Rhapsody®

IBM Engineering Systems Design Rhapsody - TestConductor Add On

TestConductor User Guide

Release 2.9.2

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® Engineering Systems Design Rhapsody®, IBM® Engineering Systems Design Rhapsody® - Automatic Test Generation Add On, and IBM® Engineering Systems Design Rhapsody® - 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 2000-2025 BTC Embedded Systems AG. All rights reserved.

Contents

Content

About this document

This document is part of the documentation of the IBM Engineering Systems Design Rhapsody - 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
- ETMTestConductorAdapter_HowTo.pdf
  Small document describing the TestConductor Adapter to IBM Engineering Test Management and how to use the adapter.
- RTC_Release_Notes.pdf
  TestConductor Release Notes.
- legacy/RTC_User_Guide.pdf
  TestConductor User Guide for 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 16).

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, objects and Files on page 41 and Grey Box Testing on page 45), 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 23). 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 ba 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 23), 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[1].
  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 52), 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 23 ff and Using Objects on page 40).

  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

[1]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® Engineering Systems Design 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.

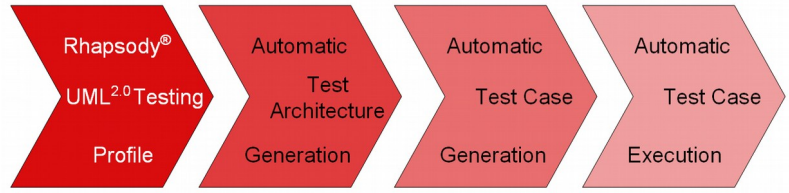


Figure 1: Rhapsody Testing Profile and TestConductor

TestConductor supports the two main features “Automatic TestArchitecture Generation” and “Automatic TestCase Execution” of the Rhapsody Testing Environment. The optional IBM® Engineering Systems Design 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 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, Rhapsody in C++ and Rhapsody in Java only. For Rhapsody in Ada, please have a look into the document RTC_User_Guide.pdf in sub folder legacy.

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 180 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 resusing 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 20 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 can not be instantiated as parts of a TestContext. Such model elements have to be instantiated as global objects.
For files in Rhapsody in C, TestConductor supports TestFiles in TestArchitectures. TestFiles can also be used in TestArchitectures using parts. There is no need to use global objects mode when working with files.

TestConductor supports BlackBox- and GreyBox testing (cf. sections Black Box Testing and Grey Box Testing on page 45 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 test application of 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 test execution results as reports which are maintained as part of the model. Besides an execution report (see section Execution Results on page 79), also reports for various coverage measures are generated and added to the model:

- Model coverage (cf. section Computing Model Coverage during Test Execution on page 99)
- Requirement coverage (cf. section Computing Requirement Coverage on page 102)
- Code coverage for Rhapsody in C++ and Rhapsody in C (cf. section Computing Code Coverage on page 109).

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 180 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 (*.sbsx)’
- Select in Rhapsody installation folder: ‘`...\Share\Profiles\TestingProfile\TestingProfile_rpy\TestingProfile.sbsx`’
- 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[2] 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

[2]For Rhapsody in Java, stereotype `<<AssertionBasedTestingConfiguration>>` is used instead of `<<TestingConfiguration>>`. In fact, `<<AssertionBasedTestingConfiguration>>` defines the core properties for assertion based testing. For Rhapsody in C++ and Rhapsody in C, the specialization `<<TestingConfiguration>>` extends these core properties and add specific capabilities. In turn, `<<TestingConfiguration>>` is extended by `<<TargetTestingConfiguration>>` and possible further specializations, adding dedicated properties for testing on particular targets.

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, stereotypes <<AssertionBasedTestingConfiguration>>, <<TestingConfiguration>> or <<TargetTestingConfiguration>>, respectively, are used to distinguish standard configurations from TestingConfigurations which are adjusted to the special needs of the TestConductor TestArchitecture. A <<AssertionBasedTestingConfiguration>>, <<TestingConfiguration>> and <<TargetTestingConfiguration>>, respectively, has additional tags for configuring additional features (like coverage measurement) or fine-tuning the test execution (e.g. rtc_log_kind to define the way 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[3].

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

[3]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: Blocks,)
- 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 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, etc)
  - socket connection between tested application and Rhapsody application.
- assertion based testing mode (applicable to C, C++ and Java models only, not available for 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 23) 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).
'Wrapper' denotes a full copy of the original class (a clone of the replaced element in the TestArchitecture), whereas 'Stub' denotes an abstracted replacement, i.e. parts of the behavior of the original element, which are not necessary for testing purposes have been removed from the replacement.

TestArchitecture generation can be customized interactively using property `TestConductor::Settings::CreateTestArchitectureMode` (cf TestConductor settings "General Properties", page 126).

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 not 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 126), 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 C 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 126), 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.

SUT and TestComponents

TestArchitectures mainly consist of the System Under Test (SUT) and TestComponents (cf. chapter Preliminary Note on page 9). The SUT is treated 'as is' by TestConductor, i.e. TestConductor must not modify the model elements belonging to te SUT or the generated code for testing purposes – with limited exceptions for GreyBox testing (see GreyBox TestArchitectures for classes, objects and Files on page 41 and Grey Box Testing on page 45), testing of private operations (see TestConductor Support for Testing Private Operations in Rhapsody in C and TestConductor Support for Testing Private and Protected Operations in Rhapsody in C++ on page 53 ff.) and Code Coverage measurement instrumentation (see Computing Code Coverage on page 109 ff).
All model elements in a TestArchitecture that don't belong to the SUT can be modified by TestConductor or the user for testing purposes. In order to enable testing related modifications to the model elements belonging to the test environment of a SUT, TestArchitectures provide TestComponents for all relevant relations of SUT model elements. TestComponents are created for the following relations of the SUT model elements:

- for each outgoing directed or bidirectional association with a class or SysML block either an inheriting or replacement TestComponent will be created[4].
- for each outgoing directed or bidirectional association with an interface an inheriting TestComponent will be created.
- for each outgoing directed or bidirectional association with a VariationPoint a TestComponent Variant will be created and used for the VariationPoint.
- for each outgoing directed or bidirectional association with an implicit object a replacement TestComponentObject will be created[5].

[4] whether inheritance can be used or not, depends on the operations of the original target of the association:
- if all operation are virtual, an inheriting TestComponent can be used,
- if at least one of the operations is abstract, an inheriting TestComponent must be used,

if at least one of the operations is non virtual, a replacement TestComponent must be used.
For Rhapsody in C, inheritance is in general only used for associations with interfaces.
[5] only if 'Architecture using global objects' has been chosen, otherwise no TestComponent can be created.

- for each port, flow port or proxy port on the top level of SUT classes or SysML blocks a TestComponent providing the reverse port, flow port or proxy port will be created.
- no TestComponents will be created for ingoing directed associations, since these TestComponents could only be used for providing the SUT with stimuli. Driving the SUT with stimuli can be done using the TestContext itself or a dedicated DummyDriver TestComponent.
- no TestComponents will be created for associations with aggregation kind 'Shared' or 'Composition' set on one association end.Accoding to the UML specification AggregationKind=shared or composite expresses a kind of ownership[6]: While AggregationKind=composite for a SUT association with a particular class clearly has to be interpreted as the associated class being part of the SUT, the interpretation of AggregationKind=shared may vary by application and modeler. TestConductor adheres to the interpretation of the associated class being owned by the SUT, i.e. the SUT is responsible for creation and destruction of instances and hence no instances of the associated class have to be provided or manipulated by the TestContext.
- for Rhapsody in C, a TestComponent file (or TestFile) will be created for <<Usage>> dependencies on files.

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.
For Rhapsody in Java, inheritance can not be applied only to final classes or classes with final operations. In all other cases, inheritance is applicable.

[6] „Sometimes a Property is used to model circumstances in which one instance is used to group together a set of instances; this is called aggregation. To represent such circumstances, a Property has an aggregation property, of type AggregationKind; the instance representing the whole group is classified by the owner of the Property, and the instances representing the grouped individuals are classified by the type of the Property. AggregationKind is an enumeration with the following literal values:

- none – Indicates that the Property has no aggregation semantics.
- shared – Indicates that the Property has shared aggregation semantics. Precise semantics of shared aggregation varies by application area and modeler.
- composite – Indicates that the Property is aggregated compositely, i.e., the composite object has responsibility for the existence and storage of the composed objects

Composite aggregation is a strong form of aggregation that requires a part object be included in at most one composite object at a time. If a composite object is deleted, all of its part instances that are objects are deleted with it.” (Source: Unified Modeling Language v2.5.1, https://www.omg.org/spec/UML/2.5.1/)

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[7]. 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`.

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.

Creation of replacements has to take packages and namespaces into account. If properties {CPP,C,JAVA}`_CG::Package::GenerateDirectory` or {CPP,JAVA}`_CG::Package::DefineNameSpace` are set on some package in the declaration or instantiation path of a class or object, replacement creation for that class or object has to establish the same package hierarchy as for the original decalaration or instantiation. For C++ and C, the TestPackages of the TestArchitecture itself, do not generate separate directories or define namespaces. Hence, package hierarchies for replacement declarations or instantiations can be provided in the TestArchitecture TestPackage hierarchy.
For Java, generation of directories and definition of namespaces for packages is almost mandatory. Since the TestArchitecture TestPackages define an own package and namespace hierarchy, replacements for design classes and objects have to be provided in an own TestPackage hierarchy. For each TestArchitecture, there exists a separate replacements TestPackage on the toplevel of the model hierearchy. This replacements TestPackage is associated to the TestContext using a navigable `<<Associated ReplacementsPackage>>` dependency.

