by which TestCase. Detailed information are also available for each requirement.

The TestCases specified in the model are listed, too. Again detailed information are available for each TestCase.



## Creating Report Templates

How report templates can be created using Rational Publishing Engine Document Studio is described in the RPE documentation. An XML schema file of the testing profile (`testingprofile.xsd`) which can be used for template creation can be found in the folder “Share\RRE\Schemas” of your Rhapsody installation.

## Using the TestConductor API

(see also TestingCookbook:

- “How can I automate test execution using the TestConductor API?”

)

Similar to Rhapsody, TestConductor provides an API that can be used to access TestConductor functionality from

- Programs using the Rhapsody COM API
- Programs using the Rhapsody Java API

In order to use the TestConductor API the Rhapsody API function “`IRPApplication::runHelper(String)`” must be used. In order to apply this function correctly, one has to provide as an argument a valid TestConductor command. Additionally, before the “runHelper” function can be executed, an appropriate model element (e.g. a TestCase) must be selected by using the Rhapsody API.

The sample “CppSamples/TestConductor/TestingCookbook/CppTestAutomationSample” shows how to use the Java API in order to automate your testing work flows..

## Available TestConductor API Commands

The following TestConductor API commands are available and can be called by using the “runHelper” Rhapsody API function:

### Applicable to TestCase elements:

- “Edit TestCase SDInstances”
- “Update TestCase”
- “Build TestCase”
- “Execute TestCase”
  - Performs asynchronous TestCase execution, i.e., the function returns immediately and the execution of the TestCase is performed in a separate thread. The API script has to ensure itself (e.g. by waiting a specified amount of time) that the TestCase execution has finished before additional TestConductor API commands can be executed.

- “Execute TestCase Sync”
  - Performs synchronous TestCase execution, i.e., the function returns only after the execution of the TestCase has finished. This ensures that subsequent TestConductor API commands are only performed after the TestCase execution has finished. This is the preferred way of executing TestCases via the TestConductor API.

#### **Applicable to TestContext elements**

- “Create SD TestCase”
- “Create Flowchart TestCase”
- “Create Code TestCase”
- “Update TestContext”
- “Build TestContext”
- “Execute TestContext”
  - Performs asynchronous TestContext execution, i.e., the function returns immediately and the execution of the TestContext is performed in a separate thread. The API script has to ensure itself (e.g. by waiting a specified amount of time) that the TestContext execution has finished before additional TestConductor API commands can be executed.
- “Execute TestContext Sync”
  - Performs synchronous TestContext execution, i.e., the function returns only after the execution of the TestContext has finished. This ensures that subsequent TestConductor API commands are only performed after the TestContext execution has finished. This is the preferred way of executing TestContexts via the TestConductor API.
- “Execute TestPackage”
- “Update TestArchitecture”

#### **Applicable to TestPackage elements**

- “Create TestContext”
- “Update TestPackage”
- “Clean TestPackage”
- “Build TestPackage”
- “Execute TestPackage”
  - Performs asynchronous TestPackage execution, i.e., the function returns immediately and the execution of the TestPackage is performed in a separate thread. The API script has to ensure itself (e.g. by waiting a specified amount of time) that the TestPackage

execution has finished before additional TestConductor API commands can be executed.

- “Execute TestPackage Sync”
  - Performs synchronous TestPackage execution, i.e., the function returns only after the execution of the TestPackage has finished. This ensures that subsequent TestConductor API commands are only performed after the TestContext execution has finished. This is the preferred way of executing TestPackages via the TestConductor API.

#### **Applicable to Class elements**

- “Create TestArchitecture”

## **Defining Callbacks for TestConductor functions**

In addition to using the TestConductor API directly, one can also execute automated scripts after certain Rhapsody actions like e.g. 'After Add Element' or 'Before Code Generation'. For instance, to specify that 'After Add Element's certain helper should be activated automatically, one has to do the following steps:

- Define a helper with the Helper Trigger “After Add Element”. The helper can be implemented e.g. using a Java plug in or by an external program that uses the Rhapsody API.
- Now, whenever an element has been added the specified helper is invoked automatically. Hence, the helper itself has to decide whether certain things have to be done or the helper shall return without performing anything.

If, for example, the helper is designed to perform some action after creating a new TestArchitecture, then the helper has to decide whether the triggering action was TestArchitecture creation or any other model element creation.

Helpers with helper trigger “After Add Element” are invoked automatically for all actions that create new elements, like e.g. “Create Code TestCase”, “Create TestArchitecture”, e.t.c. pp. (cf. Rhapsody Extensibility Samples and IBM Knowledge Center “Creating Helpers”).

# Specifying Requirements with Sequence Diagrams

---

*Sequence diagrams* play a dominant role in the TestConductor test process. They are a key means for the graphical specification of tests, and enable TestConductor to visualize design flaws.

## Supported Diagram Elements in TestScenarios

- **Life-Line:**

A life-line represents an object, i.e.

- an instance of a class, block or actor or
- an implicit object or
- a file or
- the environment/system border, i.e. objects not explicitly represented in the TestScenario.

An object represented by a life-line is determined by their name and their realization. The realization refers to a classifier in the model (cf. mapping for replacements on page and dependencies for navigation on page).

The name refers to a unique access path to the object. It is a path -separated by dots- from a global object to the represented instance according to the instantiation hierarchy in the TestArchitecture. If necessary w.r.t. multiplicities of objects along the instantiation path, indices can be used in the access path.

Life-line names of are navigation expressions referring to instances of the respective classifiers in the TestArchitecture. TestConductor makes use of these navigation expressions e.g. for identifying suitable links when generating driver operations.

- asynchronous Message – **Event:**

- Stereotype <<RTC\_MsgInfo>> on Messages

- synchronous Message

- **Operation:**
- **Triggered Operation:**
- Stereotype <<RTC\_MsgInfo>> on Messages

- **dataflow Message:**
  - Stereotype <<RTC\_MsgInfo>> on Messages
- **Condition:**
  - <check> Condition. Deprecated (cf. section Using <check> Conditions / TestConditionpage 127). Replaced by TestCondition.
  - <precond> Condition. SysML HarmonySE only and deprecated ((cf. page 127). Replaced by TestAssignment
- **TestAction:**
  - general TestAction: see. section (general) TestActions, TestAssignments and TestConditions page 125.
  - Message-related TestAction: see section Influencing DriverOperation and Stub generation using TestActions in TestScenarios on page 122.
    - <InitAction>
    - <PreCallAction>
    - <CallAction>
    - <PostCallAction>
    - <StubAction>
- **TestCondition:** see pages 127 and 125.
- **TestAssignment:** see page 125.
- **Time-Interval** on TestComponent- or TestContext life-line
- **InteractionOccurrence:** see page 117.
- **InteractionOperator:** (see section Using Interaction Operators in SD TestCases on page 129)
  - `opt` : optional/conditional sub-scenario.
  - `alt` : alternative sub-scenarios, depending on conditions.
  - `consider` : consideration of only dedicated occurrence of a message. Other not specified occurrences of the message are ignored instead of interpreting them as unexpected occurrences.
  - `parallel` : sub-sequences are considered parallel or interleaving, respectively. Only order within the particular sub-sequence is relevant, parallel sub sequences aren't ordered w.r.t. each other.
  - `loop` : iterated sub-sequence. Loop depends on a condition like a while-loop.
  - `break` : operator to leave a sub-sequence conditional.
  - Stereotype <<RTC\_OperatorInfo>> on InteractionOperators (see section Using Interaction Operators in SD TestCases on page 129)

## Limitations of design elements (sequence diagrams)

Currently, TestConductor does not support the following sequence diagram features:

- Create arrow
- Destroy arrow
- Reply message
- Timeout
- Canceled timeouts
- Constraints
- Language for condition marks

Condition marks must obey the same syntax as activation conditions. Currently, simple expressions with equality or inequality are not yet allowed in activation conditions and condition marks.

**Note:** TestConductor will ignore condition marks during test execution.

If you use these unsupported features in a sequence diagram, TestConductor ignores them during test execution.

## Message Realization

(See also TestConductor Tutorial for Rhapsody in C and Rhapsody in C++.)

For specification of a TestScenario, messages can be drawn among the life-lines. Messages have to be *realized* for being considered by TestConductor. Realization of a message means formally establishing a reference to an interface item of the receiving life line in a sequence diagram. Messages can be realized either using the context menu item 'Select Message' or by opening the features dialog and selecting the realization on the general tab of the features dialog. On 'Update TestCase/TestContext/TestPackage' TestConductor treats unspecified realizations of messages as specification flaws and issues a warning for each message which has not been realized.

In witness TestScenarios messages with unspecified realization appear blue, since TestConductor does not regard such messages in execution.

## Ignoring Unrealized Messages

Messages with stereotype <<Unrealized>> are filtered out and ignored in the test execution.

On 'Update TestCase/TestContext/TestPackage' TestConductor treats <<Unrealized>> messages as intentionally unrealized and ignores such messages without issuing a warning for such messages.

In contrast to explicitly *unrealizing* a message, leaving a message unspecified (or selecting <Unspecified> as realization in the features dialog) is treated as '*not intended*' and leads to a warning when updating the TestCase/TestContext/TestPackage.

## Virtual Call vs Nonvirtual Call (Rhapsody in C++)

If the receiving life line of a message represents a class that virtually inherits from other classes, then the 'Select Message' context menu item offers different possible realizations for the implementations along the inheritance hierarchy.

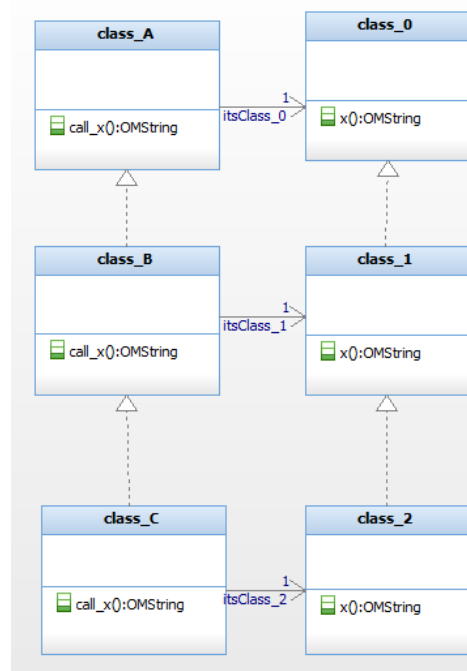


Figure 22: Example relations with inheritance

In a TestScenario specifying communication among **class\_C** and **class\_2**, 'Select Message' for a message to a life line representing **class\_C** offers `call_x()`, `class_B::call_x()`, `class_A::call_x()` (among others):

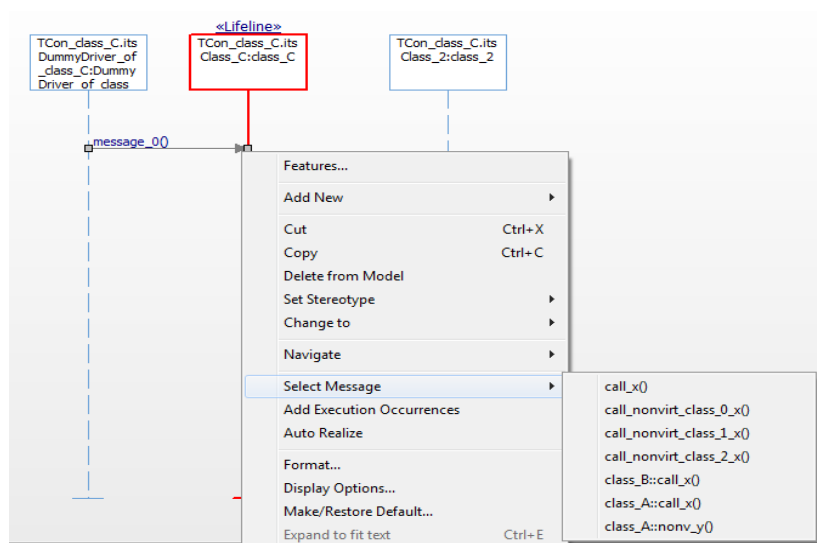


Figure 23: Select Message with virtually inherited operations

And accordingly, for a message to a life line representing `class_2`:

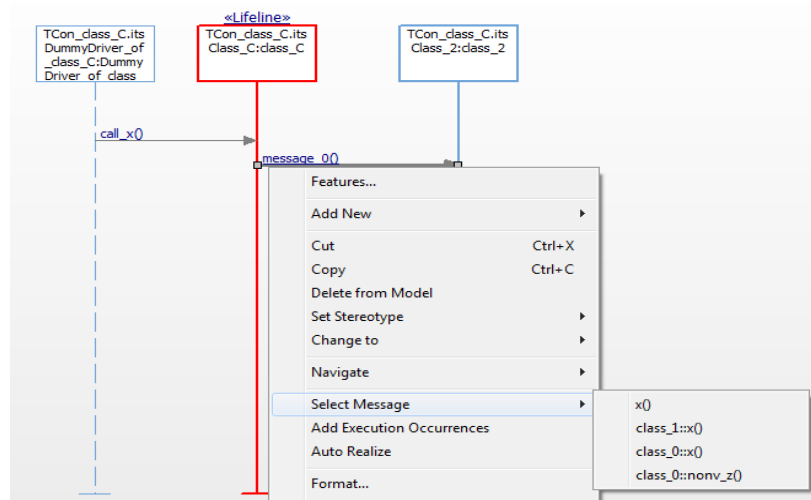


Figure 24: Select Message and Inheritance

By default, TestConductor treats the possible realizations different dependent on the message context:

- if a DriverOperation will be generated from the message - as it is the case in figure 23, where the message is send by a TestComponent life line and received by a SUT life line – selection of base implementation `class_A::call_x()` will yield a non-virtual call in the DriverOperation, whereas selection of `call_x()` will yield a virtual call in the DriverOperation. This behavior can be overridden by stereotyping the message `<<call_virtual>>`. This will yield a virtual call, regardless of selecting `call_x()` or `class_A::call_x()` as realization.

- if a StubOperation will be generated from the message – as it is the case in figure 24, where the message is send by a SUT life line and received by a TestComponent life line – TestConductor will by default always generate a StubOperation in the most specialized TestComponent (as if `<<call_virtual>>` was set).

This behavior can be overridden by stereotyping the message `<<call_nonvirtual>>`. This will generate the StubOperation in the TestComponent for `class_2` if `x()` is selected as realization and will attempt to generate the StubOperation in the TestComponent for `class_0`, if `class_0::x()` is selected as realization.

Note that by default TestArchitecture creation will only create a replacement TestComponent for `class_0` if `class_0` has non-virtual operations. Hence, a non-virtual call of a StubOperation in `class_0` is only possible if `x()` can be stubbed in `class_0` – which requires a replacement TestComponent for `class_0`.

If there is no replacement TestComponent for `class_0`, an appropriate warning will be issued during 'Update TestCase/TestContext/TestPackage'. In this case the TestArchitecture has to be adapted manually to the needs of the TestCase.



## Self-Messages in BlackBox and GreyBox Testing

(cf. TestingCookbook:

- “How can I observe the communication between parts of a greybox SUT?”

)

In BlackBox testing self-messages of SUT life lines are not observable, since the SUT can not be instrumented for observation (cf. section Black Box Testing on page 39). Therefore self-messages are ignored by TestConductor and appear blue in witness TestScenarios.

In GreyBox testing (cf. section Grey Box Testing on page 40 and GreyBox TestArchitectures for classes and objects on page 35) self-messages of SUT life lines are observable in principle, since the TestArchitecture instantiates copies of the SUT elements as scope replacements (cf. section Replacements on page 24). These replacements can be instrumented with assertions for observation purposes without affecting the original model elements.

This way operations invoked by a SUT element on itself or events sent to itself become feasible in GreyBox specifications. Note, that this only regards member operations of the SUT element itself and events received by its own statechart or activity diagram, but does not automatically involve also messages to parts of the SUT or among parts of the SUT.

## SelfMessageRealizationInParts

(cf. Samples:

- TestCase 'SD\_tc\_gb\_observation\_record' in  
TPkg\_Coffeemachine\_GB::TCon\_Coffeemachine\_Architecture::TCon\_Coffeemachine  
  - of Sample  
Samples/CppSamples/TestConductor/TestingCookbook/CppCompositeCoffeMachine\_RAL
  - or in  
Samples/CppSamples/TestConductor/TestingCookbook/CppCompositeCoffeMachine\_wo\_ports

)

TestConductor automatically introduces replacements for one decomposition level when creating a GreyBox TestArchitecture.

This enables support for TestScenario specifications with life lines for the direct parts of the SUT for which the TestArchitecture has been created. But often it is more desirable to view the parts of a class or objects as belonging to the same life line as the instantiating class or object (as in the animation feature enabled in animated sequence diagrams by Animation.ClassifierRole.MappingPolicy=ObjectAndItsParts).

TestConductor supports this treatment of a life line as representing a composite class or object and its parts also in TestScenarios: for self-messages of such a life-line the realization has to be either chosen as member of the composite class or object itself or as member of one of its parts. To enable offering of message realizations in parts in the 'Select Message' context menu on messages and in the features dialog of messages, stereotype <<SelfMessageRealizationInParts>> has to be set on the respective TestScenario.