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

[7]Except for replacements of files (modules). For TestFile – replacement of file – the TestScenarios refer to the TestFile instead of the replaced file.

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).

- Ports classes and objects (affects only classes and objects with ports in RiC – in particular composite GreyBox SUT replacements): In contrast to ports in RiC++, ports cannot be represented as nested classes in RiC. Hence, additional files are generated for the contracts of ports – port contracts are not declared in the same source file as the port owning class or object.
  Since, by construction of replacements, the original model element with its ports does not belong to the scope (whereas the replacement is in code generation scope – it 'replaces the original model element), but instances in the TestArchitecture refer to the original model element, Rhapsody code generation omits includes of these additional port contract files in the files generated for replacements.
  To avoid compiler warnings and errors, it may become necessary to add appropriate entries to the C_CG::Class::ImpInclude property of the replacement. The names of the required headers can be found by comparison of the source code files of original model element (for some original model code generation configuration) and its replacement (for the code generation configuration belonging to the TestArchitecture under consideration).

- Inline operations (C++ only[8]): Replacements are used to be instrumented for testing purposes instead of instrumenting original model elements, Testing instrumentation refers to artefacts in the TestArchitecture such as Arbiter TestComponents, TestContext etc. If necessary, appropriate <<Usage>> dependencies are added to replacements.
  The header-file of the TestContext includes the class definition, i.e. header file, of the replacement, (at least if the replacement is embeddable). The code of an instrumented function body in turn needs the definition, i.e. header, of the TestContext. In general, definitions of inline operations are generated into the respective class header file. Thus, an instrumented inline function in the class header file causes an include-cycle, which leads to a compile error.
  Therefore, TestConductor unsets the inline flag of replacement operations which

[8]Rhapsody in C does by default not support inlining in code generation. Code generation generates a #define macro for inline operations, i.e. operations for which the `inline` check-box in the features dialog was checked, or property `C_CG::Operation::Inline` was set to `in_header`, respectively. Due to macro generation such operations must not contain return statements. Thus, TestConductor can not simply treat such 'inlining' macros as normal operations, without causing compile-errors - in general inline operations are, hence, not supported for Rhapsody in C
For code generation, one can make use of `C_CG::WriterTemplates::OperationAsMacro` to overrule generation as define for such operations
([[$description]][[$requirements]][[$annotation]]$prolog inline $name($arguments) { [[$indentation$synthesized_prefix]][[$indentation$body]][[$indentation$synthesized_suffix]] } $epilog.) In this case, the same treatment as for C++ can be applied successfully. Note, that TestConductor is not able to regard and interpret WriterTemplates. There is no check for the contents of the named property.

are declared inline in the original model element.
Note, that this does not affect the behavior of the caller since inline is only a compiler optimization hint, without affecting the signature or semantics of the function (if the original function works as well in the scope of the caller and in the scope of the callee).

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 `TestComponentInstance itsA` is `A`	`TestComponentInstance itsA` has `<<use_replacement>>` dependency on replacement `TestComponent A'`	Replacement `TestComponent A'` has `<<replacement>>` dependency on class `A`
`TestSUT` replacement of class `A` (*greybox testing*)	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`
`TestSUTFile` Replacement of file Y	type of TestSUTFile Y'is implicit	`TestContext` has `<<use_greyboxfilereplacement>>` dependency on `TestFile X'`	Replacement `TestSUTFile Y'` has `<<greyboxfilereplacement>>` dependency on file `Y`
`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 `<<instancereplacment>>` dependency on implicit object `O`
`TestComponentObject` Instance of design class	type of `TestComponentObject O` is *inheriting* TestComponent Class	TestContext has `<<instantiated>>` association end to inheriting TestComponent class of `O`. Association end has `<<usedTestComponentObject>>` dependency on	`TestComponentObject O` has no further dependencies

		O	
TestComponentObject Instance of explicit replacement TestComponent class	type of `TestComponentObject O` is replaced class in user design.	TestContext has `<<instantiated>>` association end to replaced TestComponent class of `O`. Association end has `<<usedTestComponentObject>>` dependency on `O`	`TestComponentObject O` has `<<use_replacement>>` dependency on replacement TestComponent.
`TestSUTObject` replacement of implicit object `O` (*greybox testing*)	type of `TestSUTObject O'` is implicit	`TestContext` has `<<use_greyboxinstancereplacement>>` dependency on `TestSUTObject O'`	`TestSUTObject O'` has `<<greyboxinstancereplacement>>` dependency on implicit object `O`
`TestSUTObject` Instance of explicit design class (blackbox testing)	type of `TestSUTObject O` is *design* Class	TestContext has `<<instantiated>>` association end to inheriting TestComponent class of `O`. Association end has `<<usedSUTObject>>` dependency on O	`TestSUTObject O` has no further dependencies
TestSUTObject Instance of explicit design class (greybox testing)	type of `TestSUTObject O` is replaced class in user design.	TestContext has `<<instantiated>>` association end to replaced design class of `O`. Association end has `<<usedSUTObject>>` dependency on `O`	`TestSUTObject O` has `<<use_greyboxreplacement>>` dependency on replacement TestComponent.
`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` (*greybox testing*)	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
`TestSUTFile` Replacement of file Y	type of TestSUTFile Y'is implicit	`TestContext` has `<<use_greyboxfilereplacement>>` dependency on `TestFile X'`	Replacement `TestSUTFile Y'` has `<<greyboxfilereplacement>>` dependency on file `Y`
`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++ and Java 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 61 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++ and Java, 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 fully 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.
For the special case of classes inheriting from templates see section Special Case: Class inheriting from Template on page 38.

Multiplicity of Relations

UML and SysML allow for specification of multiplicity of relations in various places. E.g. for objects, association ends or ports it can be specified, that some number of instances of the model element will be used in the design. Besides concrete constant numbers like ‘1’ or ‘5’, also symbolic notations such as

- ‘*’ (arbitrary many) or
- ranges like ‘3..*’ (at least 3 instances – up to arbitrary many) or
- names of constant expression are supported in Rhapsody like ‘MAX_N’, where MAX_N is the name of a define or constant expression somewhere in the scope of the code generation.
  Note, that it is sufficient for *code generation*, that the constant value of the symbolic parameter is known at compile time: the parameter can be defined as constant in the design, can be defined as *language* type definition or even as a define in some external header file. TestConductor as a model based testing tool suffers from such symbolic multiplicity specifications. For interpreting the

multiplicity specifications correctly, it is not sufficient that their values are known at compile-time. TestConductor needs more explicit information in order to analyze the end to end connections among instances in the TestArchitecture.

Multiplicity specifications are taken into account by TestArchitecture creation.
If a relation of the SUT is defined with non default multiplicity (default: '1'), then TestArchitecture creation aims at instantiating test artifacts in an appropriate number to satisfy the multiplicity of the relation. E.g. TestArchitecture creation will create 'MAX_N' TestComponentInstances for an outgoing association with multiplicity 'MAX_N' of the SUT. Unfortunately, symbolic constant names specifying multiplicities are not resolved in the Rhapsody meta model, i.e. TestConductor has no interpretation for the concrete value of such a symbolic constant. While expressions like e.g. '3..*' are partly interpreted by TestConductor, the concrete value of names like 'MAX_N' can not be inferred by TestConductor.
In particular for driver- and stub-operation generation, TestConductor needs to know the concrete values of multiplicity specifications to identify the effective senders or receivers of messages in TestCases by analyzing the links instantiating relations in the TestArchitecture.

To support usage of symbolic multiplicity parameters in TestArchitectures, the values of such parameters can be provided as <<MultiplicityConstantMap>> in a TestArchitecture (cf. section Multiplicity of Relations on page 59 and section TestArchitecture Package on page 183 ff).

Scope Selection and Replacements – Design for Testability

The scope selection tab of the features dialog for a code generation component permits individual selection and deselection of packages, classes, implicit objects and files. Individual events, types, functions, instances of classes, links and so on can not be added or removed from the scope. If a package belongs to the scope, then in general all its contents automatically belongs to the scope, except for those model elements, which were explicitly removed from the scope. In particular, if some events, types, classes etc. of one package are needed in the scope, then automatically also all objects (of explicit classes) defined in that package belong to the scope. Since use of replacements requires a fine grained scope selection – the original model element has to be deselected from the scope, whereas the replacing mode element has to be added to the scope instead – TestArchitecture creation and adaption may suffer from packages containing a mixture of objects, events, types, links, classes etc.
Undesired global objects, their relations and their initialization may interfere with TestArchitecture instances and testing artifacts during test execution, in particular w.r.t. model and code coverage.

To be able to eliminate objects and links from the scope of a TestArchitecture, it is recommended to already separate instantiations of classes together with links instantiating relations among the instances from classes in different packages[9]. If e.g. already all objects and links among them are put in dedicated sub-packages during modeling, then getting rid of these objects for TestArchitecture code generation can simply be managed by deselecting these sub-packages from the scope.

Another source of troubles w.r.t. TestArchitecture creation is the use of property {Lang}_CG::Package::GenerateDirectory. If this property is checked, Rhapsody generates

[9]We strictly recommend to separate events, interfaces, type-declarations, instances and links among these and classes in separate packages. A clear separation of model elements according to their purpose as well as according to their functional usage and relations, will significantly ease the determination of code generation scopes for testing and for reuse of model elements in different contexts.

sub-directories in the code generation directory for the respective packages and generates the code for the package contents to these sub-directories. To deal with this, TestConductor has to organize replacements in an adequate ‘emulating’ TestPackage structure in the TestArchitecture packages. If now some of the elements from an original model package belong to the scope and some other elements are replaced by elements from the ‘emulating’ TestPackage in the TestArchitecture, then code generation will have to merge the generated code of two different packages into the same generated files. Although automatic directory organisation of generated source code is a powerful tool, it does not work properly with the techniques applied by TestConductor for testing purposes and can complicate TestArchitecture creation and adaption.

Similar problems are caused also by property {Lang}_CG::Package::DefineNameSpace in Rhapsody in C++ and Java.

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 124 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 TestProject’

When the TestingProfile is loaded, TestConductor offers a helper to create a separate test project for the active project. Such a test project is a separate Rhapsody project, that after its creation contains the complete content of the actual project as referenced units. References to not needed content of the design model can be removed from the test project after its initial creation.

When this helper is invoked on the project, a first dialog prompts for a name of the test project to be created. After entering a name and confirming using ‘OK’, a file selector dialog will open, that offers definition of the file system location, where the test project will be created.

Working with a separate test project, separates testing activities from modeling activities. Since all content of the original model is included as read-only reference, testing activities

in the test project can not affect the design model.
Changes in the design model will affect the test project on saving, since the units of the design model are included as referenced units in the test project.

Context Menu 'Create TestArchitecture'

By default, context menu 'Create TestArchitecure' 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.

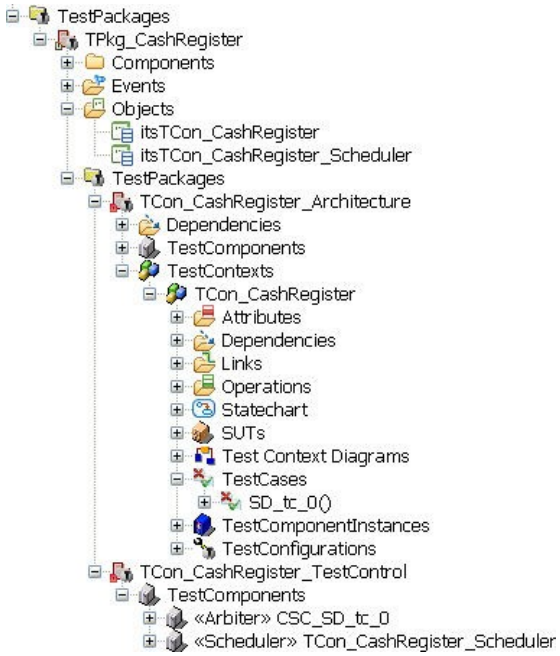


Figure 2: TestArchitecture in Rhapsody Browser

For Rhapsody in Java also a second toplevel TestPackage (parallel to the `TPkg_`<sutname> TestPackage) for replacement classes and objects is created. Since properties `JAVA_CG::Package::GeneratDirectory` and `JAVA_CG::Package::DefineNameSpace` are both set for packages by default, replacements almost always have to be created within a TestPackage hierarchy defining a directory and namespaces-path that is similar to the original path of the model element. In contrast to Rhapsody in C and C++, Java code generation requires also the objects of TestContext and TestCaseScheduler to be

instantiated in a dedicated directory and namespace. Therefore, model element replacements, such as greybox SUT or TestComponent replacements can not be placed in the TestArchitecture as for Rhapsody in C++ and C, but require a separate toplevel TestPackage to form an appropriate TestPackage path for their declaration and instantiation.

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>>` for C and C++, `<<AssertionBasedTestingConfiguration>>` for Java. The respective stereotype provides several tags that can be used to define several testing options (cf. section Tags of the <<AssertionBasedTestingConfiguration>> and <<TestingConfiguration>> Stereotypes on page 69).

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++ and Java, 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[10], 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.

[10]for Java i.e. if at least one operation of a class is final or if the entire class is marked as final.

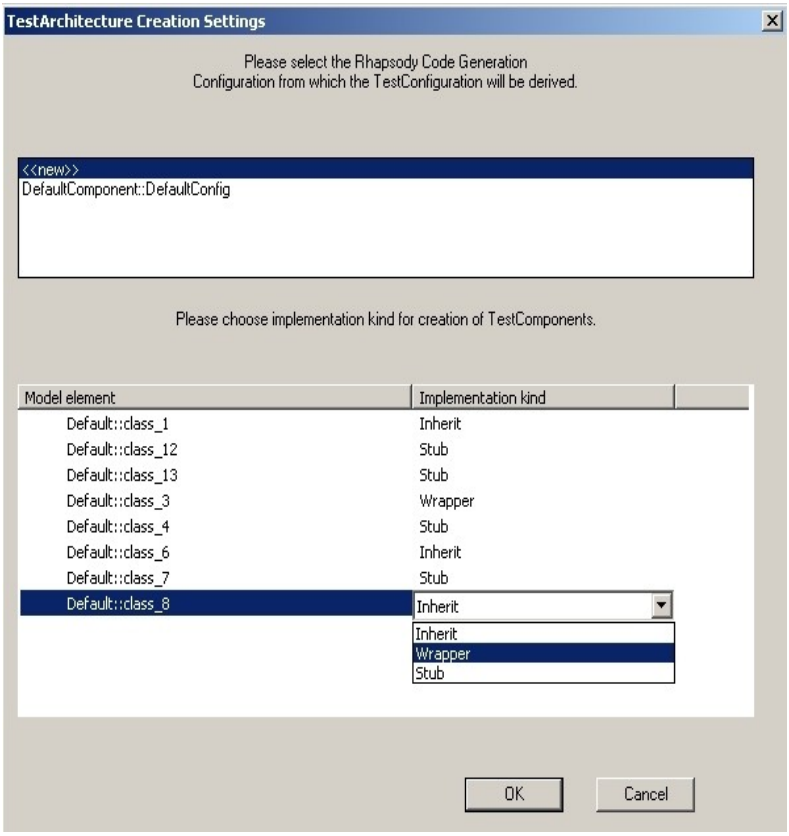


Figure 3: Advanced TestArchitecture Creation Dialog

As 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:

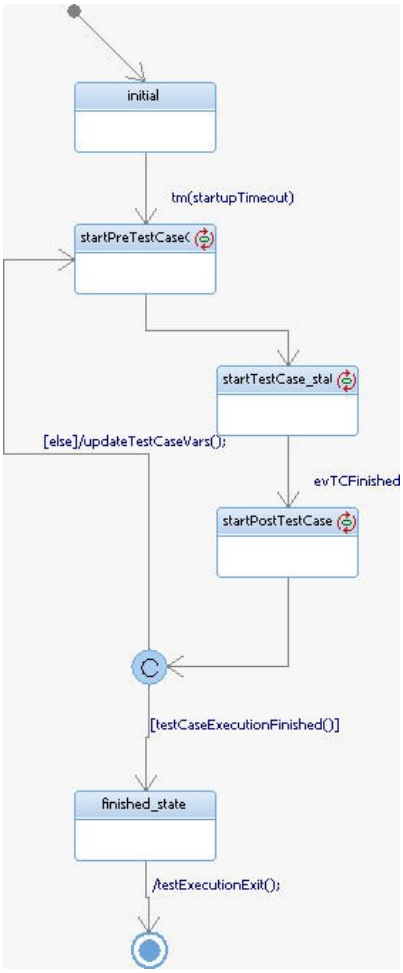


Figure 4: TestCase Scheduler Statechart

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 for all TestCases of a TestContext on page 89.

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[11] 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 130).

[11]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.

The UML TestingProfile defines a weak relation among TestContext and Test Schedulers. In theory, a TestContext could be related to multiple schedulers, although there is no real benfit from this, since the scheduling of TestCases can be influenced by various other measures (order of TestCase, using TestSets, executing entire TestContext, e.t.c).

In previous versions of TestConductor, the relation of a TestContext was realized by an attribute of the TestContext that only kept the reactive base class of scheduler, s.t. the TestContext can send events to the scheduler. From TestConductor 2.8.4 for Rhapsody 9.0.1 and up, this relation is typed stricter by using an association from TestContext to the scheduler. TestConductor is fully compatible with the older and weaker relation in existing TestArchitectures, but will use an association instead of an attribute when creating a TestArchitecture[12].

Test arbitration with `<<Arbiter>>` TestComponents

If you define the behavior of a TestCase by using a sequence diagram, in assertion based testing mode 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>>`:

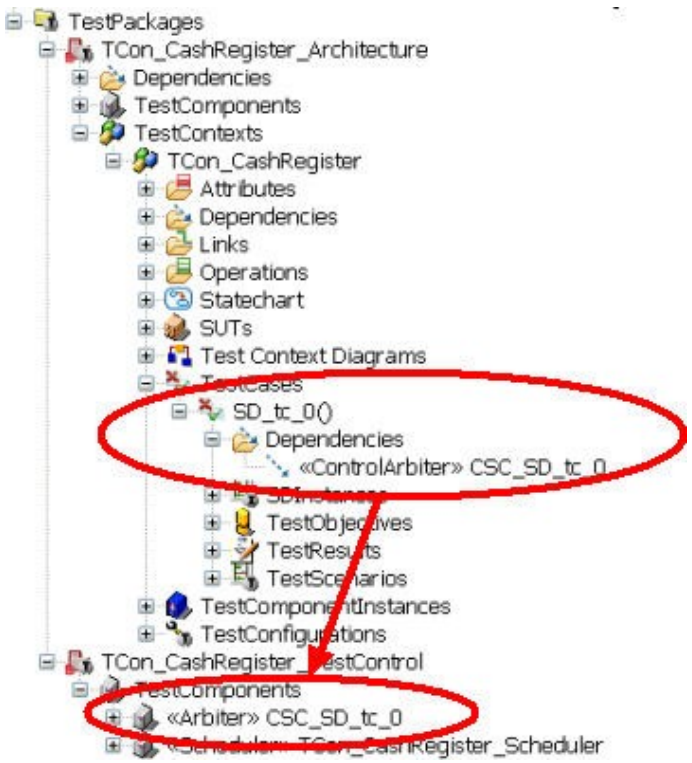


Figure 5: Arbiter of SD TestCase

[12]Exception: For models using frameworks MXF and SMXF, TestConductor still uses the base-class attribute.