## Using Time Interval for Delay Driving from TestContext and TestComponents

(see also TestingCookbook:

- “How can I specify upper/lower time bounds for responses of the SUT?”

)

TestConductor provides capabilities to automatically drive messages (events, operations or triggered operations) with a certain delay. Users can specify that TestConductor should drive messages from a TestComponent or TestContext to the SUT with a certain time delay. Whenever a message must be driven, users can specify that TestConductor waits for a certain amount of time (`ms`, `sec`, `min`) in order to delay actual message generation. This is expressed on the sending life-line (either the system border or a TestComponent) with the time interval notation of the sequence diagram editor.

**Note:** TestConductor will regard only time intervals between messages, if driving messages are defined from a TestComponent or TestContext life-line and the time interval definition is specified also on a TestComponent or TestContext life-line. Any Time Interval on a SUT life-line will be ignored.

Time delays will be specified with the time interval notation in sequence diagrams. TestConductor supports time intervals only if they are drawn on TestComponent life-lines. The label of a time interval specifies the time unit (`ms`, `msec`, `sec`, `min`) and a time value.

**Syntax:** `relop value unit`

`relop := <= | < | > | >=`

`value := integer`

`unit := ms | msec | sec | min`

Time Intervals can be used to specify upper and lower temporal bounds for two successive observations or events, i.e. the TestScenario element above the start of the Time Interval and the TestScenario element below the end of the Time Interval. For example, `> 3 msec` specifies that *at least* 3 milliseconds have to pass after the first observation before observing/driving the second one, whereas `< 3 msec` specifies that *at most* 3 milliseconds may pass between the two observations.

## Specifying Argument Values

(See also TestConductor Tutorial for Rhapsody in C and Rhapsody in C++.)

For messages referring to operations or events with arguments, argument-values have to be specified. For messages from TestComponents to SUT, TestConductor will generate appropriate DriverOperations based upon the specified argument values. For messages

from SUT to TestComponents, TestConductor will generate appropriate assertions for checking argument value adherence to the specification. By default<sup>22</sup>, TestConductor will display the actual argument value in the witness TestScenario (cf. page 133) for failed assertions.

The argument value specification has to be in 'named' form, i.e. for a message referring e.g. to `void op(int a, int b, bool c)`, `'a=42,b=17,c=true'` has to be entered in the 'Arguments'-field of the features dialog for the message, or directly in the message annotation displayed in the TestScenario. A 'positional' argument specification, as in `'42,17,true'` is not supported by TestConductor.

TestConductor will issue a warning on updating TestCase/TestContext/TestPackage for messages with unspecified argument values.

For individual argument values, also don't care '\*' can be used (cf. page 117). For specifying that some argument value has to be only in a certain range instead of specifying a particular value, range specifications can be used (cf. page 118). A dedicated syntax supports specification of input and output value for InOut arguments separately (cf. page 116).

## Specifying dataflows

(see also TestingCookbook:

- “How can I specify sequence diagram test cases for SUTs that use flow ports?”

)

dataflows can be used to specify value communication via flow ports or proxy ports.

In the feature dialog of the dataflow message, name of the dataflow has to be the name of the concerned data item on the receiver side and the flowport of the receiver has to be set in the flowport-entry. Otherwise the dataflow is treated unrealized.

On updating TestCase/TestContext/TestPackage, TestConductor generates driver code for dataflows from TestComponents to SUT and generates appropriate assertions for checking the communicated data for dataflows from SUT to TestComponents.

## Specifying Return Values

(examples can be found in many of the sample models, e.g.:

- Samples/CppSamples/TestConductor/TestingCookbook/CarRadio

see also TestingCookbook:

- “How can I specify checks on return- or output values of function calls in a sequence diagram test case?”

)

Users can specify expected return values and output values for operation calls. To specify a return value for an operation, open features dialog of an operation in a sequence diagram. Specify the expected return value in the **Return Value** field.

---

<sup>22</sup>TestingConfiguration tag `rtc_assert_handling` has to be set to `by_string`

Depending on the direction of the message, TestConductor will generate a check of the return value (message from TestComponent to SUT) or provide a stub returning the specified return value (message from SUT to TestComponent).

**Note:** Serialization/deserialization functions will be used in comparison of values of user defined types, if available/defined (cf. Sample model: Samples/CppSamples/TestConductor/TestingCookbook/CppListUsage).

For specifying return values, also don't care '\*' can be used (cf. page 117). For specifying that a return value has to be in a certain range instead of specifying a particular value, range specifications can be used (cf. page 118).

## Specification of Out and InOut Argument Values

(see also TestingCookbook:

- “How can I specify checks on return- or output values of function calls in a sequence diagram test case?”

)

Besides specifying the type of an operation argument, also the direction of the argument can be defined in the features dialog. Operation arguments can be In, Out or InOut. Properties<sup>23</sup> {Lang\_CG}.Type.In, {Lang\_CG}.Type.InOut and {Lang\_CG}.Type.Out determine the real type of the operation argument in the generated code. TestConductor supports not only the resulting code type of the operation argument, but also the model based view of argument direction.

As example, we consider an operation `m(int p1, int p2, int p3, int p4)`, where

- p1 and p2 are In arguments and
- p3 is an Out argument, and
- p4 is an InOut argument.

In a sequence diagram users can specify the expected In argument values and the expected Out and InOut parameter values.

When '`m(p1=1, p2=2, p3=42, p4=17)`' is specified in a TestScenario, this specifies that `m()` is invoked with arguments

- p1=1 and p2=2 as *input* values,
- p3 with *any input* value – since p3 is an Out argument the input value does not play a role for the call, but it will be checked or assured<sup>24</sup> in the stub that p3 equals 42 after the call - and
- p4=17 as *input* value. The effect of the call on p4 is not specified.

Since p4 is an InOut argument, it is necessary to be able to specify the input and the output value of p4 separately. Therefore, the syntax for specifying an InOut argument is

<sup>23</sup>where {Lang\_CG} is either C\_CG or CPP\_CG, depending on the language setting of the model.

<sup>24</sup>depending on whether the message is drawn from a TestComponent to SUT or the other way round.

```
<parameter> = In:<in_value>;Out:<out_value>
```

Thus, "m(p1 = 3, p2 = 5, p3 = 7, p4 =In:9;Out:12)" specifies that m() is called with input "p1=3", input "p2=5", input "p4=9". Message m() returns with output "p3=7", and output "p4=12". TestConductor checks or assures the output values according to the specification of the message w.r.t. the direction of the message<sup>25</sup>

**Note:** Serialization/deserialization functions will be used in comparison of values and for driving values of user defined types, if available/defined (cf. Sample model: Samples/CppSamples/TestConductor/TestingCookbook/CppListUsage)

## Interaction Occurrence – Reference Sequence Diagram

(see also Sample-model CSamples/TestConductor/CSDOperators )

Specification of message communications in a system often requires long sequences with lots of messages. TestScenarios can thus grow large and unintuitive. For structuring TestScenarios and for sharing sub-scenarios among different TestScenarios, Interaction Occurrences can be used. An Interaction Occurrence is a reference *at a certain position* in one TestScenario to another separate TestScenario as if the referenced TestScenario would be substituted in the referencing position.

For TestConductor, it is logically the same if users specify a scenario within one sequence diagram or if the scenario is specified with Interaction Occurrences. Whenever an Interaction Occurrence is reached, then the referenced TestScenario as specified in the Interaction Occurrence is tested. Test control starts with the main TestScenario, and when an Interaction Occurrence is reached, the control goes into the referenced TestScenario, and as the execution of the referenced TestScenario is completed, the control returns back into the main TestScenario.

TestConductor does not care if:

- referenced TestScenario *does not contain the same life lines* as surrounded by the interaction occurrence
- referenced TestScenario contains *fewer* life lines
- referenced TestScenario contains *more* life lines
- referenced TestScenario contains *other* life lines

TestConductor just considers the provided life lines and the specified messages as relevant TestScenario and expects exactly those messages when the SUT is executed.

For failure analysis for TestScenarios using Interaction Occurrences see section Failure Analysis for InteractionOccurrences on page 134.

## Don't care values

In some cases one might not be interested in checking actual parameter values. If

---

<sup>25</sup>a check is performed if the message originates in a testComponent and is invoked on the SUT, otherwise outvalues are provided by the TestComponent's stub.

- Message arguments have values that change whenever the application executes (sensor values, etc.). TestConductor should not compare the actual values with the specified values.
- Message argument is a pointer to e.g. a structure. TestConductor does not automatically compare the actual values in the structures and it doesn't make sense in general to compare two pointers to structures.

For such cases, message arguments or returns can explicitly be specified as don't care by using a star '\*' as message argument value or for the return value of an operation.

Specifying an argument as don't care means, that the argument will not be driven with a certain value in a message from TestComponent to SUT or will not be checked for a certain value in a message from SUT to TestComponent.

Specifying a return as don't care means, that the argument will not be checked for a certain value in a message from TestComponent to SUT or will not be stubbed with a certain return value in a message from SUT to TestComponent.

## Range Specification

Range specifications allows monitoring and checking whether argument values of messages are in a given specified range. Checking ranges is required if messages have arguments that vary literally from run to run. Body temperature may be a good example for such message argument since it is unlikely that the values are always the same. Usually temperature is in a certain range, e.g. between 36.5 and 36.9 degree Celsius for humans. Do, there is a healthy range for body temperature and body temperatures out of this range have to treated unhealthy.

- A special notation can be used to indicate ranges instead of specific values.  
Notation:

```
[<lower_value> .. <upper_value>]
```

where *lower\_value* and *upper\_value* have to values of a scalar type like integer, long, double etc.

Example: for a message *m* with arguments *int p1*, *char\* p2* and *float p3*, it can be specified "*m*(*p1*=1, *p2*\*, *p3*=[1.5 .. 1.7])" to state that *p1* must equal '1', *p2* is "don't care", *p3* must be in the range between '1.5' and '1.7'.

Ranges can be used for message arguments as well as for return values. TestConductor will treat ranges differently depending on they appear in messages for which DriverOperations or Stubs will be generated:

- for a driven message<sup>26</sup>, i.e. e.g. *msg* (*x*=[0 .. 4]) TestConductor will choose the lower\_value of the range for driving the message.
- for a driven message, i.e. e.g. [0 .. 4]=*msg2* () TestConductor will check whether the returned value is in the specified range.
- for a stubbed message<sup>27</sup>, i.e. e.g. *msg3* (*x*=[0 .. 4]) TestConductor will check whether argument *x* of the message is in the specified range.

<sup>26</sup>Message from a TestComponent to SUT with in-argument *x*.

<sup>27</sup>Message from SUT to a TestComponent with in-argument *x*.

- for a stubbed message, i.e. e.g. [0..4]=msg4() TestConductor will choose the lower\_value of the specified range as return value of the stub.

## Influencing DriverOperation and StubOperation Generation

For automatic generation of DriverOperations and StubOperations see also section Model Population – Create Driver Operations and StubOperations on page 48.

Interpretation of messages by TestConductor depends on their source and target life-lines.

The following table gives an overview about the interpretation of messages by TestConductor:

from	to	TestComponent	SUT										
TestComponent		<p>by default <b>not driven</b>, but monitored on receiver side, arguments checked, return value not checked.</p> <p>relevant tags in &lt;&lt;RTC_MsgInfo&gt;&gt;: RTC_DriveTestComponentMessage</p>	<p><b>driven</b>, return value checked if specified.</p> <p>relevant tags in &lt;&lt;RTC_MsgInfo&gt;&gt;:</p> <table><tr><td>tag</td><td>TestAction</td></tr><tr><td>RTC_DriverInitCode</td><td>InitAction</td></tr><tr><td>RTC_DriverInitCodeAdditional</td><td>PreCallAction</td></tr><tr><td>RTC_DriverCallCode</td><td>CallAction</td></tr><tr><td>RTC_DriverCallCodeAdditional</td><td>PostCallAction</td></tr></table> <p>RTC_Monitor (greybox-testing only!)</p>	tag	TestAction	RTC_DriverInitCode	InitAction	RTC_DriverInitCodeAdditional	PreCallAction	RTC_DriverCallCode	CallAction	RTC_DriverCallCodeAdditional	PostCallAction
	tag	TestAction											
RTC_DriverInitCode	InitAction												
RTC_DriverInitCodeAdditional	PreCallAction												
RTC_DriverCallCode	CallAction												
RTC_DriverCallCodeAdditional	PostCallAction												
SUT		<p><b>observed, arguments checked, return value provided</b> by stub if specified.</p> <p>relevant tags in &lt;&lt;RTC_MsgInfo&gt;&gt;:</p> <table><tr><td>tag</td><td>TestAction</td></tr><tr><td>RTC_StubBodyCode</td><td>StubAction</td></tr></table> <p>RTC_Monitor (greybox-testing only!)</p>	tag	TestAction	RTC_StubBodyCode	StubAction	<p><b>not observable in blackbox testing</b></p> <p>in <b>greybox testing: arguments checked</b>, return not checked in pre 8.1.5, although specified.</p>						
tag	TestAction												
RTC_StubBodyCode	StubAction												

Figure 25: Interpretation of Messages by TestConductor

Recall that tag RTC\_Monitor of stereotype <<RTC\_MsgInfo>> and of stereotype <<RTC\_InstInfo>> can be used to influence the standard rules for message interpretation (cf. section Model Population – Create Driver Operations and StubOperations on page 48).

## User Defined DriverOperations

The default implementation of a driver operation generated by TestConductor may be overwritten and customized by the user. To influence the automatic generation of DriverOperations from messages in a TestScenario with user-defined code, there are two alternative techniques available in TestConductor:

- using tags of the <<RTC\_MsgInfo>> stereotype on messages

- using <InitAction>, <PreCallAction>, <CallAction> and <PostCallAction> TestActions referring to the message in the TestScenario.

## User Defined StubOperations

The default implementation of a stub operation generated by TestConductor may be overwritten and customized by the user. To influence the automatic generation of StubOperations from messages in a TestScenario with user-defined code, there are two alternative techniques available in TestConductor:

- using tags of the <<RTC\_MsgInfo>> stereotype on messages
- using <StubAction> TestActions referring to the message in the TestScenario.

## Influencing DriverOperation and Stub generation using <<RTC\_MsgInfo>> tags

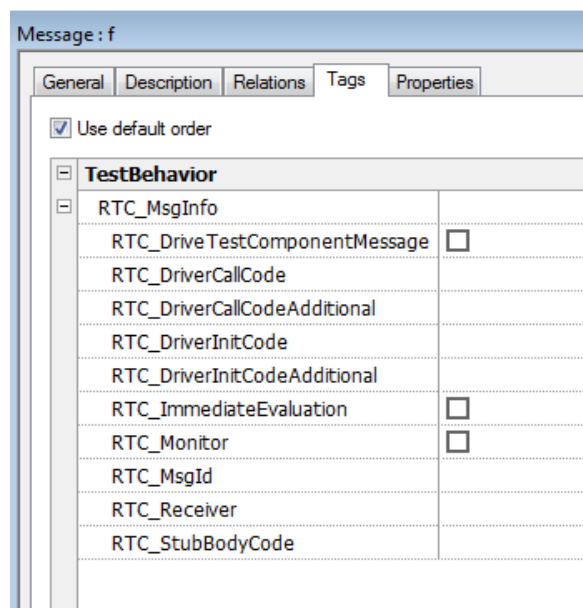


Figure 26: Tags of Stereotype <<RTC\_MsgInfo>>

Usually, if the user modifies driver or stub operations in the model, these modifications are lost if the user updates a TestCase. The user can influence the way how TestConductor automatically generates code for driver operations and StubOperations. Using the tags  
TestBehavior::RTC\_MsgInfo::RTC\_DriverCallCode,  
TestBehavior::RTC\_MsgInfo::RTC\_DriverCallCodeAdditional,  
TestBehavior::RTC\_MsgInfo::RTC\_DriverInitCode,  
TestBehavior::RTC\_MsgInfo::RTC\_DriverInitCodeAdditional,  
TestBehavior::RTC\_MsgInfo::RTC\_StubBodyCode



The contents of these tags will not get lost during update of a TestCase.

The value for `RTC_DriverInitCode` is taken as the beginning of the driver operation body containing the initialization of necessary variables, whereas the value for `RTC_DriverCallCode` is taken as the trailing part of the driver operation body containing the call of the function to be driven.

Note that both properties can be overwritten separately by the user. In case the user wants to customize the initialization section only, only the property `RTC_DriverInitCode` has to be overwritten; TestConductor will continue to automatically generate the code for the driver call section (and vice versa).

The value for `RTC_DriverInitCodeAdditional` is taken as additional initialization code that is generated in addition to the initialization code generated by TestConductor. The content of this tag is generated directly after the auto generated initialization code. Similarly, the value for `RTC_DriverCallCodeAdditional` is taken as additional call code that is generated in addition to the auto generated call code. The content of this tag is generated directly after the auto generated call code.

## RTC\_DriverInitCode and RTC\_DriverInitCodeAdditional

The user can influence the initialization of arguments before the message is driven using the tags `RTC_DriverInitCode` and `RTC_DriverInitCodeAdditional`. To do this uses have to add the stereotype `RTC_MsgInfo` to the SD message. This adds automatically the tags `RTC_DriverInitCode` and `RTC_DriverInitCodeAdditional` to the message. The user can fill these tags with code which will be used as initialization code of the driver operation when the TestCase is updated. Important is that the context of `RTC_DriverInitCode` completely replaces the initialization code that would be generated by TestConductor automatically, whereas the content of `RTC_DriverInitCodeAdditional` is simply added to the auto generated initialization code.