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 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 'Clean TestPackage' from the context menu with cheking 'Clean Code' in the opening dialog. Alternatively,'Code->Clean' can be invoked form the Rhapsody menu[13] 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 `<<AssertionBasedTestingConfiguration>>` (for Java) or `<<TestingConfiguration>>` (for C, C++) 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 Tutorials for Rhapsody in C, C++, Java and for Systems Engineers.

See also TestingCookbook:

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

[13]The Code menu is available in the Rhapsody Developer-Edition. If you are using the UML Perspectives settings, make sure to have Advanced perspective enabled. The Basic perspective hides the Code menu.

Creating TestArchitectures for single individual classes (Rhapsody in C, C++ and Java - 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, etc.

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

Special Case: Class inheriting from Template

Note, that template classes can not be instantiated directly as an object. Hence, a direct instantiation as SUT in a TestArchitecture is also not possible. In order to instantiate a template, a class has to be created as a bound instance of the template. There are in principle two alternatives to create such a parameter-bound instance of a template:

- using the model element Instantiation (radio button in the features dialog on class). The base template and template parameters have to be bound to types and values using the extra tab 'Template Instantiation' appearing in the features dialog of the class after selecting the 'Instantiation' radio button.
  Code generation will result in a typedef with the defined parameter binding. Since template instantiation results in a typedef only, adding specialization functions as well as adding additional functions to a 'Template Instantiation' is not possible.
- using a regular class and a generalization relation with the template as base.
  On the generalization, template parameters of the base template can be bound to types and values. For the inheriting class, class code will be generated as for a regular class (inheriting from the template w.r.t. the parameter bindings). Additional functions and attributes etc. can be added to the inheriting class as usual and code will be generated as for a usual class.

While there is no support for testing templates or template instances directly, TestConductor supports testing for classes inheriting from templates in a specific way: If a class inherits from a template and all template parameters are bound at the generalization relation, TestConductor can add pecializations of the template functions to the inheriting class. Such specialization functions have a signature according to the template parameter bindings. Internally in their body, they call the inherited function from the template base.

As an example, we assume a template class `MyStack<T>` with members '`void push(T x)`' and '`T pop(void)`' has been defined in the model. Then, a class `intStack` can be created that inherits from `MyStack`. On the generalization, template parameter '`T`' is bound to '`int`'.

'`intStack`' can then be equipped with the following operations:

```
void push(int x) {
   MyStack::push(x);
}
```

and

```
int pop(void) {
   return MyStack::pop();
```

```
}
```

TestConductor offers a dedicated helper '`Apply TemplateInstantiation`' for adding such specialization functions to a class inheriting from a template. In the creation of TestComponent- and GreyBox replacements, such functions will automatically be added if possible.

Using this functionality, TestCondutor lifts the semantics of generalization with parameter binding onto the modeling level.
When testing such a class inheriting from a template, the specialization functions can be invoked without caring about parameter bindings. The specializations are offered as message realizations for messages in a sequence diagram. Moreover, the specializations can be instrumented and stubbed according to the needs of testing.

Even though, replacements created from classes inheriting from template will be equipped automatically with the described wrapper functions, it is recommended to apply 'Apply Template Instantiation' on the design class before creation of a TestArchitecture. Otherwise fundamental manual adaptions to the TestArchitecture containing such replacements will be necessary in order to get TestCases working

Special Case: Static Inheritance

Static inheritance is a powerful feature in particular in Rhapsody in C, since there only exists inheritance from interfaces otherwise. Static inheritance is inheritance by copy, i.e. the specialization of the static generalization inherits everything from the base which isn't defined already in the specialization. In particular, not only virtual operations as in regular inheritance but also non-virtual operations are inheritable using static inheritance.
In Rhapsody, the feature 'static inheritance' is realized in the code generation only, by internally generating code for the base and then copying all artifacts that aren't overridden in the specialization of the static generalization to the specialization. On model level, static inheritance is represented only by a generalization relation with stereotype <<static>>. There is no representation of the inherited elements in the specialization in the internal Rhapsody model. Moreover, due to the deviation of static inheritance from regular inheritance, also editors like the SequenceDiagram editor don't apply the rules for static inheritance. E.g, a non-virtual function inherited by static inheritance will not be offered as message realization on the life-line of the specialization of a static generalization.
TestConductor offers a dedicated helper to lift static inheritance to the model level. Using '`Apply StaticInheritance`' can be applied to the specialization of a static generalization. The helper copies all model elements that are not already defined in the specialization from the base to the specialization. In order to update such a specialization according to changes in the base, the indvidual model elements in the specialization are marked by a tag, if they were copied and not locally modified. To modify locally, the tag shall be removed by the user. Subsequent application of the helper, will update the specialization with changes in the base of the static generalization.

For replacements inheriting statically, the functionality is applied automatically when creating and updating the replacement.

Even though, replacements created from classes inheriting statically will be completed automatically by copying model elements from the base, it is recommended to apply 'Apply Static Inheritance' on the design class before creation of a TestArchitecture. Otherwise fundamental manual adaptions to the TestArchitecture containing such replacements will be necessary in order to get TestCases working

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 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 126)

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>>`.

If TestArchitectures for implicit objects are created without global object support (to enable global object support 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.

TestArchitectures for objects created without global object support will not affect port connections and other relations 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 52.

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, objects and Files[14]

(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

[14]Files are supported as a modelling construct only in Rhapsody in C. Files are similar to singleton implicit objects. No me-pointers are needed for functions of files.

generation with assertions, checking the values of operation arguments, the return values from operation calls, the order of messages between TestComponents and the SUT etc.

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 45).

When property 'TestConductor::Settings::CreateTestArchitectureTransparency' is set to 'GreyBox' (default is 'BlackBox'), the SUT Class is copied to the TestArchitecture as <<TestSUT>>[15]. 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, TestArchitecure 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 etc., 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>>.

[15]For Java models, the <<TestSUT>> will be created in the dedicated replacements TestPackage associated with the TestArchitecture.

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 40). 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 49 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*. Since the MicroC code generation is highly scope-dependent, code generation 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, instances of replacements are often not initialized at all. 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'. Also 'C_CG::Configuration::FrameworkInitializingMode' should be set to 'RunTime' in many cases.

Note, that testing for run time initialization scheme tests, of course, a different behavior than for compile time initialization, but might be a necessary workaround to be able to test at all.

For greybox Testing (cf. section Grey Box Testing on page 45) this setting of properties is required.

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 45 and GreyBox TestArchitectures for classes, objects and Files on page 41).

For using assertion based testing mode with MicroC models cf. section TestArchitectures for MicroC Models on page 44)

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[16] 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 to apply 'Update TestArchitecture' (cf. section Updating TestArchitectures) after modifying the SUT in the model. TestConductor offers a dedicated helper ‘Check if TestArchitecture is up-to-date’ for keeping track of changes in the model with respect to the TestArchitecture.

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[17].

- 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 23). 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.

[16]For Java models, the copy of the SUT class will be created in the dedicated replacements TestPackage associated with the TestArchitecture.
[17]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>> , <<greyboxinstancereplacement>> and <<greyboxfilereplacement>> dependencies of the copy on the original element in order to consider the dependencies on requirements in the original model elements.

- since TestConductor has instrumented event processing in order to establish observability of event consumption, GreyBox testing can not deal with non-standard event consumption[18].
- Code coverage measurement (cf. section Computing Code Coverage on page 109) is not meaningful for GreyBox testing, since the SUT copies are instrumented for testing purposes and Code coverage 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 workaround-ed 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.

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.

[18]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.
For Java, classes with statecharts inherit from RiJStateConcept. The treatment of the statechart behavior is realized in an implicit reactive part (added by code generation) of the statechart owning class. TestConductor overrides RiJStateConcept::takeEvent() and RiJStateConcept::gen() with own implementations. RiJStateConcept::gen() is overridden in order to correct the destination setting for the generated event. Normally, the destination of a generated event is set to the implicit reactive part of a class owning a statechart. In order to be able to observe event consumptions, the destination has to be the statechart owning class itself. Observation of the event consumption is then realized in the overridden RiJStateConcept::takeEven() function. After performing the observation notifications for testing purposes, this overridden function calls the original takeEvent() operation of the implicit reactive part of the statechart owning class.

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 (not available for Rhapsody in Java)
- 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 for C and C+++ can be found in the chapter TestConductor Assert Macro on page *200*. Assertion functions for Java can be found in chapter TestConductor Assertion Functions (Java) on page 203.

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);
RTC_ASSERT_NAME("reaction",RiCIS_IN(&(me->itsClass_0),
                                    reaction_state));
```

- Example code in Java:
```
itsClass_0.gen(new evX());
try {
   Thread.sleep(1000);
} catch (InterruptedException e) {}
TestConductor.ASSERT_NAME("reaction",itsClass_0.isIn(
                                 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.

FlowChart TestCases are not available for Rhapsody in Java.

Defining a Flow Chart TestCase

- Right-click on the TestContext 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 48).

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 staechart.