In some cases it is advisable that the user copies all or the needed parts of the automatically generated *driver initialization code* section and paste it into the tag `RTC_DriverInitCode` before starting to implement his own changes.

## RTC\_DriverCallCode and RTC\_DriverCallCodeAdditional

The user can also influence the call of the driven operation using the tags `RTC_DriverCallCode` and `RTC_DriverCallCodeAdditional`. To do this he users have to add the stereotype `RTC_MsgInfo` to the sequence diagram message. This adds automatically the tags `RTC_DriverCallCode` and `RTC_DriverCallCodeAdditional` to the message. The user can fill these tags with code which will be executed after the initialization of arguments. Important is that the content of `RTC_DriverCallCode` completely replaces the code that would be used to invoke the driven operation if TestConductor generated the code automatically, whereas the content of `RTC_DriverCallCodeAdditional` is simply added to the auto generated call code.

Note, in this scenario the user has has the responsibility that the sequence diagram TestCase is indeed executable after customization. Basically, the specified message of the sequence diagram TestCase, which now is present as source code, has to be represented in the user defined code.

In some cases it is advisable that the user copies all or the needed parts of the automatically generated *driver call code* section and paste it into the tag `RTC_DriverCallCode` before starting to implement his own changes.

## RTC\_StubBodyCode

Normally, if the user modifies StubOperations in the model, then this information will be lost on updating TestCase/TestContext/TestPackage. The user can influence the code of the stub using the tag `RTC_StubBodyCode` (or using `<StubAction> TestAction`, cf. page 122). To do the respective message has to be stereotyped `<<RTC_MsgInfo>>` - the stereotype adds automatically the tag `RTC_StubBodyCode` to the message. The value of this tag will be used as body of the StubOperation when the TestCase is updated. The content of the tag completely replaces the body that would be generated by TestConductor automatically.

If an operation is stubbed multiple times in the same TestComponent in the same sequence diagram instance, then for each occurrence an individual StubOperation is generated.

## Deleting `<<RTC_MsgInfo>>` Tags (User Defined Driver and Stubs)

TestConductor regards the tags `RTC_DriverInitCode`, `RTC_DriverCallCode`, `RTC_DriverCallCodeAdditional`, `RTC_DriverInitCodeAdditional` and `RTC_StubBodyCode` of stereotype `<<RTC_MsgInfo>>` if the tags are overwritten. To delete the user defined operation call and use the auto generated operations from TestConductor, it is thus not enough to empty them but the tags have to be unoverridden. It is thus necessary to reset the tags to return to TestConductor's default behavior.

## Influencing DriverOperation and Stub generation using TestActions in TestScenarios

(see also sample model:

- `Samples/CppSamples/TestConductor/CppTestActions`

)

In the previous section, the tags of the `<<RTC_MsgInfo>>` stereotype have been used in order to customize the driver code and stub code generation of TestConductor. Alternatively, the same can be done in a more graphical fashion by using so-called TestActions. A TestAction is an action that can be placed on one of the TestComponent life lines in the sequence diagram. The TestAction contains code that is considered by TestConductor when the model is populated with test code, and it can be used to e.g.

- create (complex) input data
- access e.g. global variables of the TestArchitecture
- create (complex) checks for (complex) output values
- define (complex) behavior of stubs

In order to support the use cases mentioned above, besides a *general* TestActions, TestConductor provides a set of message related TestActions.

- a *general* TestAction is a container for a code-block of statements that is executed by TestConductor if test execution reaches the TestAction. In order to define a general TestAction, just add a TestAction block to one of the TestComponent life-lines in the TestScenario. In contrast to the message-related TestActions described below, a general TestAction is not related to another message in the TestScenario.

While general TestActions offer the possibility to execute arbitrary user defined code in a certain position in the TestScenario, i.e. in a certain instant of time during TestCase execution, *message related* TestActions only have a meaning in combination with the message to which they refer. They are used to deviate from the automatic DriverOperation and StubOperation generation for particular messages. TestConductor supports the following kinds of message related TestActions:

- **<InitAction>**: An init action is a TestAction that can be used to initialize test data. The code contained in the init action is handled as the tag `RTC_DriverInitCode` of the stereotype `<<RTC_MsgInfo>>` (cf. section “RTC\_DriverInitCode and RTC\_DriverInitCodeAdditional” on page 121 ).  
**<InitAction>** TestAction must be placed on sending TestComponent life-line *directly*<sup>28</sup> before message sending.
- **<PreCallAction>**: A pre call action is a TestAction that can be used to either initialize test data or to do some other test related activities before a message is sent from a TestComponent to a SUT instance. The code contained in the pre call action is handled as the tag `RTC_DriverInitCodeAdditional` of stereotype `<<RTC_MsgInfo>>` (cf. section “RTC\_DriverInitCode and RTC\_DriverInitCodeAdditional” on page 121).  
**<PreCallAction>** TestAction must be placed on sending TestComponent life-line *directly* before message sending.
- **<CallAction>**: A call action is a TestAction that can be used to call a particular operation or to send a particular event. The code contained in the call action is handled as the tag `RTC_DriverCallCode` of stereotype `<<RTC_MsgInfo>>` (cf. section “RTC\_DriverCallCode and RTC\_DriverCallCodeAdditional” on page 121).  
**<CallAction>** TestAction must be placed on sending TestComponent life-line *directly* before message sending.
- **<PostCallAction>**: A post call action is a TestAction that can be used to perform any kind of actions after a particular call to an operation or a sending of an event, e.g. code for checking output values of the called operation. The code contained in the call action is handled as the tag `RTC_DriverCallCodeAdditional` of stereotype `<<RTC_MsgInfo>>` (cf. section “RTC\_DriverCallCode and RTC\_DriverCallCodeAdditional” on page 121).  
**<PostCallAction>** TestAction must be placed on sending TestComponent life-line *directly* after message sending.
- **<StubAction>**: A stub action is a TestAction that can be used to define the behavior of stubbed operations, e.g. checking arguments of the called operation or returning specific values. The code contained in the stub action is handled as the

<sup>28</sup>there must be no other message or *general* TestAction between *message related* TestActions and the related message. If more than one of **<InitAction>**, **<PreCallAction>** and **<CallAction>** are used, they have to be used in this order.

tag "RTC\_StubBodyCode" of stereotype <<RTC\_MsgInfo>> (cf. section "RTC\_StubBodyCode" on page 122).

<StubAction> TestAction must be placed on receiving TestComponent life-line *directly* after message reception.

In order to add a TestAction to a sequence diagram TestCase:

- On the TestScenario toolbar, select the TestAction icon.
- Place the TestAction on the TestComponent life-line next to the affected message in the TestScenario according to the above positioning rules.
- Write <InitAction> ,<PreCallAction> , <CallAction> , <PostCallAction> or <StubAction> , respectively, into the first line of the TestAction according to the intended meaning of the TestAction.
- Write code to be performed in place of the message related TestAction into the TestAction beginning after the first line.

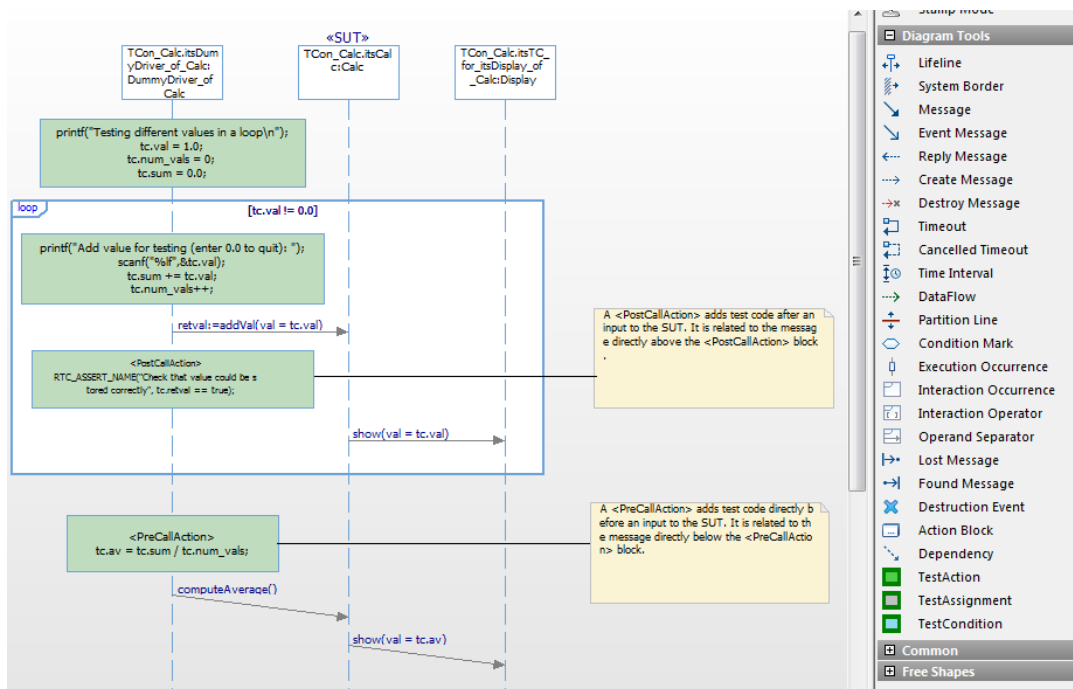


Figure 27: Using TestActions

After adding the TestActions to the TestScenario, the TestCase has to be updated. During the update, the TestActions are populated into the driver operations and StubOperations in the model. For instance, the <PostCallAction> in the TestScenario depicted above is populated to the driver operation for the message "addVal" that is specified directly above the <PostCallAction>:

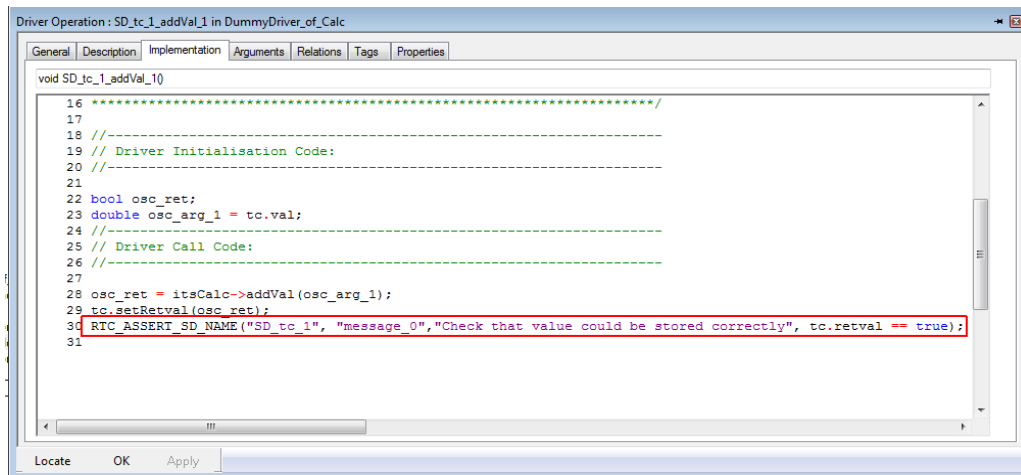


Figure 28: Assertion generated from <PostCallAction> TestAction

After building the TestCase, the TestCase can be executed. The code in the TestActions is executed when the TestCase reaches the specified TestActions. For instance, the assertion specified in the <PostCallAction> of the TestScenario depicted above is executed directly after operation “addVal” was called on the SUT.

Note: When doing “Show as SD”, *message related* TestActions of kind <InitAction>, <PreCallAction>, <CallAction>, <PostCallAction> or <StubAction> will always be colored blue, no matter if assertions in the TestAction have failed or passed, if the code has been executed or not. Instead, the corresponding message is colored according to the assertions. Only *general* TestAction will be colored in green, red or blue according to the result of assertions defined since *general* TestAction are not message related.

## Clean TestComponent

Driver and StubOperations can be deleted manually, but TestConductor provides the functionality to delete the automatically generated operations of a TestComponent at once. To clean a TestComponent select the TestComponent and invoke item **Clean TestComponent** from the context menu (cf. section Clean TestComponent on page 52).

## Clean TestPackage

Driver and StubOperations can be deleted manually, but TestConductor provides the functionality to delete the automatically generated operations of all TestComponents of a TestPackage at once. To clean a TestPackage select the TestPackage and invoke item **Clean TestPackage** from the context menu (cf. section Clean TestPackage on page 52).

## (general) TestActions, TestAssignments and TestConditions

(see also TestingCookbook:

- “How can I specify checks on return- or output values of function calls in a sequence diagram test case?”
- “How can I set or check attribute values in sequence diagram test cases?”

see also Sample-Model CppSamples/TestConductor/CppTestActions.

)

While general TestActions can be used on TestComponent and TestContext life-lines to execute arbitrary code statements in the TestCase – e.g. for assigning variables a value, for initializing relations, for invoking operations or sending events or for applying explicit checks using RTC\_ASSERT- macros (cf. pg 156) - TestActions can not be placed on SUT life-lines. On updating a TestCase, a TestContext or a TestPackage, TestConductor generates a driver-operation in the respective TestComponent or (or in the TestContext, respectively) to be invoked during test execution. Thus, the expressions in the TestAction have to be valid expressions in the scope of the respective TestComponent and have to be specified accordingly – requiring the user to provide appropriate program-code for the desired effects.

Hence, TestActions provide the user with a powerful and expressive tool to influence the model execution tailored to the specific TestScenario. On the other hand, programming skills are required to realize the desired effects of e.g. setting a SUT attribute's value, establishing dedicated value checks, e.t.c.

TestAssignments and TestConditions aim at easing two important use cases of applying TestActions to TestScenarios:

TestActions can be placed on TestComponent (and TestContext) instances only, but aren't restricted w.r.t. permitted expressions. In contrast, TestAssignments (applicable to TestComponent as well as SUT instances) and TestConditions (limited to SUT instances) are aimed at supporting value assignments to attributes of the respective SUT instance and offering support for simple value checks. Due to the dedicated use case of TestAssignments and TestConditions, the supported syntax of the assignments and check-conditions can be kept very simple and using TestAssignments and TestConditions hence doesn't require complicated navigation expression programming:

#### Syntax of TestAssignment:

```
assignments ::= assignment {assignment}*  
assignment ::= name = value  
name ::= identifier | partidentifier.identifier
```

value will not be interpreted by parser

#### Syntax of TestCondition:

```
check ::= name relop value  
        | check_op  
        | partidentifier.check_op  
check_op ::= IS_IN(stateidentifier)  
name ::= identifier | partidentifier.identifier  
relop ::= == | > | >= | <= | < | !=
```

value will not be interpreted by parser

## Preconditions (for SysML/HarmonySE)

For SysML/HarmonySE models, i.e for SysML models that contain the HarmonySE profile, TestConductor provides a special kind of condition, so-called preconditions. With preconditions, in SysML/HarmonySE models one can set attributes of SUTs to specified values. This is useful whenever the behavior of the SUT depends on values of local attributes. In order to define a precondition in a TestScenario, add a condition on the life line of the SUT instance that contains the attribute, write “<precond>” into the first line of the condition's text, and specify the value the attribute should have in the next line:

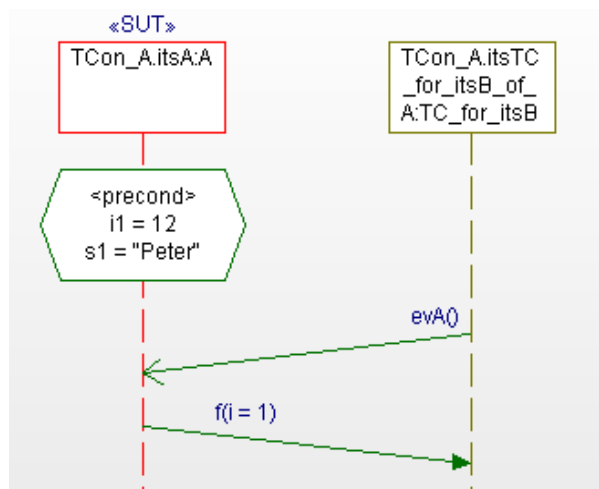


Figure 29: <precond> Condition (HarmonySE)

In the example depicted above, a precondition is specified that defines value “12” for the attribute “i1” and value “Peter” for attribute “s1” of block A. When executing the TestCase, and TestConductor reaches the precondition, it sets the specified values for the attributes. When the TestCase continues, now the behavior of the SUT reflects the new values for the attributes. The usage of preconditions is restricted to SysML/HarmonySE models. If multiple attributes should be set by a precondition, the attribute value specification must be separated by newlines in the condition mark.

<precond> conditions are deprecated. Instead of <precond> conditions, TestAssignments (cf. pages 125) should be used. TestAssignments are generally supported in assertion based testing mode and are not limited to SysML/HarmonySE, but can also be used in regular C and C++ models.

## Using <check> Conditions / TestCondition

For assertion based testing mode, TestConductor supports the use case of testing attribute

values of the SUT with a specialized variant of TestActions:

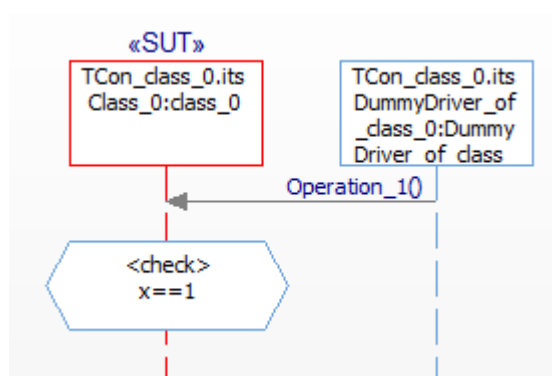


Figure 30: <check> Condition

A condition with '<check>' mark in its first line followed by expressions of the form 'attributename relop value' – each of the expressions in a single line without line feeds in the individual expression, can be used to check attributes of the SUT for particular valuations. TestConductor will check this condition in the scope of the SUT. The condition is evaluated according to the order of observations specified by the SD. In particular, TestConductor will not wait unless the condition becomes true. If the condition is not true at the moment of evaluating it, the TestCase will fail.

In order to use a '<check>' condition, choose “Condition Mark” in the drawing tool for sequence diagram, write “<check>” into the first line of the condition's text and add the conditions to be checked to the subsequent lines.

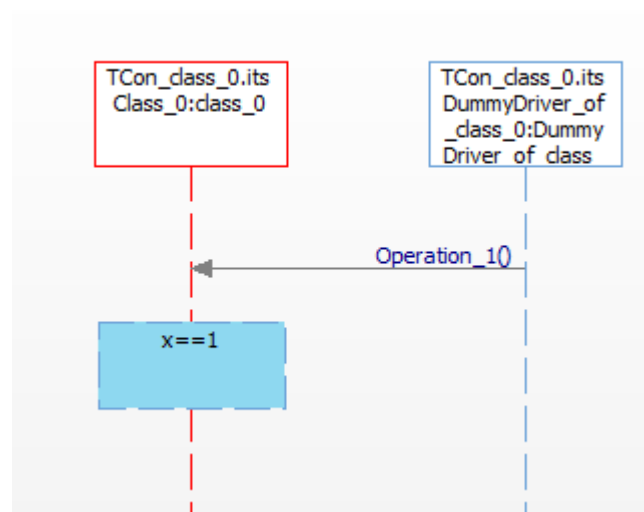


Figure 31: TestCondition

'<check>' conditions are deprecated – TestConditions as specialization of TestActions have been introduced for the same purpose. The same as '<check>' conditions, TestConditions aim at checking conditions in the scope of a SUT instance-line – in particular checking the values of attributes in the SUT. TestConditions are offered by a dedicated icon in the drawing tool bar of TestScenarios. The figure above shows an example usage of a TestCondition testing attribute `x` of `class_0` having the value 1.



# Using Interaction Operators in SD TestCases

(see also sample model

- Samples/CSamples/TestConductor/CSDOperators
- Samples/CSamples/TestConductor/CModelCodeCoverage

)

In assertion based testing mode, Interaction Operators can be used in TestScenarios when specifying a TestCase. TestConductor supports the following Interaction Operators:

- `opt`  
The “opt” Interaction Operator must have exactly one operand. Depending on the condition of the operand, the scenario within the operand is considered or ignored during TestCase execution.
- `alt`  
“alt” can have one or more operands. Depending on the conditions of the operands, at most one of the operands is chosen.
- `loop`  
The “loop” operator must have exactly one operand. The operand is repeated as long as the condition of the operand is true.
- `break`  
The “break” operator must have exactly one operand. If the condition of the operand is true, the scenario within the operand is considered and the remainder of the sequence diagram or the enclosing Interaction Operator (if the “break” operator is specified within another operator) is ignored.
- `consider`  
“consider” must have exactly one operand. Normally TestConductor considers all operations/events at least once specified within the sequence diagram. This operator provides a possibility to specify that operations/events should only be considered locally. Operations/events which are only specified within the operator, but not in an enclosing sequence diagram/ Interaction Operator, are ignored outside of the operator and only considered locally within the operator.
- `parallel`  
The “parallel” Interaction Operator can be used to specify a parallel merge between scenarios of different operands. The order within each operand must be adhered to but messages from different operands may be interleaved.  
The same message (i.e. *messages having the same realization in the same classifier*) must not be specified in different operands. Messages of different operands *must not have the same realization in the same classifier*, i.e. they are realized by different operations or events or the receiving life-lines represent different classifiers.

The execution semantics of Interaction Operators can be adapted by using the stereotype <<RTC\_OperatorInfo>> and the tag `RTC_ImmediateEvaluation`. By default it will be waited until the SUT is idle before operand conditions are evaluated for an operator. This is for example the desired behavior if the return value of a SUT operation

call right before an Interaction Operator is used in the condition of the Interaction Operator.

But sometimes operand conditions of an operator have to be evaluated immediately without waiting for some computation to finish before. To support this behavior, stereotype <<RTC\_OperatorInfo>> can be applied on the Interaction Operator and its tag `RTC_ImmediateEvaluation` be checked.

Prior to Rhapsody version 8.2, the conditions of SD Interaction Operators were interpreted directly in the TestCase arbiters, which are generated by 'Update TestCase/TestContext/TestPackage' from the individual TestScenarios. The drawback of this representation is that hence the conditions had to be specified relative to the scope of the respective arbiter – making navigation expressions to e.g. member attributes of SUT or TestContext more complex than necessary.

For Rhapsody 8.2, the strategy has changed: SD Interaction Operator conditions are generated to boolean functions in the TestContext and thus interpreted in the scope of the TestContext. Thus SD Interaction Operator conditions can be specified from the perspective of the TestContext and hence refer e.g to attributes of SUT or TestContext quite more directly and with significantly shorter navigation expressions.

In order to keep existing TestCases in pre 8.2 models valid and to preserve their semantics, the compatibility profile defines property

`TestConductor::TestCase::EvaluateSDOperatorInArbiter` and sets it to `True` on existing TestCases. Overriding this property with `False`, enables the modified treatment of SD Interaction Operator condition in the TestContext also for existing models.

## Using Serialize/Unserialize Functions for User Defined Types

(see also sample model

- `Samples/CppSamples/TestConductor/TestingCookbook/CppListUsage`

)

Rhapsody can animate (display) the values of simple types and one-dimensional arrays. However, if you want to animate a more complex type, the type must be converted to a string (char \*) for Rhapsody to display it. This can be done generally in two different ways, either by using auto-generated serialization/unserialization functions or by using manually defined serialization/unserialization functions.

## Using auto generated serialization/unserialization functions

For enum types and structure types that are explicitly defined in the model, Rhapsody provides the possibility to use automatically generated serialization/unserialization functions in order to display values of these types e.g. in animated sequence diagrams. In order to use the auto generated serialization/unserialization functions for a specific type that is defined in the model, property

`"{Lang_CG}.Type.GenerateSerializationFunctions"` must be set to `"SerializationAndUnserialization"`.

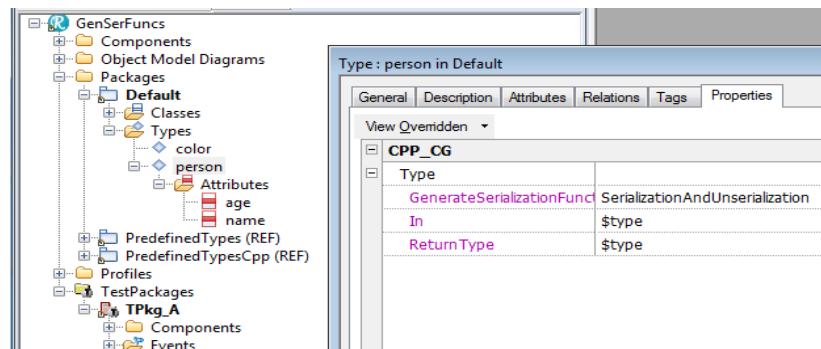


Figure 32:  $\{Lang\}_{CG.Type.SerializationAndUnserialization}$

If this property is set to `SerializationAndUnserialization` for a user defined type, automatically generated serialization and unserialization functions can be used for specifying argument values for message arguments of that type:

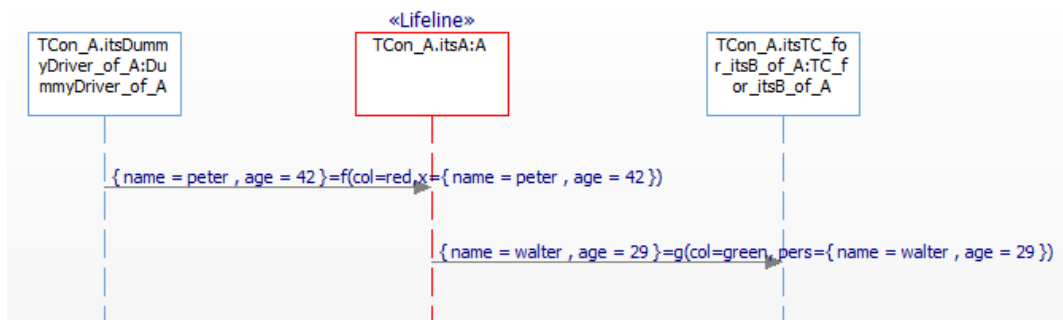


Figure 33: Using Serialization

## Using manually defined serialization/unserialization functions

Besides using the auto generated serialization/unserialization functions of Rhapsody, one can also manually define serialization/unserialization functions. These functions are global instrumentation functions, that takes one argument of the type you want to display, and returns a `char *`. Further information can be found in the chapter *Guidelines for Writing Serialization Functions* of the Rhapsody User Guide. The usage of serialization functions for Testing is demonstrated by the sample model “Samples/CppSamples/TestConductor/CppListUsage”. Please note that serialization functions can only be used for testing purposes if the type that should be serialized is used as an “existing type” in Rhapsody. If only the type signature is used to specify the type of an argument type or return type, serialization functions cannot be used for testing.

# Failure Analysis

---

TestConductor detects and reports a *failure* if a message contained in the message set of a sequence diagram does not appear in the specified order or if a *RTC\_ASSERT* isn't fulfilled during test execution. A message from the message set is specified by its name, the value(s) of its argument(s), the names of sending and receiving objects.

Failure analysis is an important but sometimes difficult task. This is due to the fact that industrial-sized models show very complex behavior, with many messages flowing during test execution.

All possible failures monitored by TestConductor can be caused:

- By errors in the model – the computed model behavior does not meet requirements specified by a sequence diagram
- By inconsistencies in the test configuration or/and in the requirements

In case of using sequence diagrams for test definitions, the task of model debugging is simplified by using TestConductor's graphical failure reports. You can use a combination of diverse Rhapsody analysis capabilities (for example, statechart animation, sequence diagram animation, and sequence diagram comparison) with TestConductor to show test executions as sequence diagrams. The colors and percentage information in the **Execute Test** dialog are useful indicators in determining where the failure occurred.

Remember that during model execution TestConductor ignores all messages which are not specified in the sequence diagram instances of the executed test. TestConductor treats a test specification as violated

- The real order of message actions during model execution does not correspond to specifications in sequence diagram instances.
- The real argument values (w.r.t. directions In, Out, InOut) of messages during model execution do not correspond to those specified in the specification.
- The real return values of messages during model execution do not correspond to those specified in the specification.
- Not all messages preceding the current message in the corresponding run-time instance have already occurred.
- Some user defined assertion fails.
- An expected message does not appear and the TestCase is terminated by timeout.

## Failure Analysis using Witness Scenarios

For TestScenario based TestCases, TestConductor offers with 'Show As SD' a powerful feature for analyzing failed execution results: A witness TestScenario is created which illustrates the successful part of the execution and the exact position of the observation causing the failed result.

The witness TestScenario is a colored copy of the original TestScenario:

- Scenario elements that have already been observed in the executed application according to the specification are shown in *green*. In particular, this means for messages:
  - argument (and return) values have been observed according to the specification.
  - message has been observed in the correct position in the observed sequence.

A green Interaction Occurrence indicates that the referenced TestScenario has been fully traversed without violation of the test specification.

- Scenario elements that aren't yet observed or were omitted are shown in *blue*.
  - note that in Black Box testing self-messages of the SUT are ignored. If self-messages appeared in the specification, they are colored blue in the witness, since self-messages of the SUT can't be observed in Black Box testing.
  - if the TestCase got stuck in execution, all pending TestScenario elements will be colored blue, since test execution was waiting for their occurrence but didn't observe them until the TestCase was terminated.

A blue Interaction Occurrence indicates that the referenced TestScenario has not been fully traversed.

- Scenario elements which were observed contradicting the specification are shown in *red*.

If an argument or return value of a message differs from specification, per default the observed value is shown in the witness scenario.

A *red* message indicates a failure. In the resulting exported sequence diagram, a red message is annotated with a short explanation of the failure, which can be one of the following:

- *Unexpected occurrence of <msg>*  
for Event/Operation/dataflow message.

The value change, operation call or event reception was observed too early – before other still pending observations have been made.

- *Unexpected additional occurrence of <msg>*  
for Event/Operation/dataflow message.

A specified message – dataflow, operation call or event reception – was additionally observed at an instant of time, when it was unexpected.

- *Check of in value of argument <argument> failed*  
for Event/Operation/dataflow message.

The observed value of an operation input argument, an event parameter or a dataflow value differs from the specified value or range for this message.

- *Check of out value of argument <argument> failed*  
for Operation message.

The observed output value -after operation call – of an operation InOut or Out argument differs from the specified value or range for this message.

- Check of return value failed  
for Operation message.

Observed return value differs from specified return value or range.

- *<Assertion> failed*  
a user defined assertion failed.

By default, TestConductor will report the actual argument or return value that causes an argument or return value related assertion to fail in the witness TestScenario<sup>29</sup>.

## Failure Analysis for InteractionOccurrences

'Show As SD' shows a colored witness TestScenario containing all the original TestScenario elements which have been monitored (*green* color) or which were omitted or pending at the moment of witness creation (*blue* color), and also failed messages (*red* color).

An Interaction Occurrence is colored green in the witness scenario if test execution fully traversed the referenced TestScenario.

An Interaction Occurrence is colored red in the witness scenario, if the TestCase failed in the referenced TestScenario.

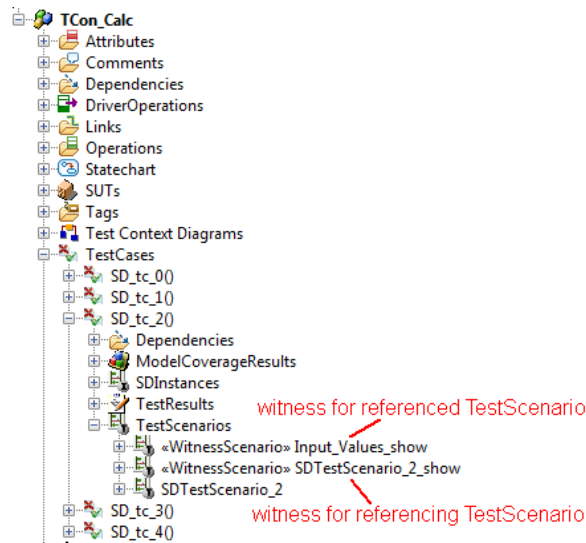
It is colored blue if the execution omitted the Interaction Occurrence or if test execution didn't complete the scenario specified by the Interaction Occurrence.

'Show As SD' will also generate a detailed witness for the TestScenario referenced by an Interaction Occurrence. This witness can be found under the respective TestCase together with the witness of the referencing TestScenario.

---

<sup>29</sup>provided that TestingConfiguration tag `rtc_assert_handling` is set to `by_string`.





*Figure 36: Witnesses for referencing and referenced TestScenario*

## Debugging TestCases

Debugging TestCases using Rhapsody's animation feature can be a powerful tool in failure analysis when animation is available. See section Debugging TestCases on page 76.

## Result Verification

see section Performing result verification for TestCase execution on page 68.



# Using TestConductor from Eclipse

---

As an alternative to the standalone Rhapsody application, Rhapsody can also be used directly from Eclipse (Rhapsody platform integration with Eclipse, see “Integrating Rational Rhapsody and Eclipse” in the Rhapsody online documentation in the IBM knowledge center). Also TestConductor can be used directly from Eclipse when using Rhapsody platform integration with Eclipse; TestConductor does not support Rhapsody work flow integration with Eclipse. In general, all TestConductor functionality can be used when working with Eclipse. Similar to the standalone Rhapsody application, almost all TestConductor functionality is available in context menus of Rhapsody elements, and this holds also when working from Eclipse.

However, there are some differences that needs to be considered when using TestConductor from Eclipse:

- In contrast to executing TestConductor from the standalone Rhapsody application, the test execution windows of TestConductor are not always in front of the Eclipse main window. Selecting the Eclipse main window may hide the TestConductor test execution windows.
- In Eclipse, when creating a new TestArchitecture, TestConductor automatically creates a new Eclipse configuration instead of a normal Rhapsody configuration. Additionally, TestConductor automatically launches the Eclipse New Project Wizard that can be used to create a new Eclipse project that is connected to the created Eclipse configuration.
- *TestConductor does not support Rhapsody work flow integration with Eclipse.*
- *TestConductor does not support computation of code coverage when using Rhapsody platform integration with Eclipse.*

# TestConductor Rhapsody Plugins

---

TestConductor installs some Rhapsody plugins with additional functionality. The plugins are integrated in the TestConductor Testing Profile, this means the plugins are available for Rhapsody projects containing the Testing Profile.