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. Also a directed association to the SUT is added to the <<SCArbiter>> TestComponent. These associations are initialized in the constructo/initializer of the <<SCArbiter>>TestComponent with the arguments provided by the <<SCTCInstance>> TestComponentInstance, which is added to the TestContext on “Update TestCase”, “Update TestContext”, and “Update TestPackage”, respectively.
'Update TestCase' also instruments the TestCase operation with sending an evTCStart event to the TestComponentInstance to start statechart execution, i.e. trigger its initial transition.

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 *137 ff*.

Detailed information on generation of DriverOperations and StubOperations can be found in section Model Population – Create Driver Operations and StubOperations on page 55 and in section Influencing DriverOperation and StubOperation Generation on page 152).

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 45, 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 TestSceanario specification of the newly created TestCase.

Specification using TestScenarios is discussed in detail on pages 137 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 *68* (in particular: Interpretation of witness diagrams on page 86).

Further information about failure analysis can be found in chapter Failure Analysis on page *168*.

Specification and Verification of Invariants for Test Execution

Using the separate '`InvariantCheckProfile`' profile, expressions ranging over attribute valuations and state names can be specified to be checked for invariant validity during particular test executions. These expressions are expected to hold invariantly during the execution of associated TestCases/TestSets/TestContext/TestPackage. If 'invariant checking' is enabled by setting a dedicated tag on the code generation configuration, the validity of the respective expressions will be checked during the execution of these TestCases to which the expression has been related.

For details of invariants specification and verification see chapter Specification of Invariants and Verification of Invariants during Test Execution on page 96 ff.

Rhapsody in C++ and C: TestConductor.h, TestConductor_C.h and TestConductor_C.c

In order to use the predefined assertion macros, TestContext, TestComponents, and and grey box SUT replacements have to include the TestConductor header file by setting property `CPP_CG::Class::ImpIncludes` 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::ImpIncludes` 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).
For a list of supported assertion macros see chapter TestConductor Assert Macros (C/C++) on page 200.

Rhapsody in Java: TestConductor.jar and import of com.btces.testconductor.assertionbased.TestConductor

In order to enable use of the predefined assertion functions, TestContext, TestComponents, and and grey box SUT replacements have to import the TestConductor class by setting property `JAVA_CG::Class::SpecIncludes` to `com.btces.testconductor.assertionbased.TestConductor`.
Additionally, TestConductor adds the path '`%RHAP_JARS_DIR%/../../../TestConductor/TestConductor.jar`' to the classpath of the code-generation component when creating a TestArchitecture.

For a list of supported assertion functions, see chapter TestConductor Assertion Functions (Java) on page 203.

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 `<<CIntefaceFile>>` 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. I such cases, stereotype `<<nopublicwrapper>>` 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 `PrepareForTestingPivateOps`.

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[19] as the static operations and of course also an appropriate header has to 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[20].

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. I 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.

[19]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.
[20]This comment can be added to the generated file by e.g. round tripping or via `CPP_CG::WriterTemplates::ClassSpec` property.

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

TestConductor Support for Testing Private and Protected Operations in Rhapsody in Java

There is currently no dedicated support for testing private and protected operations in Java.

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 152).

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, etc.

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.

A variable declaration and assignment for a return of the driven operation will only be generated if a return value is specified. If a variable declaration and assignment of the return to this variable is neede, but no return value shall be specified, then the appropriate declaration and assignment can be enforced by specifying '*' as return value. This way, the returned value is not checked by TestConductor, but can be used further, e.g. in a `<PostCallAction>` TestAction (cf. section Influencing DriverOperation and Stub generation using TestActions in TestScenarios on page 157).

If the sequence diagram specifies that the returned value has to be checked, the macro `RTC_ASSERT_SD_NAME` for C and C++ or TestConductor.ASSERT_SD_NAME function for Java, respectively, is used to check if the returned value adheres to the specified return value[21]. The same macro is used also to check out or in/out argument values (cf. 149) 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 *153*.

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

[21]If suitable, TestConductor uses RTC_ASSERT_SD_NAME_ACTUAL for C++ and C or TestCondcutor.ASSERT_SD_NAME_ACTUAL for Java, in order to report the actual value of e.g. checked arguments or returns.

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 59), 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.

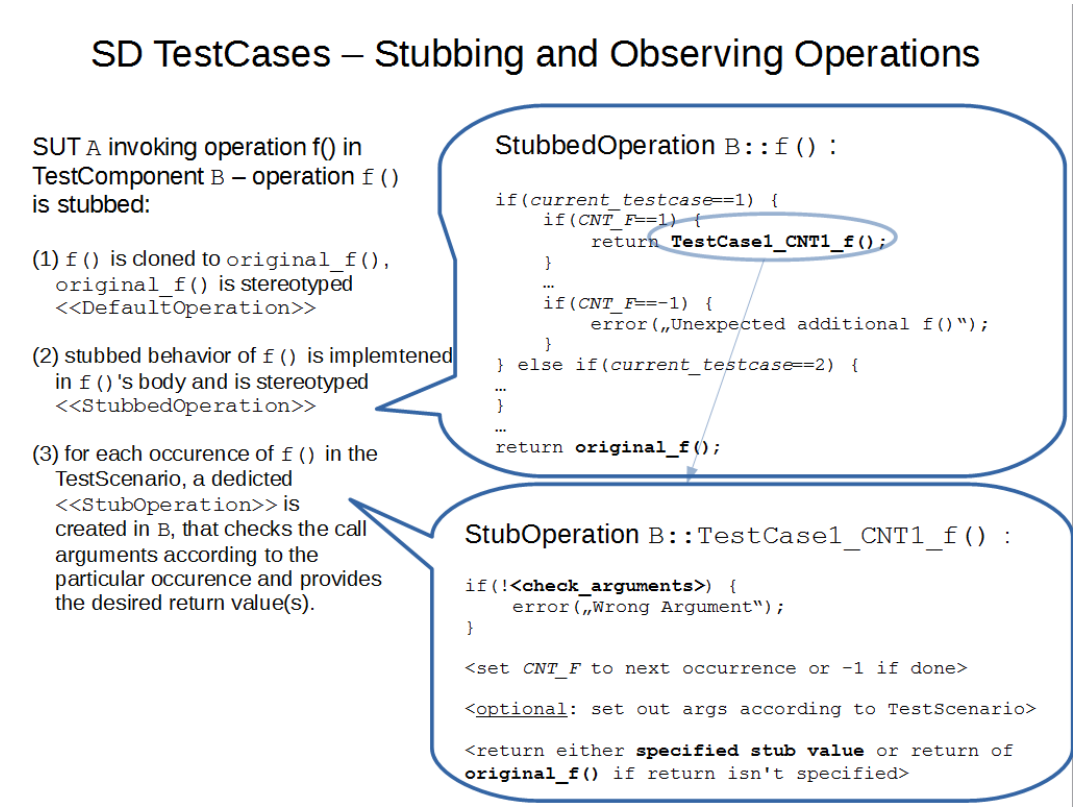


Figure 6: SD TestCases - Stubbing and Observing Operations

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 User Defined StubOperations and following chapters at page *154 ff.*

Multiplicity of Relations

For the generation of driver- and stub-operations, TestConductor has to analyze end to end connections among instances in the TestArchitecture, to e.g. identify the correct sender instance for a message reception specified on a particular life-line in a TestScenario or to identify the real receiver instance for a message sent by a particular life-line in a TestScenario. When multiplicities of relations are specified by symbolic constant names, such as e.g. 'MAX_N', where 'MAX_N' denotes a defined constant somewhere in the scope of the code generation, TestConductor can not infer the value of 'MAX_N' automatically and, hence, not calculate the end to end connections established by links or connectors involving relations of such symbolic multiplicities.

On updating a TestCase/TestContext/TestSet/TestPackage affected by unknown symbolic multiplicity parameter, TestConductor will issue warnings, that without knowledge of its value, the end to end connections among instances in the TestArchitecture can not be calculated.

To provide TestConductor with the needed information about the concrete values of such symbolic multiplicity specifiers, a <<MultiplicityConstantMap>> (cf. section Multiplicity of Relationson page 29 and TestArchitecture Package on page 183) can be defined by the user for the TestContext of the TestArchitecture.

On the TestContext, 'Add new ->TestingProfile->MutliplicityConstantMap' can be invoked from the context menu to add a <<MultiplicityConstantMap>>. For each symbolic multiplicity parameter relevant for the TestArchitecture, a tag can be added, where the name of the tag is the name of the symbolic multiplicity parameter and its value is the concrete value of that parameter.

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.**

Clean TestPackage opens a dialog offering three check boxes for selection what exactly will be deleted by the clean operation:

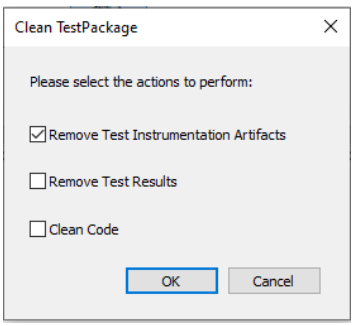


Figure 7: Clean Package Dialog

- Remove Test Instrumentation Artifacts: all automatically generated driver, stub and stubbed operations will be removed that were generated by updating TestCase, TestContext or TestPackage.
- Remove Test Results: all results and reports of the previous test executions are removed.
- Clean Code: all object files and the executable will be removed from the code generation directory, s.t. the next build will have to recompile all generated sources files.

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.

For Rhapsody in Java: 'Clean TestPackage' on the `TPkg_<element>` TestPackage will also automatically apply on test artifacts in the replacements package associated to the TestArchitecture. There is no need to apply a clean on the replacements package separately.

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, etc. 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 137 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 Error: Reference source not found on page Error: Reference source not found).