## TestConductor Merge Coverage Reports Plugin

The plugin offers the functionality to merge several model coverage reports into one combined report and to merge several code coverage reports into one combined report.

**Note:** The plugin supports only merging of model or code coverage reports which have been created with Rhapsody 8.0.3 or higher. Merging of reports generated with previous releases of Rhapsody is not supported.

### Merging model coverage reports

This function can be invoked using the menu helper 'Merge Model Coverage Reports'. The helper is available on TestPackages and supports multi selection. After invocation, the helper collects all model coverage reports inside the selected TestPackage(s) and merges them into one combined model coverage report which is added to the model. The combined report contains a list of the merged reports.

If one TestPackage is selected, the combined report is added to this TestPackage. If multiple TestPackages are selected the combined report is added to the joint parent TestPackage of the selected TestPackages (if exist) or to a TestPackage 'MergeModelCoverageResults' if the joint parent of the selected TestPackages is the project itself.

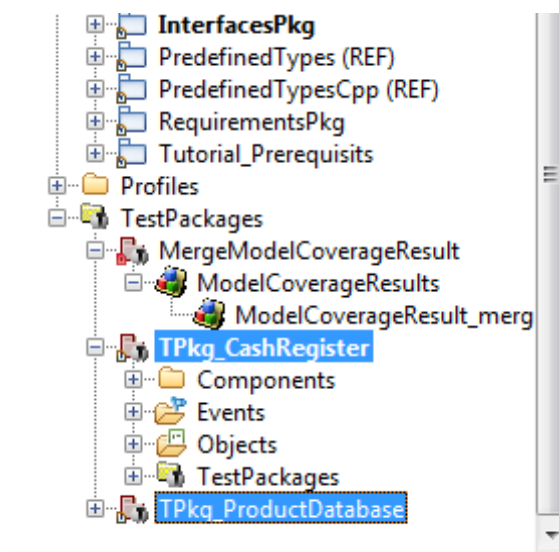


Figure 37: Merging ModelCoverage Results

## Merging code coverage reports

This function can be invoked using the menu helper 'Merge Code Coverage Reports'. The helper is available on TestPackages and on CodeCoverageResults and supports multi selection. After invocation, the helper collects all code coverage reports inside the selected TestPackage(s) or the selected CodeCoverageResults and merges them into one combined code coverage report which is added to the model.

If one TestPackage is selected, the combined report is added to this TestPackage. If multiple TestPackages or CodeCoverageResults are selected the combined report is added to the joint parent TestPackage of the selected elements (if exist) or to a TestPackage 'MergeCodeCoverageResults' if the joint parent of the selected elements is the project.

**Note:** Merging of code coverage reports for one source code file is supported only if the different incarnations of this source code file are the same. If for example operations have been added or removed or if statecharts have been modified between the generation of the code coverage reports to be merged, then the combined code coverage report will be wrong (and the report contains a warning).

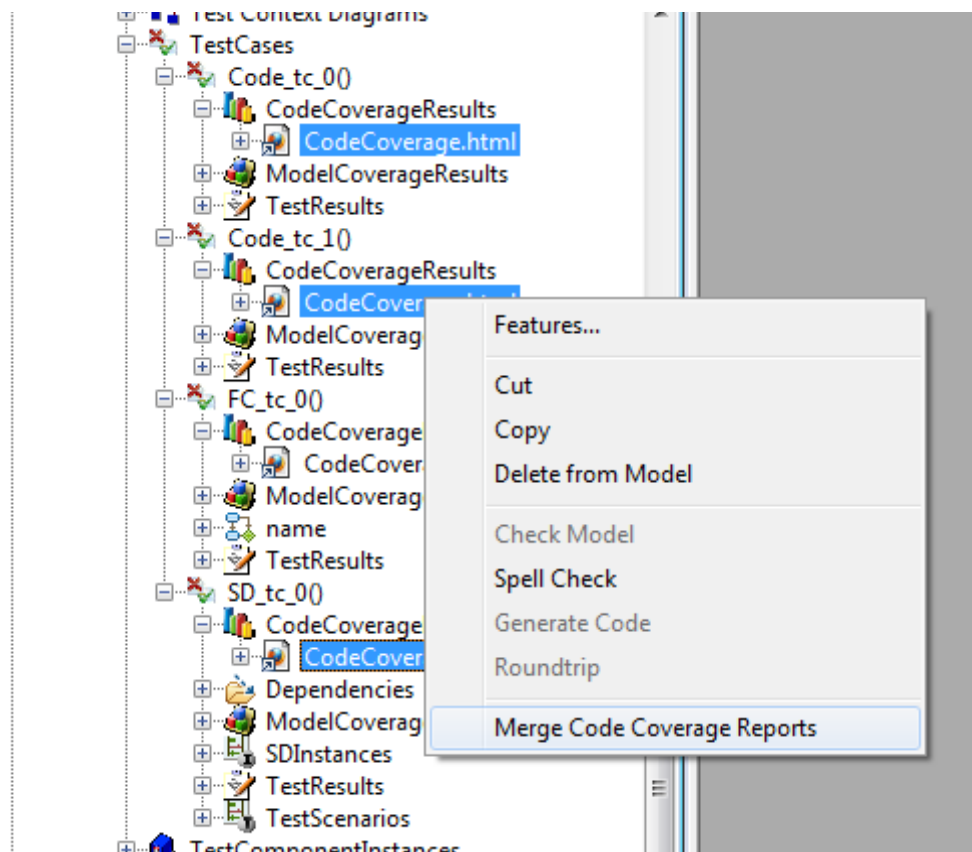


Figure 38: Merging CodeCoverage Results

## Merging requirement coverage reports

This function can be invoked using the menu helper 'Merge Requirement Coverage Reports'. The helper is available on TestPackages and on RequirementCoverageResults and supports multi selection. After invocation the helper collects all requirement coverage reports inside the selected TestPackage(s) or the selected RequirementCoverageResults

and merges them into one combined requirement coverage report which is added to the model.

If one TestPackage is selected, the combined report is added to this TestPackage. If multiple TestPackages or RequirementCoverageResults are selected the combined report is added to the joint parent TestPackage of the selected elements (if exists) or to a TestPackage 'MergeRequirementCoverageResult' if the joint parent of the selected elements is the project itself.

**Note:** Requirement coverage reports can only be merged if the settings the reports have been generated with (stored in their model based testing tags) are identical. If the settings of different requirement coverage reports are not compatible only a subset of the selected requirement coverage reports are merged. Two additional tags, `involved_coverage_results` (contains all the reports that are part of the merge result) and `ignored_coverage_results` (contains all reports that are omitted from the merge process), are added to a resulting requirement coverage result to document which reports are included in the merged report.

## TestConductor Rhapsody Quality Manager Plugin

TestConductor TestCases can be referenced and executed from Rational Quality Manager. A detailed description how to integrate Rational Quality Manager and TestConductor can be found

In the document “RQMTestConductorAdapter\_HowTo.pdf” in `<Rhapsody installation>/Doc/pdfbooks`.

To improve the integration between TestConductor and RQM, this plugin introduces the possibility to directly create and link RQM TestScripts while working with Rhapsody and TestConductor. An additional Helper 'Create RQM TestScript' is available which is applicable on TestCases, TestContexts and TestPackages.

After running the helper, the user has to specify the RQM server to connect to, user login and password for the server as well as the ProjectArea where the TestScript should be created.

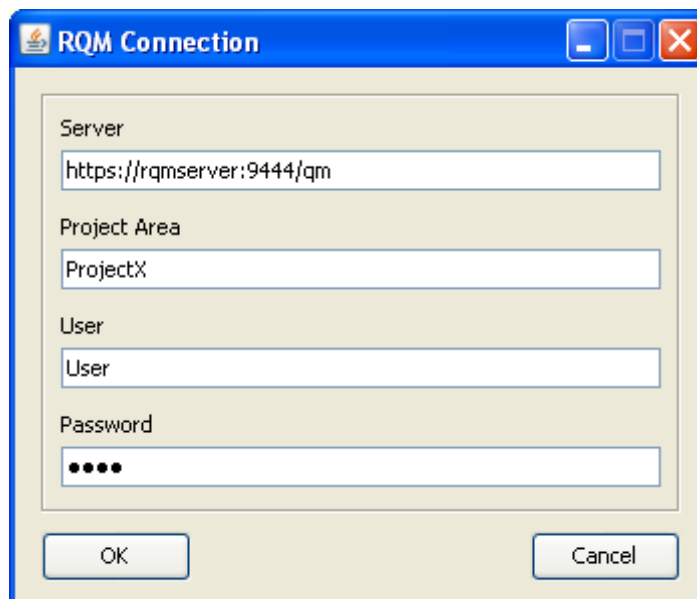


Figure 39: RQM Connection

After that, a RQM command line TestScript will be created in the specified ProjectArea. The required fields of the command line TestScript like the path to the used Rhapsody model or the full model path to the element which should be tested are set automatically. If additional options should be specified for the test, the necessary adaptations have to be done manually.

If the model is located on a RDM (Rational Design Manager) Server, the execution variables SERVER\_URL, PROJECT\_AREA\_NAME, STREAM\_NAME, USER\_NAME and PASSWORD are automatically added to the TestScript.

In RQM, the TestScript can now be executed using the TestConductor RQM Adapter as described in the document “RQMTestConductorAdapter\_HowTo.pdf”

Also a Hyperlink to the newly created RQM TestScript is added automatically underneath the model element for which the helper has been called. Following the Hyperlink, the RQM TestScript can be opened directly from Rhapsody.

**Note:** This functionality is not available when using Rhapsody in Eclipse platform integration.

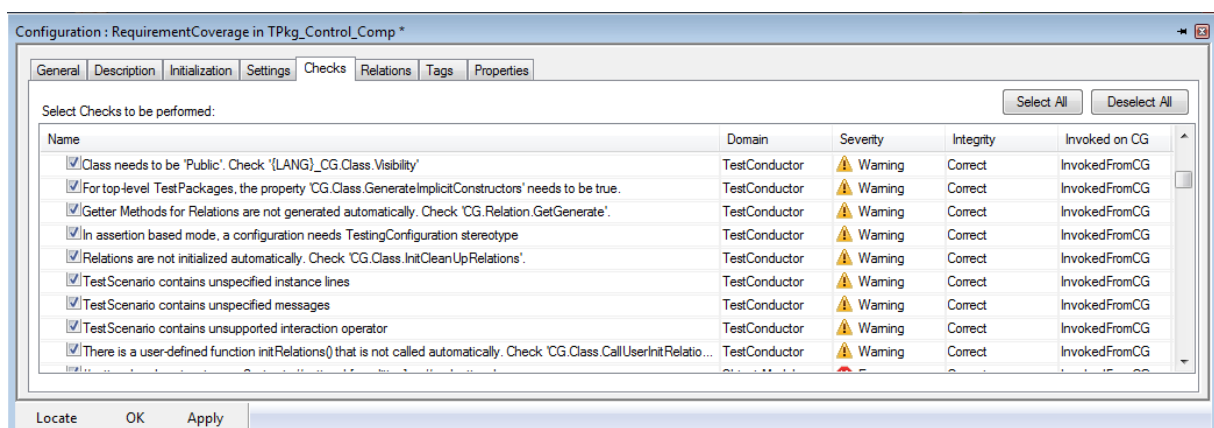
## TestConductor Check Model Plugin

Rhapsody has a checker feature which provides the possibility to perform structural and behavioral checks of the model. In addition to the predefined internal checks which are included in Rhapsody, further external checks can be defined and added to the list of checks.

The model checks can either be performed for the active configuration or for selected classes (Tools -> Check Model). The TestConductor checks are also automatically invoked from the code generation.

More information about Rhapsody model checks in general can be found in the Rhapsody User Guide in the chapter 'Checks'.

If the TestingProfile is loaded, the external TestConductor model checks are available. For these checks TestConductor is set as its domain.



The following TestConductor checks are currently available:

- Class needs to be 'Public': Check '{Lang}\_CG.Class.Visibility' (Warning)

- For top-level TestPackages, the property 'CG.Class.GenerateImplicitConstructors' needs to be true. (Warning)
- Getter Methods for Relations are not generated automatically. Check 'CG.Relation.GetGenerate' (Warning)
- In assertion based mode, configuration needs <<TestingConfiguration>> stereotype (Warning)
- Relations are not initialized automatically. Check 'CG.Class.InitCleanUpRelations'.
- TestScenario contains unspecified life-lines (Warning)
- TestScenario contains unspecified messages (Warning)
- TestScenario contains unsupported interaction operator (Warning)
- There is user-defined function initRelations() that is not called automatically. Check 'CG.Class.CallUserInitrelations'. (Warning)

# Appendix

---

## Definitions of the Rhapsody Testing Profile

### Structure Overview

The Rhapsody Testing Profile is prearranged in four major packages with additional sub-packages and the *TestingProfile* stereotype.

- Rhapsody Testing Profile (**UML20TP**) – contains the adaption of the basic definitions taken from the UML TestingProfile. Detailed information on page 143
  1. TestArchitecture
  2. TestBehavior
- Rhapsody TestConductor (**RTC**) – detailed information on page 145
  1. TestArchitecture
    1. CppUnit
    2. CUnit
    3. Diagrams
    4. SDReuse
    5. TestRT
  2. TestBehavior
  3. TestDocumentation
- Automatic Test Generation (**ATG**)
  1. Documentation
- Rhapsody Formal Testing (**FormalTesting**)
  1. Architecture

### UML Testing Profile (UML20TP) Package

The UML20TP package contains stereotypes and new terms derived from the official UML Testing Profile. It consists of two major packages:

- *TestArchitecture* and
- *TestBehavior*

## TestArchitecture Package

The *TestArchitecture package* consists of the stereotypes

- ***SUT***  
The *system under test* (SUT) is the component being tested. A SUT can consist of several objects. The SUT is exercised via its public interface operations and events by the TestComponents, the TestContext or the system environment (ENV).
- ***TestComponent***  
A *TestComponent* is a class of a test system. The TestComponent objects (TestComponentInstances) realizes partially the behavior of a TestCase. An instance of a TestComponent may have a set of interfaces which are used to communicate via connections with other TestComponent instances or with SUT objects. It also may have operations, so called driver operations (DriverOperations) that can drive SUT operations or call events of the SUT and so called StubOperations (StubOperations) which are able to generate necessary “stub” return values.
- ***TestConfiguration***  
The *TestConfiguration* is a dependency to a code generation configuration. Depending on this configuration the code for the complete TestContext including its TestCases can be generated, built and executed.
- ***TestContext***  
A *TestContext* describes the context in which TestCases are executed. A TestContext is responsible for defining the structure of the test system, i.e., which TestComponent instances and which SUT objects exists and how they are interconnected. The TestComponent instances and SUT objects are normally parts of a TestContext. Since TestCases are operations of a TestContext, a TestCase can access both the TestComponent instances and also the SUT objects.

## TestBehavior Package

The *TestBehavior package* contains two stereotypes named

- ***TestCase***  
A *TestCase* is a specification of one case to test the system under test including what to test. It defines the input stimuli and the expected results to be observed. It implements a test objective. A TestCase is an operation of a TestContext (described above).
- ***TestObjective***  
A *TestObjective* is a named element describing what should be tested. It is associated to a TestCase.



## TestConductor (RTC) Package

E.g. the term 'TestComponent' is defined in the UML TestingProfile 1.2 (<http://www.omg.org/spec/UTP/1.2>) as part of the test environment and is meant to be an extension of class. For practical purposes TestConductor extends this terms also to TestFiles (Rhapsody in C) and to TestComponentInstances (objects of TestComponents instantiated as part of the TestContext) and TestComponentObjects (global objects of TestComponents instantiated outside the TestContext). Such extensions of the UML TestingProfile for practical purposes are collected in RTC::TestArchitecture and RTC::TestBehavior instead of the UML20TP package and its sub packages).

The TestConductor (RTC) package consists of the packages

- *TestArchitecture* (page 145)  
The TestArchitecture package contains further sub packages:
  - **TestRT** – stereotypes and types for using Rational Test Realtime with TestConductor – support for TestRT integration with TestConductor is deprecated,
  - **CppUnit** – definitions for integration of the CppUnit testing framework with TestConductor (relevant for animation based testing mode only)
  - **Diagrams** – in this sub package only the new term `<<TestContextDiagram>>` (on structure diagram) is defined.
  - **CUnit** – definitions for integration of the CUnit testing framework with TestConductor (relevant for animation based testing mode only)
  - **SDReuse** – see section Creating Sequence Diagram TestCases from existing Scenarios using an explicit instance mapping on page 55 ff for a detailed explanation of the new terms defined in this sub package.
- *TestBehavior* (page 150)
- *TestDocumentation* (page 154)

## TestArchitecture Package

The *TestArchitecture package* contains the stereotypes:

- Sub package *CppUnit* – *CppUnit-Testing is supported only in animation based testing mode.*  
*Sub packages* CppUnit and CUnit contain stereotypes for the integration of CppUnit and CUnit testing with Rhapsody.
  - **CppUnitConfig**  
Stereotype `<<CppUnitConfig>>` can be applied to a configuration and provides a set of tags for customization of the CppUnit testing integration with Rhapsody.
  - **CppUnitContext**  
Stereotype `<<CppUnitContext>>` can be applied to a class and sets some properties for CppUnit testing integration. A TestContext can be

'changed to' CppUnitContext – and vice versa – by right-clicking a TestContext and selecting “Change to > CppUnitContext”.

- Sub package *CUnit* - *CUnit-Testing is supported only in animation based testing mode*.  
Sub packages CppUnit and CUnit contain stereotypes for the integration of CppUnit and CUnit testing with Rhapsody.
  - **CUnitConfig**  
Stereotype <<CUnitConfig>> can be applied to a configuration and provides a set of tags for customization of the CUnit testing integration with Rhapsody.
  - **CUnitContext**  
Stereotype <<CUnitContext>> can be applied to a class and sets some properties for CUnit testing integration. A TestContext can be 'changed to' CUnitContext – and vice versa – by right-clicking a TestContext and selecting “Change to > CUnitContext”.
- Sub package *Diagrams*
  - **TestContextDiagram**  
A TestContext diagram (TestContextDiagram) is a structure diagram that contains the SUT instances, the TestComponent instances and their interconnections. It is used to define the structure of the TestContext graphically.  
  
The TestContext diagram is generated during the TestArchitecture generation inside the TestContext.
- Sub package *SDReuse*
  - **ActiveSDMapping**  
Only one SDMapping can be active at any instant of time. The active SDMapping is selected by setting this stereotype.
  - **maporigin**  
Stereotype on dependency. TestConductor adds such stereotyped dependency on the respective source TestScenario or SequenceDiagram to mapped TestScenarios.
  - **SDInstanceRealizationMapPair**  
New term on constraint. Simple mappings of individual classifiers to classifiers, SDInstanceRealizationMapPair has two tags 'Origin' and 'Target' of type ModelElement. life-lines referring to 'Origin' shall be mapped to 'Target'.
  - **SDInstanceRealizationMerge**  
New term on constraint. Defines merging life-lines of a set of classifiers to one life-line of a particular classifier.  
SDInstanceRealizationMerge has a tag 'Target' denoting the classifier for which the origins will be merged and arbitrary many SDInstanceRealizationMergeOrigin elements
  - **SDInstanceRealizationMergeOrigin**  
New term on constraint. Defines tag 'Origin'. The set of

`SDInstanceRealizationMergeOrigin` elements belonging to a `SDInstanceRealizationMerge` define the set of elements for which the referring life-lines shall be merged to a life-line referring to 'Target' classifier.

- ***SDInstanceRealizationSplit***  
New term on constraint. Defines splitting life-lines of into a set of life-lines of particular classifiers. `SDInstanceRealizationSplit` has tag 'Origin' for defining, which Classifier shall be split and can have arbitrary many `SDInstanceRealizationSplitTargets`
- ***SDInstanceRealizationSplitTarget***  
New term on constraint. Defines tag 'Target'. The set of `SDInstanceRealizationSplitTarget` elements belonging to a `SDInstanceRealizationSplit` define the set of classifiers to which the life-lines referring to 'Origin' classifier shall be split.
- ***SDMapping***  
New term on constraint. The top level element of each mapping is an `SDMapping`
- ***SDMappingTable***  
new term on Table View with layout `SDMappingTable` as defined in sub package `SDReuse`.
- **AUTOSAR\_RTE**
- **AUTOSAR\_RTEInstance**
- **Arbiter**  
Stereotype Arbiter is used by TestConductor for auto generated TestComponents that control the execution of a SD TestCase.
- **ArbiterInstance**  
Stereotype ArbiterInstance is used by TestConductor for TestComponent instances that are instances of Arbiter TestComponents.
- **ControlArbiter**  
Stereotype ControlArbiter is used by TestConductor to mark a dependency of a SD TestCase on an Arbiter TestComponent that controls the SD TestCase.
- **instantiated**  
Stereotype instantiated is used to label associations that are always instantiated with a valid link during runtime. TestConductor interprets associations labeled with this stereotype like links. <<instantiated>> associations are expected to own a stereotyped dependency on the object to which the association will be initialized at run time. This dependency will be stereotyped <<usedSUTObject>> if the association points to an object used as SUT. It will be stereotyped <<UsedTestComponentObject>> if the associations points to a TestComponentObject.
- **usedSUTObject**  
see <<instantiated>> stereotype on association ends.
- **usedTestComponentObject**  
see <<instantiated>> stereotype on association ends.

- **NoConsoleApp**  
Stereotype <<NoConsoleApp>> can be applied to configurations in order to suppress opening a console when running the application. This stereotype has no effect on <<TestingConfiguration>> code generation configurations and is used only in animation based testing mode. In assertion based testing mode, a NoConsoleApp tag of <<TestingConfiguration>> stereotype can be used to suppress consoles for execution.
- **ParameterTable**  
Stereotype <<ParameterTable>> is used to mark a controlled file as a parameter table definition that contains values for all external test parameters of a TestContext.
- **replacement**  
Stereotype <<replacement>> is used to mark a dependency of a TestComponent on the original class that is replaced by the TestComponent in the TestArchitecture.
- **greyboxreplacement**  
Stereotype <<greyboxreplacement>> is used to mark a dependency of a <<TestSUT>> on the original class that is replaced by the <<TestSUT>> in the TestArchitecture (for Grey Box Testing).
- **greyboxinstancereplacement**  
Stereotype <<greyboxinstancereplacement>> is used to mark a dependency of a SUT greybox object (implicit object) on the implicit object that is replaced by the greybox SUT object in the TestArchitecture.
- **instancereplacement**  
Stereotype <<instancereplacement>> is used to mark a dependency of a TestComponentObject (implicit object) on the implicit object that is replaced by the TestComponentObject in the TestArchitecture.
- **filereplacement**  
Stereotype <<filereplacement>> is used to mark a dependency of a test file on the original file that is replaced by the test file in the TestArchitecture (Rhapsody in C).
- **scheduled**  
Stereotype <<scheduled>> is used to mark a dependency of a TestContext on a Scheduler TestComponent that controls the starting and stopping of TestCases of the TestContext.
- **Scheduler**  
Stereotype <<Scheduler>> is used to mark an auto generated TestComponent that is used to control the activation and termination of TestCases.
- **SCTCInstance**  
Stereotype <<SCTCInstance>> is used to mark a TestComponentInstance to be an instance of a statechart TestCase TestComponent.
- **stubbed**  
Stereotype <<stubbed>> is used to mark an operation of a TestComponent to be stubbed, i.e., that the behavior of the operation has been changed for testing purposes.

- **Stub**  
Stereotype <<Stub>> prevents model elements in TestComponent, TestComponentObject, TestFile, TestSUT, and TestSUTObject from being modified by TestArchitecture update. TestArchitecture update will omit updating model elements stereotyped <<Stub>>.
- **TestActor**  
New term <<TestActor>> is used for TestComponents that have the role of an actor in the TestArchitecture. Test actors replace actors for testing purposes.
- **TestFile**  
New term <<TestFile>> is used for test files in the TestArchitecture. TestFiles replace files of the design for testing purposes.
- **TestComponentInstance**  
New term <<TestComponentInstance>> is used to specify instances of TestComponents.
- **TestComponentObject**  
New term <<TestComponentObject>> is used to stereotype copies of implicit objects in the role of TestComponents.
- **TestingConfiguration**  
Stereotype <<TestingConfiguration>> is used to mark a configuration that is used for testing purposes. The stereotype <<TestingConfiguration>> provides several tags that can be used in order to define specific settings for the generated testing code (cf section Tags of the <<TestingConfiguration>> Stereotype on 60)
- **TestPackage**  
New term <<TestPackage>> represents a package that contains testing related model elements, e.g. other TestPackages, TestContexts or TestCases. It allows grouping of multiple test related elements into one package, and it can be used to separate testing related elements from design related elements.
- **TestParameter**  
Stereotype <<TestParameter>> is used to mark an attribute of a TestContext to be a parameter that can be controlled by a TestingConfiguration by using a <<ParameterTable>> controlled file.
- **TestLink**  
Stereotype <<TestLink>> is a stereotype on links or connectors (SysML). <<TestLink>> sets a code generation property that forces generation of link initialization code for link, regardless of its location in the design/TestArchitecture hierarchical. Normally, a link has to be located at least on the least level containing the linked instances. Using stereotype <<TestLink>> allows TestConductor defining the link locally to the TestArchitecture although the link refers to instances anywhere in the browser hierarchy.
- **use\_ParameterTable**  
Stereotype <<use\_ParameterTable>> is used to mark a dependency of a TestingConfiguration on a <<ParameterTable>> controlled file in order to specify that the TestingConfiguration shall apply the linked parameter table for the test parameters of the TestContext for which the TestingConfiguration generates code for.

- **use\_replacement**  
Stereotype <<use\_replacement>> is used to mark a dependency of a TestComponentInstance on a TestComponent that is a replacement of a design class for testing purposes.
- **use\_greyboxreplacement**  
Stereotype <<use\_greyboxreplacement>> is used to mark a dependency of a SUT instance on a TestSUT – which is a replacement of a SUT class for Grey Box testing purposes.
- **use\_greyboxinstancereplacement**  
Stereotype <<use\_greyboxinstancereplacement>> is used to mark a dependency of the TestContext on a TestSUTObject (i.e. a greybox replacement of an implicit SUT object).
- **use\_instancereplacement**  
Stereotype <<use\_instancereplacement>> is used to mark a dependency of the TestContext on a <<TestComponentObject>> (i.e. a greybox replacement of an implicit object used in the role of a TestComponent).
- **use\_filereplacement**  
Stereotype <<use\_filereplacement>> is used to mark a dependency of a TestContext on a test file indicating that this test file is used by the TestContext for testing purposes.
- **use\_superclass**  
Stereotype <<use\_superclass>> is used to mark a dependency of a replacement on a replacement of its super class. If a replacement for a class with generalization is introduced, then the generalization of the replacement always refers to the original super class in the model, regardless of existing replacement for the super class. A <<use\_superclass>> stereotyped dependency is introduced for navigation from a replacement to the replacement of its super class.
- **TestSUT**  
New term <<TestSUT>> is used to mark a replacement class that is basically a copy of the original SUT class (used only for Grey Box Testing).
- **TestSUTObject**  
New term <<TestSUTObject>> is used to mark a replacement object (basically a copy of the original implicit SUT object – used only for Grey Box Testing).

## TestBehavior Package

The *TestBehavior package* is composed of a number of stereotypes like:

- **call\_virtual**
- **call\_nonvirtual**
- **consideredTestCase**
- **CodeCoverageResult**  
A CodeCoverageResult is a document that reports the code coverage by one or more TestCases. Code coverage computation is supported only for assertion based testing mode. Code coverage can be enabled using tag ComputeCodeCoverage on the TestingConfiguration.

- **CoverageResult**  
A CoverageResult is a document that reports which model elements are covered by one or more TestCases. This stereotype is maintained only for compatibility reasons.
- **CodeCoverageResultRef**
- **ModelCoverageResult**  
A ModelCoverageResult is a document that reports which model elements are covered by one or more TestCases. Model coverage can be enabled using tag `ComputeModelCoverage` on the `TestingConfiguration` for assertion based testing mode. For animation based testing mode, model coverage is enabled by property `TestConductor.TestCase.ComputeCoverage`.
- **ModelCoverageResultRef**
- **RequirementCoverageResult**
- **RequirementCoverageResultRef**
- **ExecutedElement**
- **GeneralResult**  
Base stereotype of `TestResult`, `ModelCoverageResult`, `RequirementCoverageResult`, `CodeCoverageResult`.
- **GeneralResultRef**  
Stereotype on Hyperlink for unique treatment of results (cf. `<<GeneralResult>>`).
- **DefaultOperation**  
A DefaultOperation defines the default behavior of an operation of a `TestComponent`. A `TestCase` in which the behavior of this operation is not explicitly specified uses this default behavior in the current `TestCase` execution.
- **DefaultTriggeredOperation**  
A DefaultTriggeredOperation defines the default behavior of a triggered operation of a `TestComponent`. A `TestCase` in which the behavior of this triggered operation is not explicitly specified uses this default behavior in the current `TestCase` execution.
- **DriverOperation**  
A DriverOperation is an operation of a `TestComponent` which is able to inject input stimuli to the SUT objects. It is generated automatically by `TestConductor` for the `TestComponent` class that calls a message of a SUT object defined in a sequence diagram. During execution of the `TestCase`, `TestConductor` calls the driver operation, and as a result the `TestComponent` stimulates the SUT as it is described in the used sequence diagram.
- **RTC\_InstInfo**  
The stereotype *RTC\_InstInfo* contains two tags *RTC\_IgnoreSCBehavior* and *RTC\_Monitor*. When adding this stereotype to a life-line of a `TestScenario`, the user can set these tags. `TestConductor` uses these tags when executing the test. If the tag *RTC\_IgnoreSCBehavior* is set, `TestConductor` ignores the normal statechart behavior of the tagged instance. If the tag *RTC\_Monitor* is set, `TestConductor` just monitors all messages starting from the tagged instance.

- **RTC\_MsgInfo**  
The stereotype RTC\_MsgInfo contains tags RTC\_Monitor, RTC\_Receiver, etc. When adding this stereotype to a message in a TestScenario, the user can set these tags. If the tag RTC\_Monitor is set, the tagged message is just monitored by TestConductor. If the tag RTC\_Receiver is set, the tagged value is used as the real receiver instance of the tagged message. If the tag RTC\_DriverCallCode is set, TestConductor generates the string contained in this tag instead of the standard call code TestConductor generates for driver operations. If the tag RTC\_InitCode is set, TestConductor generates the string contained in this tag instead of the standard init code TestConductor generates for driver operations. If the tag RTC\_MsgId is set, the specified string is used to reference the message in macros RTC\_ASSERT\_SD\_NAME. If the tag RTC\_StubBodyCode is used, TestConductor generates the string contained in this tag instead of the standard stub code TestConductor generates for StubOperations. For further information please read the chapter User Defined DriverOperation at page 119.
- **RTC\_OperatorInfo**  
Stereotype applicable on InteractionOperator (cf. section Using Interaction Operators in SD TestCases on page 129 ff).
- **RTC\_RefInfo**  
The stereotype RTC\_RefInfo is used internally for unique identification of messages in sequence diagrams which are referenced by other sequence diagrams.
- **RemoteRequirementReference**
- **SDInstance**  
*A sequence diagram instance (SDInstance) represents one instance of a TestScenario. When using a sequence diagram for testing purposes, several parameters must be defined that influence the behavior of a TestCase. A combination of a sequence diagram with such a set of parameters forms a sequence diagram instance.*
- **SelfMessageRealizationInParts**  
Stereotype applicable to SequenceDiagram. If stereotype is set, Rhapsody's SD Editor offers for message realizations not only the interface items of the classifier itself that is represented by the receiver life line, but also the receptions and operations of parts of the respective classifier.  
<<SelfMessageRealizationInParts>> is used in GreyBox testing for specifying scenarios involving also parts in the SUT.
- **StatechartTestCase**  
Stereotype is used to stereotype the dependency of a statechart TestCase on the TestComponent owning the statechart defining the test.
- **StubbedOperation**  
*A stubbed operation is an operation for which at least one TestCase specifies a behavior that is different from the default behavior. The different behavior is stored in a stub-operation. The stubbed operation decides at runtime depending on the executed TestCase if either the default behavior should be executed or a specific stub-operation.*
- **StubOperation**  
A StubOperation is a replacement of an operation of a TestComponent class for an individual call. It realizes the code for the individual operation call return value



specified in the referenced sequence diagram. The code of the StubOperation is generated automatically by TestConductor.

- **TCRequirementCoverageDocument**
- ***considerForCoverage***  
Normally, the coverage scope is automatically computed w.r.t. to the CoverageKind tag of the <<TestingConfiguration>> code generation configuration (cf. Tags of the <<TestingConfiguration>> Stereotype on page 60 and section Choosing the Coverage Kind for Model Coverage on page 81). In order to force a class, object or file to be considered in coverage scope computation, a <<considerForCoverage>> dependency on the respective model element can be added to the TestContext.
- ***considerNotForCoverage***  
Normally, the coverage scope is automatically computed w.r.t. to the CoverageKind tag of the <<TestingConfiguration>> code generation configuration (cf. Tags of the <<TestingConfiguration>> Stereotype on page 60 and section Choosing the Coverage Kind for Model Coverage on page 81). In order to disregard a class, object or file that is by default considered in coverage scope computation, a <<considerNotForCoverage>> dependency on the respective model element can be added to the TestContext.
- ***TestAction***  
A *TestAction* is an action block that can be placed on life lines in TestScenarios. There are different kinds of TestActions: <InitAction>, <PreCallAction>, <CallAction>, <PostCallAction>, <StubAction>. Inside these actions, one can place e.g. assertions to perform complex checks on output values (return or out arguments), or one can write code that initializes complex input data. These kinds of TestActions correspond to the tags of <<RTC\_MsgInfo>>
  - <InitAction> - RTC\_DriverInitCode
  - <PreCallAction> - RTC\_DriverInitCodeAdditional
  - <CallAction> - RTC\_DriverCallCode
  - <PostCallAction> - RTC\_DriverCallCodeAdditional
  - <StubAction> - RTC\_StubBodyCode

Note, that both specification techniques are mutual exclusive. If such TestActions are used in order to determine the code populated for the respective message, the RTC\_MsgInfo tags are ignored for this message.