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.

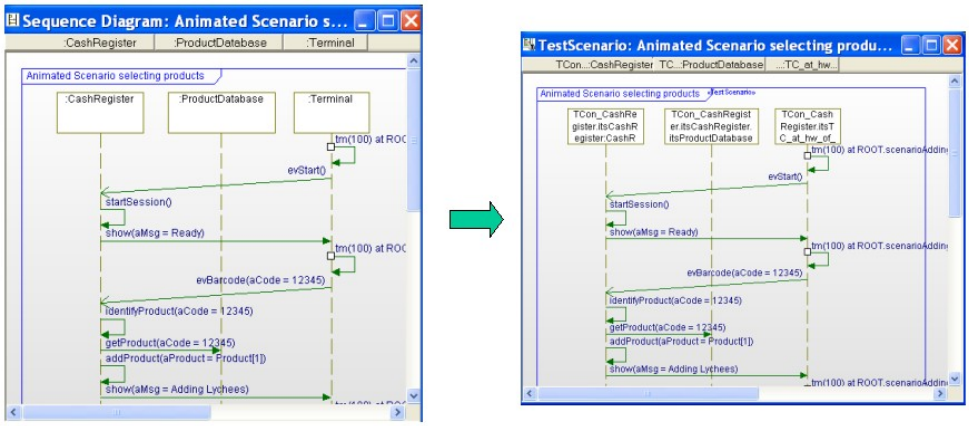


Figure 8: SD TestCase mapping using TestCase Wizard

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 automatically use instance labels and, thus, much shorter and better readable names, TestConductor offers property 'TestConductor::TestContext::CreateSDTestCaseWithInstanceLabels'. If the property is set, the life-lines in the TestScenario will show the part names as labels. The names are still generated according to the navigation path, but will not show up in the scenario.

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 61 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 TestCase 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' classifiers in the selected TestArchitecture – according to the chosen mapping strategy (weak or strict, cf. pages Settings – General Properties, page 126 ff.). 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 and *activated* 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 64.

Mappings refer to the classifiers of life-lines. Mapping of individual messages is currently not supported.
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 automatically use instance labels and, thus, much shorter and better readable names, TestConductor offers property 'TestConductor::TestContext::CreateSDTestCaseWithInstanceLabels'. If the property is set, the life-lines in the TestScenario will show the part names as labels. The names are still generated according to the navigation path, but will not show up in the scenario.

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 `<<AssertionBasedTestingConfiguration>>` 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[22].

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.

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

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.

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 23). 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[23]. Thus, for mappings also the original model elements have to be selected – not the replacements.

Problems with SequenceDiagrams recorded in interactive sessions

In some cases, recorded SequenceDiagrams may contain invisible elements such as messages without source or target location. This can appear when recording Sequence Diagrams from interactive animated sessions using panels, using the animation toolbar, webify interaction or event generation from animated Sequence Diagram,
In such cases, TestCase creation from existing scenarios may fail to read the respective scenario or it may create a TestCase that is not usable with TestConductor - TestConductor refuses updating the created TestCase with error messages regarding invalid messages in the TestScenario.
To repair such Sequence Diagrams or TestScenarios, i.e. remove the invalid (invisible) messages from the diagram, TestConductor offers a dedicated helper "TestConductor->Repair Sequence Diagram" in the context menu of Sequence Diagrams and

[23]Except for TestFiles replacing files.

TestScenarios.
Invoked on any SequenceDiagram or TestScenario, the helper will provide a short report in the Rhapsody log window. Only if defective elements are detected, the helper will ask, whether the diagram shall be repaired (if possible). If the user decides to let the helper attempt to repair the diagram, a backup of the affected diagram will be created before attempting to repair it.

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.

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.

Overview

TestConductor supports execution of

- individual TestCases (cf. TestCase Execution on page 81 ff)
- TestContext (cf. TestContext Execution on page 88 ff)
- TestPackage (cf. TestPackage Execution on page 93 ff)
- TestSet (cf. Organizing TestCases in TestSets and TestSet Execution on on page 94 ff)

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.

Orthogonal to the definition and execution of individual TestCases, TestConductor supports the definition of invariant expressions and their verification w.rt. the individual TestCases or TestSets or for all test executions of an entire TestContext (cf. Specification of Invariants and Verification of Invariants during Test Execution on page 96 for details).

The execution of TestCases optionally makes use of the external tool hstart.exe (cf. Usage of hstart.exe to hide console application windows on page 208) by checking tag NoConsoleApp on the respective code generation configuration (see next section). Although suppression of consoles popping up during the execution of TestCases makes the execution of TestCases less perturbing and allows for working on the machine in parallel on other things, hstart.exe is considered a security problem by some antivirus and security checking tools. If your administration has doubts regarding hstart.exe, hstart.exe can be removed from the installation (it is installed into the Share/etc directory of your Rhapsody installation)- only the NoConsoleApp feature won't work without that tool.

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>>`(C and C++) or `<<AssertionBasedTestingConfiguration>>` (Java) or a specialization of these stereotypes. Figure 9 gives an overview of the hierarchy of code generation configuration stereotypes defined by the TestingProfile.

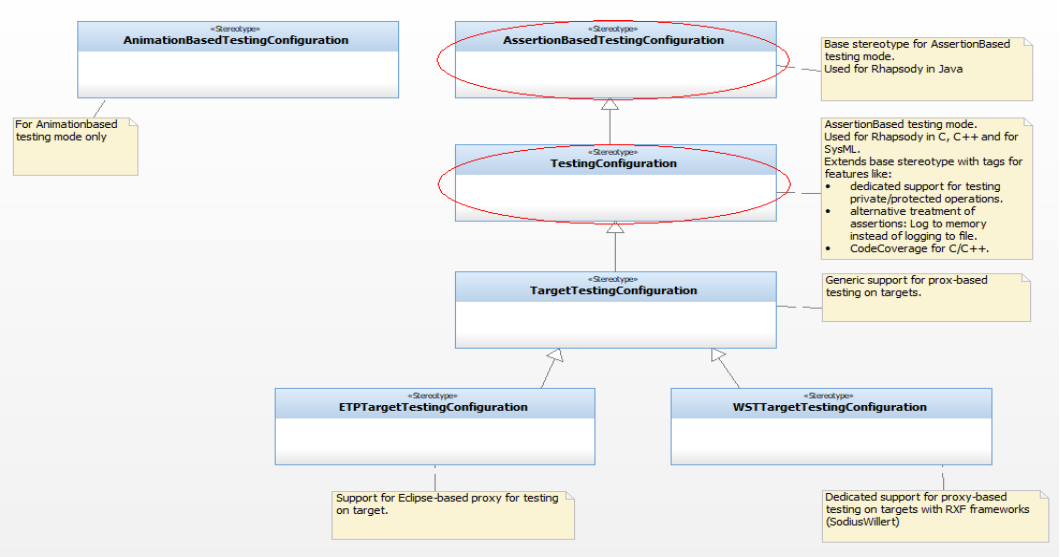


Figure 9: Hierarchy of the code generation configuration stereotypes defined by the TestingProfile

Each TestArchitecture is automatically equipped with either a `<<AssertionBasedTestingConfiguration>>` or `<<TestingConfiguration>>` code generation configuration upon TestArchitecture creation using the context-menu 'Create TestArchitecture' TestConductor[24].

Tags of the `<<AssertionBasedTestingConfiguration>>` and `<<TestingConfiguration>>` Stereotypes

The stereotype `<<AssertionBasedTestingConfiguration>>` and its extension `<<TestingConfiguration>>` contains several tags that can be used in order to

[24]In the context of this document, we focus on the AssertionBased testing mode. Property TestConductor::Settings::TestingMode = {AssertionBased,AnimationBased} on the project determines, for which testing mode, a TestArchitecture will be created. If AssertionBased is chosen (default for C, C++, Java and SysML) and a TestArchitecture is created for some selection, the automatically created code generation configuration will be stereotyped appropriately, i.e. `<<AssertionBasedTestingConfiguration>>` for Java models and `<<TestingConfiguration>>` for C- , C++ - and SysML models, respectively.

control how the test executable is created, and which test execution options should be applied when executing TestCases using that configuration.

Tags of stereotype <<AssertionBasedTestingConfiguration>>:

- **ComputeModelCoverage:**
  If this option is turned on, when executing TestCases TestConductor measures which model elements of the SUT are covered. TestConductor computes which states, transitions and operations are executed by the TestCases. Which parts of the model are considered for code coverage can be controlled by the tags "CoverageScopeIncludeInheritance", "CoverageScopeIncludeParts" and "CoverageScopeIncludeTestArchitecture".
  Note: Model coverage was restricted to animated configurations before Rhapsody 9.0.1. This restriction does not apply anymore. Model coverage is based on source code instrumentation from Rhapsody 9.0.1 on and can also be applied to TestArchitectures with code generation configuration not enabling animation instrumentation

  (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 102 ff. for details on the coverage measure and its configuration options).

  Note: Requirement coverage was restricted to animated configurations before Rhapsody 9.0.1. This restriction does not apply anymore. Model and Requirement coverage are based on source code instrumentation from Rhapsody 9.0.1 on and can also be applied to TestArchitectures with code generation configuration not enabling animation instrumentation

  (Default: false)

- **CoverageScopeIncludeParts**:
  When tags ComputeModelCoverage or ComputeCodeCoverage are checked, TestConductor adds coverage related instrumentation to the generated source files, for which coverage information will be computed. By default, this instrumentation for coverage measure affects not only the SUT element itself, but also all parts instantiated in the SUT. By unchecking tag CoverageScopeIncludeParts, the instrumentation for coverage measure can be restricted to the SUT[25] itself. If CoverageScopeIncludeParts is set to false, i.e. tag is unchecked, then no coverage measure will be computed for the parts instantiated in the set of elements considered by coverage instrumentation.

  (Default: true)
  Despite determining the coverage scope using tags 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.

[25]Depending on tag CoverageIncludeTestArchitecture, not only the SUT but also the TestComponents can be instrumented for coverage measure.

- **CoverageScopeIncludeTestArchitecture**:
  When tags ComputeModelCoverage or ComputeCodeCoverage are checked, TestConductor adds coverage related instrumentation to the generated source files, for which coverage information will be computed. By default, only the SUT and its parts (depending on tag CoverageScopIncludeParts) will be instrumented for coverage measure. If tag CoverageScopeIncludeTestArchitecture is set to true, i.e. tag is checked, then coverage instrumentation will be performed also for the TestComponents and the TestContext.

  (Default: false)
  Despite determining the coverage scope using tags 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.

- **CoverageScopIncludeInheritance**:
  By default, coverage instrumentation will only affect the source files of the model elements directly appearing in the TestArchitecture. In case of inheritance, base classes of these elements will not be considered at all.
  Tag CoverageScopeIncludeInheritance can be set to true, i.e. checked, in order to instrument also the base classes of the set of elements considered in coverage instrumentation.

  (Default: false)

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

  (Default: false)

- **EnableInvariantVerification**
  Orthogonal to the definition and execution of individual TestCases, TestConductor supports the definition of invariant expressions and their verification w.r.t. the individual TestCases or TestSets or for all test executions of an entire TestContext. There is a separate document in addition to this user guide, describing the definition, handling and verification of such invariants. This document can be found in the '`InvariantCheckProfile`' profile in the Share directory of your Rhapsody installation.
  Tag `EnableInvariantVerification` is used to switch the 'Invariant Verification feature' on or off for an `<<AssertionBasedTesting-Configuration>>` code generation configuration.
  (Default: false)

- **NoConsoleApp**:
  This tag controls whether TestConductor should hide console windows (cmd windows) when executing test applications and helper tools on Windows.
  Please see also Usage of hstart.exe to hide console application windows on page 208 regarding the helper tool used for this option on Windows machines.

  (Default: false)

- **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)

- **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)

- **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[26].

  (Default: true)

- **rtc_log_filename**
  This tag specifies the name of the log file that can be generated by the test executable[27].

  (Default: $CONFIGDIR/rtclog.txt)

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

  (Default: $CONFIGDIR)

[26]For stereotype `<<TestingConfiguration>>`: 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.
[27]For stereotype `<<TestingConfiguration>>`: If the file is generated or not during test execution depends on the value of the tag "rtc_log_kind"

- **rtc_result_filename**:
  This tag denotes the file from which TestConductor will read the result of TestCase execution[28].

  (Default: $CONFIGDIR/rtcresult.txt)

- **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.

  Note: When testing with MinGW environment a trailing "$OMROOT/etc/MinGWrun.bat" (with $OMROOT expanded) is added to the script content to make sure the tested application can load the necessary MinGW dlls and start correctly.
  This can be disabled by adding the tag "DisableUseMinGWrun" 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_uptodate_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 involevd. Tag rtc_testexecution_uptodate_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.

[28]For stereotype <<AssertionBasedTestingConfiguration>>: Assertion result will automatically be written into this file.
For stereotype `<<TestingConfiguration>>`: If tag **rtc_assert_dumptofile** is set to true, then the results will automatically be written into this file.

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)

- **tc_testreport_script_content:**
  This tag specifies the content of the script file that is used by TestConductor to generate html execution reports for TestCases, TestContexts and TestSets 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, TestContexts, and TestSets.

  (Default: “$RTCINSTALLDIR/repgen" -infofile "$infofile" -resultfile "$resultfile" -language $LANG -outdirectory "$RTCREPDIR" -tcontext $fulltcontext -tset $fulltset -tcase $fulltcase)

- **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 “tc_testreport_script_content”.

  (Default: true)

Tags of stereotype `<<TestingConfiguration>>`

Stereotype `<<TestingConfiguration>>` extends stereotype `<<AssertionBasedTestingConfiguration>>` with additional tags aimed at supporting

- CodeCoverage
- assertion-handling in the memory of the application instead of directly writing a file
- support for testing of private and protected operations.

These features of TestConductor need dedicated configuration capabilities by tags, which are listed below:

- **ArbiterUsesGuardedOperations**
  By default, TestConductor generates triggered operations for the arbiters with a guard (property `CG::Operation::Concurrency=guarded`). Operation guards are based on operating system mutexes. For execution on tagregts e.g. without operating system or without support of mutexes, the triggered operations of arbiters can be generated as ‘normal’ `sequential` triggered operations by switching off this tag.

  (Default: true)

- **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 tags “CoverageScopeIncludeInheritance” and “CoverageScopeIncludeParts”.
  Note: Code coverage is restricted to C and C++.

  (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)

- **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 53 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 54 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)

- **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_dumptofile**:
  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_dumptofile_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 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:

Lines ::= ε
| **Lines Line**

Line ::= ASSERTION = nr,ln,e

Where ε 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_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_result_handling**: (obsolete)
  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.

TestConfiguration Dependency/Determining the code generation configuration to be used

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>>` or

<<AssertionBasedTestingConfiguration>>. If the active code generation configuration is stereotyped <<TestingConfiguration>> or <<AssertionBasedTestingConfiguration>>, 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.

TestSet with <<TestSetDedicatedConfiguration>> dependency

TestSets can determine independently of the above rules using which configuration they are updated, build and executed (cf. section Organizing TestCases in TestSets on page 94 ff). If a TestSet has a <<TestSetDedicatedConfiguration>> dependency on a code generation configuration, then this configuration will be activated on updating, building and executing the TestSet.

Execution Results

After the test executable has been built, individual TestCases, TestSets, entire TestContexts or TestPackages can be executed. When invoking a TestCase from within Rhapsody, TestConductor calls the script specified in the <<AssertionBasedTestingConfiguration>> (<<TestingConfiguration>>, respectively) tag "rtc_testexecution_script_filename" that actually calls the test executable with the parameters selecting the TestCase which shall be executed. The chosen TestCase is executed, and after termination the results are read from the result file specified in the tag "rtc_result_filename"[29]. 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 optionally verify the result (tag "ResultVerification", see below) and 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 or outside of Rhapsody.

[29]For <<AssertionBasedTestingConfiguration>> , assertion are always logged in the result-file immediately in a textual based format. For <<TestingConfiguration>>, it can be configured, whether the assertions will be represented by a textual based format or by a concise format in the memory of the test applictaion first and then be dumped into the result-file at termination of the test application (see tags "rtc_assert_handling", "rtc_assert_dumptofile_kind", "RTC_MAX_ASSERT", "rtc_assert_mem_code").

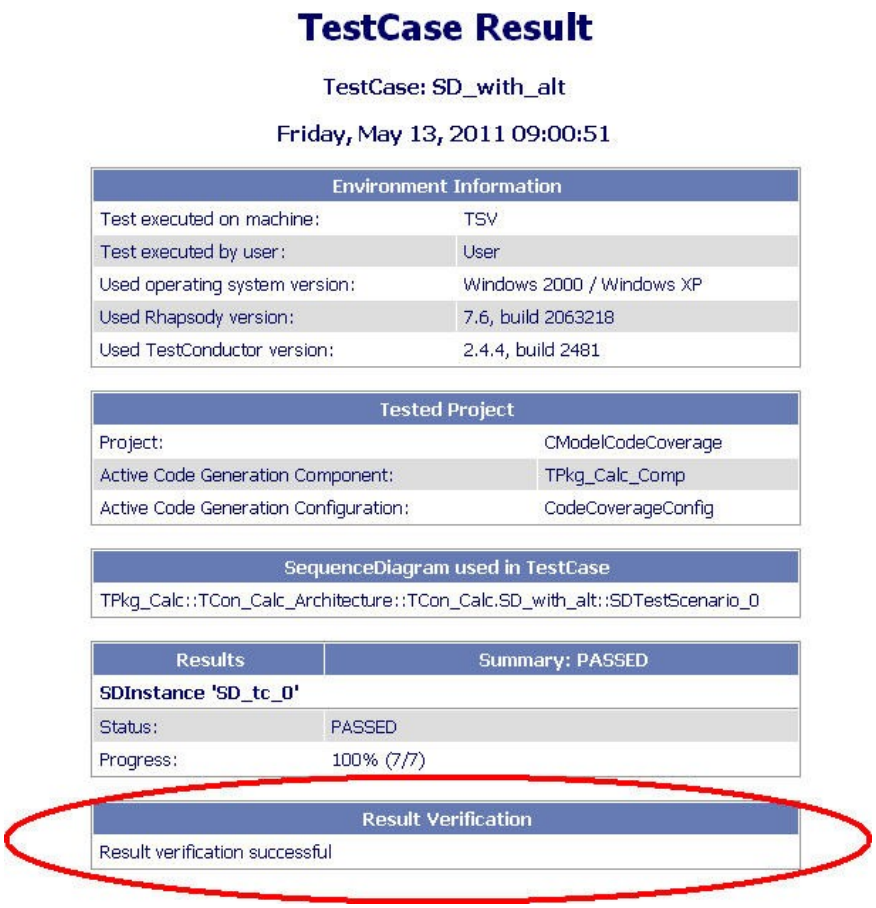
Performing result verification for TestCase execution

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. If result verification is turned on, TestConductor checks after TestCase execution if the results written to the result file by the test executable are consistent with the graphical behavior description in Rhapsody w.r.t e.g. the order of assertions in the diagram (either as sequence diagram, statechart, or flowchart). For a behavior description provided as plain code no result verification is performed[30].
For graphical TestCase specification provided as a sequence diagram, TestConductor populates the model with a statechart that represents the possible allowed execution sequences specified in the sequence diagram. Result verification is then turned out on the basis of this statechart generated from 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”.