See section (general) TestActions, TestAssignments and TestConditions on page 125 and Influencing DriverOperation and Stub generation using TestActions in TestScenarios on page 122 for details.

- ***TestAssignment***  
A TestAssignment is a special form of a TestAction that can be placed on a SUT instance line in a TestScenario in order to assign an attribute of the SUT a particular value during test execution. See section (general) TestActions, TestAssignments and TestConditions on page 125 for details.
- ***TestCondition***  
A TestCondition is a special form of a TestAction that can be placed on a SUT

instance line in a TestScenario in order to check the value of an attribute of the SUT during test execution. See section (general) TestActions, TestAssignments and TestConditions on page 125 for details.

- **TestResult**  
A test result (TestResult) represents an outcome of an execution of a TestCase. It is a textual report that contains detailed information about the TestCase execution, e.g. if the TestCase has passed or failed.
- **TestResultref**
- **TestScenario**  
The stereotype *TestScenario* (*TestScenario*) contains two tags *RTC\_ActivationCondition* and *RTC\_SDPParameters*. When adding this stereotype to a TestScenario, the user can set these tags. With the tag *RTC\_ActivationCondition* the user can specify the activation condition of the sequence diagram. With the tag *RTC\_SDPParameters* the user can set the parameters of the sequence diagram.
- **Unrealized**  
Messages with stereotype Unrealized are filtered out and ignored during test execution. See also section Message Realization on page 110).
- **WitnessScenario**  
TestScenario witnesses obtained by 'Show as SD' (cf. section Failure Analysis using Witness Scenarios on page 133) or generated due to `<<TestingConfiguration>>` tag `CreateWitnessScenariosForFailedSDTestCase` (cf. section Tags of the `<<TestingConfiguration>>` Stereotype on page 60) are stereotyped `<<WitnessScenario>>`.
- **nopublicwrapper**  
Although a lot of code generation related properties are regarded for generation of public wrappers for private (and protected) operations, the feature TestConductor Support for Testing Private Operations in Rhapsody in C and TestConductor Support for Testing Private and Protected Operations in Rhapsody in C++, respectively (cf. pages 46 and 47, respectively) may produce not compilable code under some circumstances. In such cases, stereotype `<<nopublicwrapper>>` can be applied on individual functions of the class or object to be tested, in order to omit public wrapper generation for the respective function.

## TestDocumentation Package

The *TestDocumentation* package contains a Matrix-Layout *TestRequirementCoverage* and Table-Layouts *TestResultTable* and *TestCaseResultTable* for presenting test relations with requirements and test execution results in matrix and table notation.

The layouts can be chosen as layout in the features dialog of matrix and table views, respectively.

The Rhapsody Testing Profile also defines the following new terms to ease the use of these matrix and table definitions:

- **TestRequirementMatrix**

- **TestResultTable**
- **TestCaseResultTable**

For example, a *TestRequirementMatrix* view can simply be added to a package by using the context menu item 'Add New->TestingProfile->TestRequirementMatrix' on the package.

A *TestRequirementMatrix* shows in an array view if and how requirements are tested by TestCases. The left hand side of the array shows all existing TestCases. The upper side shows all the requirements. The cells contain an entry if a TestObjective from the TestCase to the requirement exists in the model, for instance from TestCase *Code\_tc\_0* to requirement *REQ1*.

A *TestResultTable* shows in a table form the existing TestCases and their current result values. The left column of the table shows all existing TestCases. The right column shows the current TestCase results, for instance verdict *Passed* for TestCase *Code\_tc\_0*.

## Automatic Test Generation (ATG) Package

The *ATG package* consists of several stereotypes which are enhancements to the UML Testing Profile. For more information about the ATG package and its stereotypes please refer also the *Rhapsody Automatic Test Generation (ATG) User Guide*.

## Formal Testing Package

The *FormalTesting* package consists of several stereotypes which are enhancements to the UML Testing Profile to support *Formal Testing* using the *Rhapsody Formal Testing* add on. For more information about the stereotypes of this package please refer to the documentation of the add on.

## TestConductor Assert Macros (C/C++)

(see also TestingCookbook:

- “Which assertions can I use in the test case specification?”

)

As described in chapter *TestCase Definition with Code* on page 42 and in chapter *TestCase Definition with Flow Charts* on page 43 and in chapter *TestCase Definition with Statecharts* on page 43, pre-defined assertion macros are used to get results from a *TestCase* execution.

TestConductor defines several assertion macros (C/C++) listed below. Each macro might have one up to four arguments with the following notation:

*n* = Name of the assertion (String, e.g. „Check 1“)

*e*, *e1*, *e2* = Boolean Expression (e.g. *i* != 23)

*p* = text of message printed in the sequence diagram

*sd\_instance\_name* = Reference to the instance name of the sequence diagram

*msgid* = Reference to the message id of a message in the sequence diagram

- **RTC\_ASSERT** (*e*)  
Assertion with default name *e*. The assertion is *PASSED*, if the result of the boolean expression is *TRUE* (*e*!=0), otherwise the assertion *FAILED*.
- **RTC\_ASSERT\_FATAL** (*e*)  
Assertion with default name *e*. The assertion is *PASSED*, if the result of the boolean expression is *TRUE* (*e*!=0), otherwise the assertion *FAILED*. If it is failed, the *TestCase* is aborted immediately without executing further assertions.
- **RTC\_ASSERT\_NAME** (*n*, *e*)  
Named assertion. The user can define the name of the assertion within the argument *n*. The assertion is *PASSED*, if the result of the boolean expression is *TRUE* (*e*!=0), otherwise the assertion *FAILED*.
- **RTC\_ASSERT\_NAME\_FATAL** (*n*, *e*)  
Named fatal assertion. The user can define the name of the assertion within the argument *n*. The assertion is *PASSED*, if the result of the boolean expression is *TRUE* (*e*!=0), otherwise the assertion *FAILED*. If it is failed, the *TestCase* is aborted immediately without executing further assertions.
- **RTC\_ASSERT\_SD** (*sd\_instance\_name*, *msgid*, *e*)  
Assertion that can be used in *TestScenarios*. If such an assertion is used in e.g. a driver operation or a *StubOperation*, and *sd\_instance\_name* refers to a sequence diagram instance, and *msgid* refers to a message id of a message in the sequence diagram of the sequence diagram instance then the result of evaluating expression *e* will be associated with the designated message.
- **RTC\_ASSERT\_SD\_NAME** (*sd\_instance\_name*, *msgid*, *p*, *e*)  
Similar to **RTC\_ASSERT\_SD**. The user has to define the string argument *p*, (e.g. *p*=“*Evaluation of my return value*” which will be concatenated with the result of

the assert macro (*PASSED*, *FAILED* etc.) and printed as result message, e.g. “Evaluation of my return value failed.”<sup>30</sup>

- `RTC_ASSERT_SD_NAME_ACTUAL(sd_instance_name, msgid, p, e, a)`<sup>31</sup>  
Similar to `RTC_ASSERT_SD`. The user has to define the string argument `p`, (e.g. `p="Evaluation of my return value"` which will be concatenated with the result of the assert macro (*PASSED*, *FAILED* etc.) and printed as result message, e.g. “Evaluation of my return value (Actual value: a) failed.”, where a is the actual value provided by argument `a` of the assertion<sup>32</sup>.
- `RTC_ASSERT_TRUE(n, e)`  
This assertion with name `n` is *PASSED*, if `e == TRUE`. Otherwise the result of the assertion is *FAILED*.
- `RTC_ASSERT_FALSE(n, e)`  
This assertion with name `n` is *PASSED*, if `e == FALSE`. Otherwise the result of the assertion is *FAILED*.
- `RTC_ASSERT_EQUAL(n, e1, e2)`  
This assertion with name `n` is *PASSED*, if `e1 == e2`. Otherwise the result of the assertion is *FAILED*.
- `RTC_ASSERT_NOT_EQUAL(n, e1, e2)`  
This assertion with name `n` is *PASSED*, if `e1 != e2`. Otherwise the result of the assertion is *FAILED*.
- `RTC_ASSERT_PTR_EQUAL(n, e1, e2)`  
This assertion with name `n` is *PASSED*, if pointer `e1` and pointer `e2` are equal (`e1 == e2`). Otherwise the result of the assertion is *FAILED*.

<sup>30</sup> For an example consider e.g. `TestCase SD_tc_0` in

`Sample/CppSamples/TestConductor/CppModelCodeCoverage`. After updating the `TestCase`, the assertion will appear in `StubbedOperation TPkg_Calc::TCon_Calc_Architecture::Display::show(double val)`.

In order to let the assertion fail, add another `'addVal(val=2.0)'` message from the `DummyDriver` to the `SUT` below sending of `'show(val=3.14)'`.

<sup>31</sup> For Rhapsody in C++ only this one assertion macro is sufficient. For Rhapsody in C the similar functionality is provided by a set of macros:

`RTC_ASSERT_SD_NAME_ACTUAL_I(s, m, p, e, a)` – for integer `a`  
`RTC_ASSERT_SD_NAME_ACTUAL_LI(s, m, p, e, a)` – for long `a`  
`RTC_ASSERT_SD_NAME_ACTUAL_LLI(s, m, p, e, a)` – for long long `a`  
`RTC_ASSERT_SD_NAME_ACTUAL_U(s, m, p, e, a)` – for unsigned int `a`  
`RTC_ASSERT_SD_NAME_ACTUAL_LU(s, m, p, e, a)` – for unsigned long `a`  
`RTC_ASSERT_SD_NAME_ACTUAL_LLU(s, m, p, e, a)` – for unsigned long long `a`  
`RTC_ASSERT_SD_NAME_ACTUAL_G(s, m, p, e, a)` – for double `a`  
`RTC_ASSERT_SD_NAME_ACTUAL_LG(s, m, p, e, a)` – for long double `a`  
`RTC_ASSERT_SD_NAME_ACTUAL_S(s, m, p, e, a)` – for actual value `a` to be displayed provided by `char*` (requires explicit serialization in macro call)

<sup>32</sup> For an example consider e.g. `TestCase SD_tc_0` in

`Sample/CppSamples/TestConductor/CppModelCodeCoverage`: modify argument of `show`-message to display, s.t. the `TestCase` will fail. Use e.g. `'show(val=3.145)'` instead of `'show(val=3.14)'`. After updating the `TestCase`, the assertion will appear in `StubOperation TPkg_Calc::TCon_Calc_Architecture::Display::SD_tc_stub_show_1(double val)`

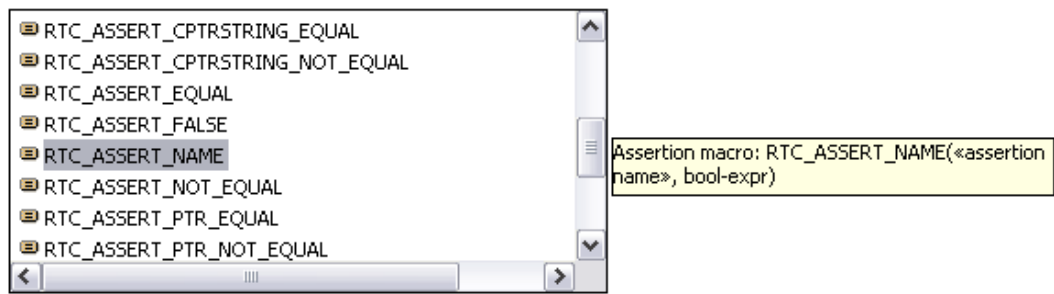
- `RTC_ASSERT_PTR_NOT_EQUAL (n, e1, e2)`  
This assertion with name *n* is *PASSED*, if pointer *e1* and pointer *e2* not equal (*e1*  $\neq$  *e2*). Otherwise the result of the assertion is *FAILED*.
- `RTC_ASSERT_PTR_NULL (n, e1)`  
This assertion with name *n* is *PASSED*, if the pointer *e1* is `NULL`. Otherwise the result of the assertion is *FAILED*.
- `RTC_ASSERT_PTR_NOT_NULL (n, e1)`  
This assertion with name *n* is *PASSED*, if the pointer is not `NULL`. Otherwise the result of the assertion is *FAILED*.
- `RTC_ASSERT_CPTRSTRING_EQUAL (n, e1, e2)`  
This assertion with name *n* is *PASSED*, if the string compare is equal (`strcmp (e1, e2) == 0`). Otherwise the result of the assertion is *FAILED*.
- `RTC_ASSERT_CPTRSTRING_NOT_EQUAL (n, e1, e2)`  
This assertion with name *n* is *PASSED*, if the string compare is not equal (`strcmp (e1, e2) != 0`). Otherwise the result of the assertion is *FAILED*.
- `RTC_ASSERT_STRING_EQUAL (n, e1, e2)`  
This assertion with name *n* is *PASSED*, if the comparison of the strings *e1* and *e2* is equal (*e1*  $==$  *e2*). Otherwise the result of the assertion is *FAILED*.
- `RTC_ASSERT_STRING_NOT_EQUAL (n, e1, e2)`  
This assertion with name *n* is *PASSED*, if the comparison of the strings *e1* and *e2* is not equal (*e1*  $\neq$  *e2*). Otherwise the result of the assertion is *FAILED*.

## Using IntelliVisor for TestConductor Assert Macros

TestConductor supports the usage of the IntelliVisor functionality of Rhapsody. To be able to use this for the defined TestConductor Assert Macros, you have to prepare Rhapsody's `site.prp` file. Please do the following steps:

- Close Rhapsody if it is open.
- Copy the file `rtc.prp` from the `..\TestConductor` folder to the `..\Share\Properties` folder of your Rhapsody installation.
- Open the `site.prp` file and add `Include "rtc.prp"`.
- Save the `site.prp` file and open Rhapsody.

Using `Ctrl+Space` in a code based TestCase definition (Flowchart TestCase or Code TestCase) the known IntelliVisor list box opens. With the modifications above you are able to select one of the defined TestConductor Assert Macros. Selecting one of the macros also shows a hint that gives you information about the parameters of the macro.



*Figure 40: IntelliVisor with TestConductor Assert Macros*

A double-click on the macro adds this to the code. For example you have chosen the `RTC_ASSERT_NAME` macro the following code will be added:

```
RTC_ASSERT_NAME("assertion name", bool-expr);
```

Now you have to replace the string "assertion name" and the expression to that expression you want to check.

# Testing AUTOSAR Models

## Unit testing of AUTOSAR Software Components

(See also TestingCookbook:

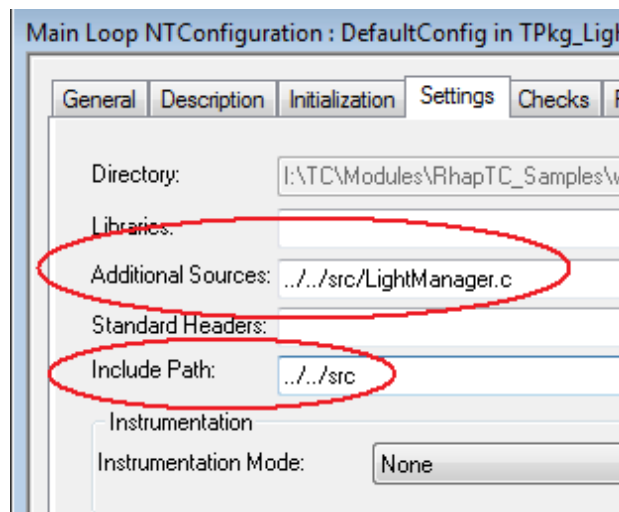
- “How can I test AUTOSAR models?”

)

TestConductor allows unit testing of AUTOSAR Software Components on the host system, supported are AUTOSAR versions 3.1, 3.2 and 4.0. When applying “Create TestArchitecture” on a Software Component, TestConductor automatically creates TestComponents for the ports of the SUT and also a Run Time Environment (Rte) TestComponent. To be able to generate the Rte matching to the SUT, the model should contain a corresponding Internal Behavior for the tested Software Component. Otherwise, it might be needed to manually add functions to the Rte to complete it.

After creating an SD based TestCase, the communication of the SUT can be specified by drawing messages between the SUT and the TestComponents created for the ports. When updating the TestCase, TestConductor adds the implementation of the specified behavior to the Rte TestComponent.

The scope of the code generation component created for testing does not contain the SUT itself. When generating code, the implementation and specification files of the SUT are not generated. Because of this, the path to the implementation file of the SUT has to be entered in the Additional Sources of the code generation component or configuration. Also, the path to the specification file of the SUT needs to be added as Include Path.



*Figure 41: Adding Include Path and Additional Sources*

To compile the tested application, some specification files containing definitions from the AUTOSAR standard are needed. The TestConductor installation contains a set of these files for compilation on a Windows host. These files are located in the folder Share/./TestConductor/AUTOSAR\_RTE. The path to the AUTOSAR specification files



also needs to be added as Include Path of the code generation configuration or component. If further self defined data types are used, definitions of these data types must be added manually, too.

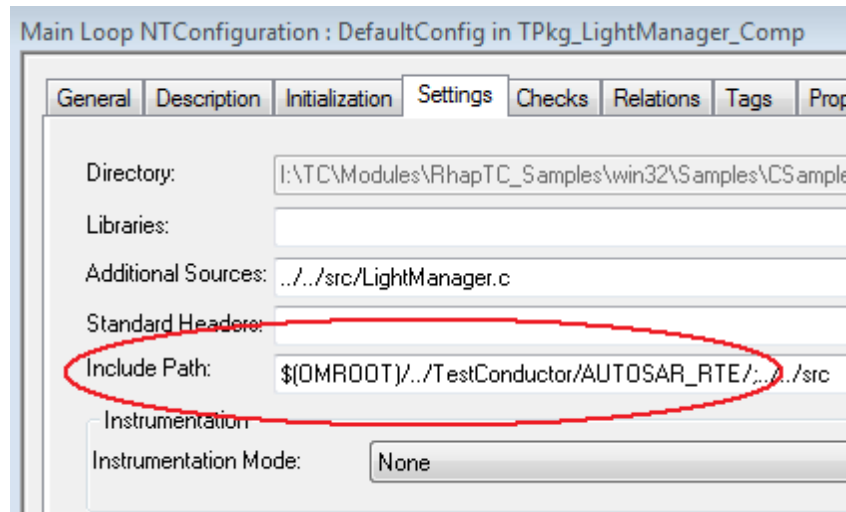


Figure 42: Include Path for AUTOSAR\_RTE

When testing AUTOSAR Software Components, some limitations should be considered:

- For AUTOSAR, animation is not available so TestConductor cannot compute model coverage.
- Computation of code coverage is not supported.
- Testing of AUTOSAR Software Components is supported only for AssertionBased testing mode.

As an example of how AUTOSAR Software Components can be tested with TestConductor, please have a look at the sample “LightsManager” in the folder “Samples/CSamples/TestConductor/TestingCookbook” (AUTOSAR 3.1).

As mentioned before, TestConductor automatically creates a Rte during TestArchitecture creation. There are some differences depending on the AUTOSAR version.

### AUTOSAR 3.1/3.2 Rte

The following Rte operations are currently created automatically by TestConductor for AUTOSAR version 3.1/3.2:

- Std\_ReturnType  
**Rte\_Send** <port>\_<dataElementPrototype>(In data)  
 If the SUT has a senderPort, such an operation is created for each DataElementPrototype of the port interface if the tag isQueued of the DataElementPrototype has the value true.
- Std\_ReturnType  
**Rte\_Write** <port>\_<dataElementPrototype>(In data)  
 If the SUT has a senderPort, such an operation is created for each DataElementPrototype of the port interface if the tag isQueued of the DataElementPrototype has the value false.

- Std\_ReturnType  
**Rte\_Receive\_**<port>\_<dataElementPrototype>(Out data)  
  
If the SUT has a receiverPort, such an operation is created for each DataElementPrototype of the port interface if the tag isQueued of the DataElementPrototype has the value true.
- Std\_ReturnType  
**Rte\_Read\_**<port>\_<dataElementPrototype>(Out data)  
  
If the SUT has a receiverPort, such an operation is created for each DataElementPrototype of the port interface if the tag isQueued of the DataElementPrototype has the value false.
- Std\_ReturnType **Rte\_Call\_**<port>\_<operation>(param\_1, ..., param\_n)  
  
If the SUT has a clientPort, such an operation is created for each OperationPrototype of the port interface.
- <return> **Rte\_IRead\_**<runnableEntity>\_<port>\_<dataElem>()  
  
If a RunnableEntity has DataReadAccess referring to a DataElementPrototype such an operation is created.
- void **Rte\_IWrite\_**<runnableEntity>\_<port>\_<dataElem>(In data)  
  
If a RunnableEntity has DataWriteAccess referring to a DataElementPrototype such an operation is created.
- <return>  
**Rte\_IrvRead\_**<runnableEntity>\_<interRunnableVar>()  
  
If a RunnableEntity is referring to a read InterRunnableVariable and the tag communicationApproach of the variable has the value explicit, such an operation is created.
- void **Rte\_IrvWrite\_**<runnableEntity>\_<interRunnableVar>(In data)  
  
If a RunnableEntity is referring to a written InterRunnableVariable and the tag communicationApproach of the variable has the value explicit, such an operation is created.
- <return>  
**Rte\_IrvIRead\_**<runnableEntity>\_<interRunnableVar>()  
  
If a RunnableEntity is referring to a read InterRunnableVariable and the tag communicationApproach of the variable has the value implicit, such an operation is created.
- void **Rte\_IrvIWrite\_**<runnableEntity>\_<interRunnableVar>(In data)

If a RunnableEntity is referring to a written InterRunnableVariable and the tag communicationApproach of the variable has the value implicit, such an operation is created.

- `void Rte_Enter_<exclusiveArea>()`

Such an operation is created for each ExclusiveArea.

- `void Rte_Exit_<exclusiveArea>()`

Such an operation is created for each ExclusiveArea.

The following types of Rte operations are currently not created by TestConductor:

- Rte\_Ports
- Rte\_NPorts
- Rte\_Port
- Rte\_Switch
- Rte\_Invalidate
- Rte\_Feedback
- Rte\_Result
- Rte\_Pim
- Rte\_CData
- Rte\_Calprm
- Rte\_IWriteRef
- Rte\_IInvalidate
- Rte\_IStatus
- Rte\_Mode

If a SUT operation should be called periodically during TestCase execution, a TimingEvent has to be added to the InternalBehavior. This TimingEvent must refer to a RunnableEntity and the tag symbol of this RunnableEntity must be set to define the name of the operation to be called periodically. How often the operation has to be called is defined by the tag period of the TimingEvent.

## **AUTOSAR 4.0 Rte**

The following Rte operations are currently created automatically by TestConductor for AUTOSAR version 4.0.

- `Std_ReturnType Rte_Send_<port>_<dataElement>(In data)`

If the SUT has a dataSenderPort, such an operation is created for each DataElement of the port interface which has event semantics (the tag swImplPolicy has the value queued for a SwDataDefPropsVariant of a SwDataDefProps defined for the DataElement).

- Std\_ReturnType **Rte\_Write\_**<port>\_<dataElement>(In data)

If the SUT has a dataSenderPort, such an operation is created for each DataElement of the port interface which does not have event semantics.

- Std\_ReturnType **Rte\_Receive\_**<port>\_<dataElement>(Out data)

If the SUT has a dataReceiverPort, such an operation is created for each DataElement of the port interface which has event semantics (the tag swImplPolicy has the value queued for a SwDataDefPropsVariant of a SwDataDefProps defined for the DataElement).

- Std\_ReturnType **Rte\_Read\_**<port>\_<dataElement>(Out data)

If the SUT has a dataReceiverPort, such an operation is created for each DataElement of the port interface which does not have event semantics.

- Std\_ReturnType **Rte\_Call\_**<port>\_<operation>(param\_1, ..., param\_n)

If the SUT has a clientPort, such an operation is created for each ClientServerOperation of the port interface.

- <return> **Rte\_DRead\_**<port>\_<dataElement>()

If a RunnableEntity is referring to a dataElement with data semantics in the dataReceivePointByValue role, such an operation is created.

- <return> **Rte\_IRead\_**<runnableEntity>\_<port>\_<dataElem>()

If a RunnableEntity is referring to a dataElement via dataReadAccess, such an operation is created.

- void **Rte\_IWrite\_**<runnableEntity>\_<port>\_<dataElem>(In data)

If a RunnableEntity is referring to a dataElement via dataWriteAccess, such an operation is created.

- <return>  
**Rte\_IrvRead\_**<runnableEntity>\_<interRunnableVar>()

If a RunnableEntity is referring to an explicitInterRunnableVariable as readLocalVariable, such an operation is created.

- void **Rte\_IrvWrite\_**<runnableEntity>\_<interRunnableVar>(In data)

If a RunnableEntity is referring to an explicitInterRunnableVariable as writtenLocalVariable, such an operation is created.

- <return>  
**Rte\_IrvIRead\_**<runnableEntity>\_<interRunnableVar>()

If a RunnableEntity is referring to an ImplicitInterRunnableVariable as readLocalVariable, such an operation is created.

- `void Rte_IrvIWrite_<runnableEntity>_<interRunnableVar>`  
(In data)

If a RunnableEntity is referring to an ImplicitInterRunnableVariable as writtenLocalVariable, such an operation is created.

- `void Rte_Enter_<exclusiveArea>()`

Such an operation is created for each ExclusiveArea.

- `void Rte_Exit_<exclusiveArea>()`

Such an operation is created for each ExclusiveArea.

The following types of Rte operations are currently not created by TestConductor:

- Rte\_Ports
- Rte\_NPorts
- Rte\_Port
- Rte\_Switch
- Rte\_Invalidate
- Rte\_Feedback
- Rte\_SwitchAck
- Rte\_Result
- Rte\_Pim
- Rte\_CData
- Rte\_Prm
- Rte\_IWriteRef
- Rte\_IInvalidate
- Rte\_IStatus
- Rte\_Mode
- Rte\_Trigger
- Rte\_IrTrigger
- Rte\_IFeedback
- Rte\_IsUpdated

If a SUT operation should be called periodically during TestCase execution, a TimingEvent has to be added to the InternalBehavior. This TimingEvent must refer to a RunnableEntity and the tag symbol of this RunnableEntity must be set to define the name of the operation to be called periodically. How often the operation has to be called is defined by the tag period of the TimingEvent.

## Migrating animation based TestArchitecture to assertion based TestArchitecture

There are several differences between an assertion based and an animation based TestArchitecture, so an animation based TestArchitecture cannot be converted into an animation based TestArchitecture just by changing the property “TestConductor.Settings.TestingMode”. Instead, it is recommended to create a new TestArchitecture and to create new TestCases based on the original ones.

To manually migrate an animation based into an assertion based TestArchitecture, the following approach should be applied:

- Make sure the project property “TestConductor.Settings.TestingMode” is set to “AssertionBased”.
- Create a new TestArchitecture for the class, file or object which was tested by the animation based TestArchitecture.
- Migrate the TestCases of the original TestArchitecture one after another. For the different kinds of TestCases, the following migration steps should be applied:
  - Code based TestCases  
A code based TestCase can be copied to the new assertion based TestArchitecture. It is recommended to inspect the code of the TestCase and check for references of TestComponents which might have a different name.
  - Flowchart based TestCases  
A flowchart based TestCase can be copied to the new assertion based TestArchitecture. It is recommended to inspect the code of the TestCase and check for references of TestComponents which might have a different name.
  - Statechart based TestCase  
A statechart based TestCase should be migrated this way:
    - Create a new TestCase by applying the helper “Create Statechart TestCase” on the new TestContext.
    - Select all elements in the new statechart and delete them
    - Open the statechart of the original TestCase
    - Select all elements in the old statechart and copy them into the new statechart
    - Adjust the first transition in the statechart (from state “Initial” to state “state\_1”):  
For language C++: Select “evTCStart” from the new TestPackage as the trigger of the transition and remove the line “itsTCon->rtc\_init()” from the Action of the transition.  
For language C: Select “evTCStart” from the new TestPackage as the trigger of the transition and remove the line  
“TCon\_<name>\_rtc\_init(me->itsTCon)” from the Action of the transition.

- Adjust the last transition in the statechart (from state “final” to the termination state):  
 For language C++: In the Action of the transition, change line “itsTCon->rtc\_exit()” to “itsTCon->finishTestCase()”.  
 For language C: In the Action of the transition, change line “TCon\_<name>\_rtc\_exit(me->itsTCon)” to “Tcon\_<name>\_finishTestCase(me->itsTCon)”.
- Sequence diagram based TestCase  
 A sequence diagram based TestCase should be migrated this way:
  - If the old and the new TestArchitecture have similar TestComponents, the TestCase wizard can be used to create a new TestCased based on the TestScenario of the old TestCase. To do this, right click the original TestScenario and select “Create TestCase...”. In the dialog, the destination TestContext can be selected: If the new TestContext of the assertion based TestArchitecture is listed, select the new TestContext and confirm the creation of a new TestCase by clicking the Ok button. The wizard will create a new TestCase in the animation based TestArchitecture, based on the original TestScenario.
  - If the wizard cannot match the TestComponentInstances of the animation based TestArchitectures with the TestComponentInstances of the assertion based TestArchitecture, the sequence diagram based TestCases need to be migrated manually. To do so, create a new new TestCase by applying the helper “Create SD TestCase” on the new TestContext. Then add the messages of the original TestScenario one after another.

## Automatic Migration of animation based TestArchitectures to assertion based Testing mode

When updating a TestContext of an animation based TestArchitecture, TestConductor checks for applicability of automatic migration to assertion based testing mode. Automatic migration is applicable to animation based TestArchitecture whose SUT is only connected to TestComponents via ports or whose SUT only has instantiated associations to interfaces.

If the TestArchitecture fulfills these applicability criteria, automatic migration is offered to the user in a dialog. If the user confirms the attempt of migration, a new TestArchitecture is created from a copy of the animation based architecture. A report of the migration steps – including warnings and potential problems – is issued on the console and stored additionally in a comment below the newly created TestContext. After application of migration or if the user doesn't confirm the attempt to migration, property TestConductor.TestContext.MigrateToAssertionBasedMode (with value 'False', unchecked boolean property) is added to the TestContext of the animation based old TestArchitecture. Automatic migration isn't offered to the user for this TestContext again unless property TestConductor.TestContext.MigrateToAssertionBasedMode is checked, i.e.set to 'True'.

In particular SD TestCases may be affected by several limitations of the assertion based TestingMode:

- assertion based execution only supports linearly ordered SDInstances.

- assertion based execution only supports 'driving and monitoring' SDInstances.
- assertion based execution only supports SDTestCases with single SDInstances.
- multiple iteration of SDInstances isn't supported in assertion based execution.
- ordered predecessors aren't supported by assertion based execution.

Potential problems are reported on the console and these migration messages are also recorded in a comment that is stored below the TestContext in the new TestArchitecture obtained by automatic migration. Note, that most TestConductor.TestCase properties aren't regarded in assertion based execution.

## Functional Limitations

- TestConductor cannot generate stubs, if the signature of overwritten operations in an inheritance hierarchy do not syntactically match to the related operation in the base class (for instance, due to different typedef-types to the same base type)
- The auto-generated code for driver- or stub-operations could be semantically incorrect, if non-default values for the properties `CPP_CG:: {Class, Type}:: {In, Out, InOut}` are used. Note that incorrectly generated code could be overwritten by setting the tag `RTC_DriverCallCode`, `RTCDriverInitCode` respectively `RTC_StubBodyCode`.
- If a TestComponentInstance is linked to a SUT using a qualified association relation, Rhapsody does not generate code to implement the link. TestConductor can not generate driver operations for messages, which use such a link.
- Auto created operations are not animated and cannot be used in TestCases: due to a limitation in the Rhapsody animation, auto generated operations like getter/setter for class attributes are not animated during execution, they do not appear in animated sequence diagrams and observers don't get notifications about these messages (even if the property `CG:CGGeneral:GeneratedCodeInBrowser` is set to `true`).



## List of Figures

Rhapsody Testing Profile and TestConductor.....	14
TestArchitecture in Rhapsody Browser.....	29
Advanced TestArchitecture Creation Dialog.....	30
TestCase Scheduler Statechart.....	31
Arbiter of SD TestCase.....	32
SD TestCases - Stubbing and Observing Operations.....	51
SD TestCase mapping using TestCase Wizard.....	54
Test Result Report with Result Verification Information.....	69
Test Execution Dialog for Code-, Flowchart- and Statechart TestCases.....	70
Test Execution Dialog for SD TestCases.....	72
Unmodified TestCase Scheduler Statechart.....	74
Introducing TestCase Timeout Transition.....	75
Debugging Button in Test Execution Window.....	76
Test Execution Window for TestContext Execution.....	77
Ordering of TestCases.....	78
Transitivity of Dependencies (Refinement of model elements and requirements).....	84
TestConductor Main Dialog.....	96
Properties - TestConductor.....	97
Properties TestConductor.Settings.....	98
Setting TestingMode.....	101
Properties TestConductor.TestContext.....	101
Example relations with inheritance.....	111
Select Message with virtually inherited operations.....	111
Select Message and Inheritance.....	112
Interpretation of Messages by TestConductor.....	119
Tags of Stereotype <<RTC_MsgInfo>>.....	120
Using TestActions.....	124
Assertion generated from <PostCallAction> TestAction.....	125
<precond> Condition (HarmonySE).....	127
<check> Condition.....	127
TestCondition.....	128
{Lang}_CG.Type.SerializationAndUnserialization.....	131
Using Serialization.....	131
TestScenario with InteractionOccurrence.....	135
Invocation of 'Show As SD'.....	135
Witnesses for referencing and referenced TestScenario.....	136
Merging ModelCoverage Results.....	138
Merging CodeCoverage Results.....	139
RQM Connection.....	140
IntelliVisor with TestConductor Assert Macros.....	159
Adding Include Path and Additional Sources.....	160
Include Path for AUTOSAR_RTE.....	161