[30]In this case, TestConductor only checks, whether the file and line-number information of the logged assertions validly denote the occurrence of the respective assertion in the source code.



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 10: 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 55) and build[31]:

- The context menu on TestCase, TestContext and TestPackage offers '`Update TestCase`', '`Update TestSet`', '`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 TestSet`' instruments the TestContext for the set of individual TestCases belonging to the TestSet. '`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.

[31]Update, build and execute require an adequate code generation configuration to be set active. TestConductor can activate a code generation configuration automatically on invoking update, build or execute on a TestCase/TestSet/TestContext/TestPackage. The rules for code generation configuration activation are described in section TestConfiguration Dependency/Determining the code generation configuration to be used on page 78.

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 TestSet`', '`Build TestContext`' and '`Build TestPackage`' on TestCase, TestContext and TestPackage, respectively.

  By default, building TestCase/TestSet/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 `<<AssertionBasedTestingConfiguration>>` or `<<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 e.g. for C and C++ 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 TestSet`, '`Execute TestContext`' and '`Execute TestPackage`' on TestCase, TestSet, TestContext and TestPackage, respectively.
  Again, this functionality is hierarchically based on executing a single TestCase very similar to update and build).
  Note, that executing a TestCase, TestSet, TestContext or a TestPackage from the context-menu is not the same as invoking ‘Run ...’ from the Rhapsody ‘Code’ menu or hitting the ‘Run’ button in the Rhapsody user interface. In contrast to the default ‘Run’, the context-menu entry will generate a call with several application arguments according to the needs of the TestCase execution.

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

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

Flow chart, code, and statechart TestCases are merely code based TestCases, since 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.

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.

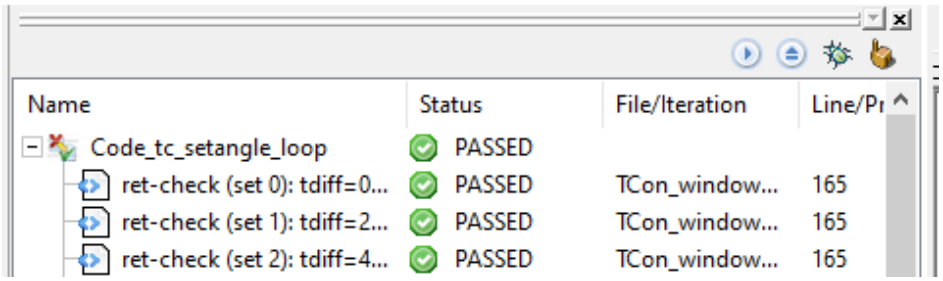


Figure 11: Test Execution Dialog for Code-, Flowchart- and Statechart TestCases

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:

 - (Re-) run the TestCase, TestContext, TestPackage that is currently displayed in the execution window.

 - Abort actual TestCase, TestContext, TestPackage execution.

 - Switch animation based debugging on and off. The animation instrumentation of Rhapsody's code generation allows for graphical debugging. Behavioral diagrams can be animated according to the actual execution, reak points can be set in various ways and execution can be performed with different kinds of steps. We refer the reader to the Rhapsody documentation regarding animation instrumentation and the animated execution of models (see "Running animated models" in the user documentation). When animation based debugging is on, pressing the  icon in the execution window, will start the respective TestCase, TestSet, TestContext, TestPackage execution in animation based debugging mode (Debuging a TestCase is described also in section Debugging TestCases on page 87).

 - Change focus behavior of test execution window: By default, the focus on the list of TestCases in the execution window follows the actual execution and displays the detailed assertion results immediately during execution. If following focus switch is pressed, the executed TestCases appear in folded manner, not immediately revealing the assertion results, but providing an alternative overview over the status of execution.

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.

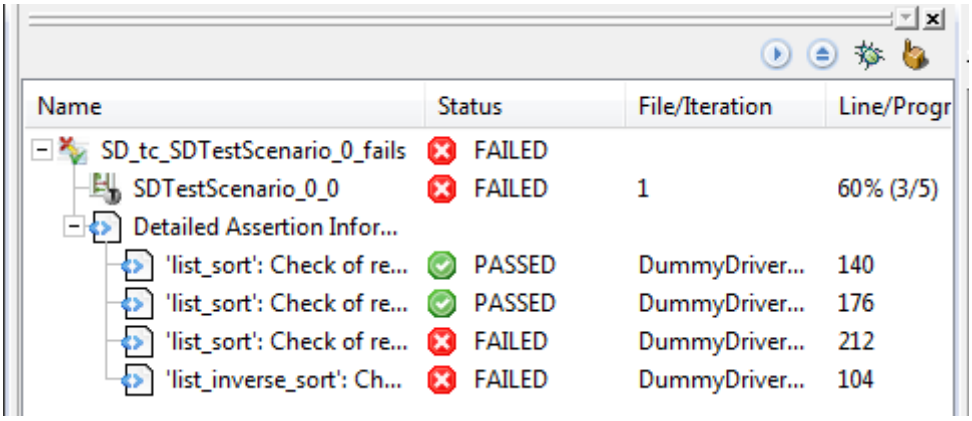


Figure 12: Test Execution Dialog for SD TestCases

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 152 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.
- **Detailed Assertion Information** for the assertions of an individual TestCase-- The detailed assertion information shows information about user defined assertions contributing to the test execution result. In TestActions, statecharts and operations of TestComponents and TestContext and in some other elements, the user can check conditions during test execution using predefined assertion macros/ functions, e.g. RTC_ASSERT_NAME() for C++ and C or TestConductor.ASSERT_NAME() for Java, respectively. The detailed assertion information shows the name of the individual assertion, its execution status, the source file and line number, where the assertion is located in the generated code. A double click on the asserion information or 'Show Assertion' form the context menu of the execution window opens a features dialog showing the location of the assertion in the model.
  Also the assertions from the 'verification of invariants' feature will appear in the detailed assertion information.

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 - either by double clicking the SD instance below the TestCase name or by using the context menu on the SD instance in the execution window.

On invoking “Show as SD”, TestConductor automatically adds a color coded TestScenario in the model to the failed TestCase. Note, that this witness scenario will not be updated when re-executing the TestCase. In order to show a fresh witness for the latest execution, “Show as SD” has to be invoked again.

Also the assertions from the ‘verification of invariants’ feature will be considered in the witness scenario. Since this assertions are not associated with particular messages, the witness can only show after which message the invariant expression was violated. Note, that there is not necessarily a causal relation between the last message observation adhering to the test specification and the instance of time at which the invariant expression was violated the first time.

The resulting witness can be used for failure analysis.

See section Failure Analysis on page *168* 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.
Tag `CreateWitnessScenarioForFailedSDTestCase` of the `<<AssertionBasedTestingConfiguration>>` stereotype (and `<<TestingConfiguration>>`, respectively) of the code generation configuration 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*, click the **abort icon** in the test execution window.

Execution Timeout

Property `TestConductor::TestCase::ExecutionIdleTimeout` (Default 600 (seconds)) can be used to define a timeout for an individual TestCase.
Note, that only very few properties of subject and metaclass `TestConductor::TestCase` – are regarded for TestCases by assertion based testing mode.

If a timeout is defined for a TestCase and the execution does not automatically terminate within the specified number of seconds, the TestCase execution will be interrupted after expiration of the timeout and the result of the execution will be set to “Error”[32].

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.[33] 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*", "*Undefined*" and “*Error*”.

A double click on the test result in the browser opens the linked HTML test report[34].

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:

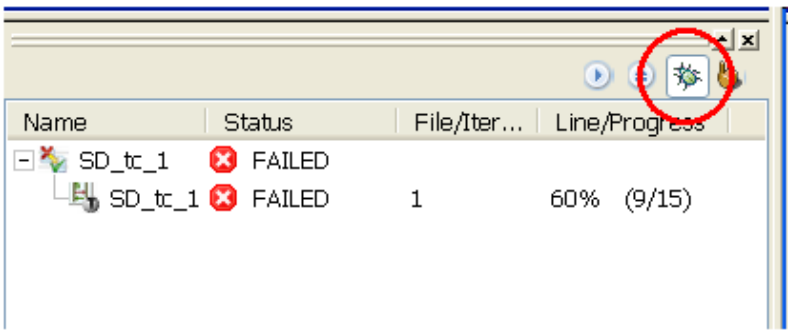


Figure 13: Debugging Button in Test Execution Window

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

[32]The result only depends on assertions in the test application. The test execution is killed after expriration of the timeout. Since the timeout is not controlled from within the test application but from outside, the result generation can not distinguish different reasons for termination before execution completion
[33]Note that using the property TestConductor::Settings::ReportLocation (see page 128) a user can specify a dedicated report location)
[34]Open policy for html documents can be influenced by properties Model::ControlledFile::OpenPolicy and Model::ControlledFile::OpenCommand

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).

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[35].

- 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 'Build 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.

[35]Update, build and execute require an adequate code generation configuration to be set active. TestConductor can activate a code generation configuration automatically on invoking update, build or execute on a TestCase/TestSet/TestContext/TestPackage. The rules for code generation configuration activation are described in section TestConfiguration Dependency/Determining the code generation configuration to be used on page 78.

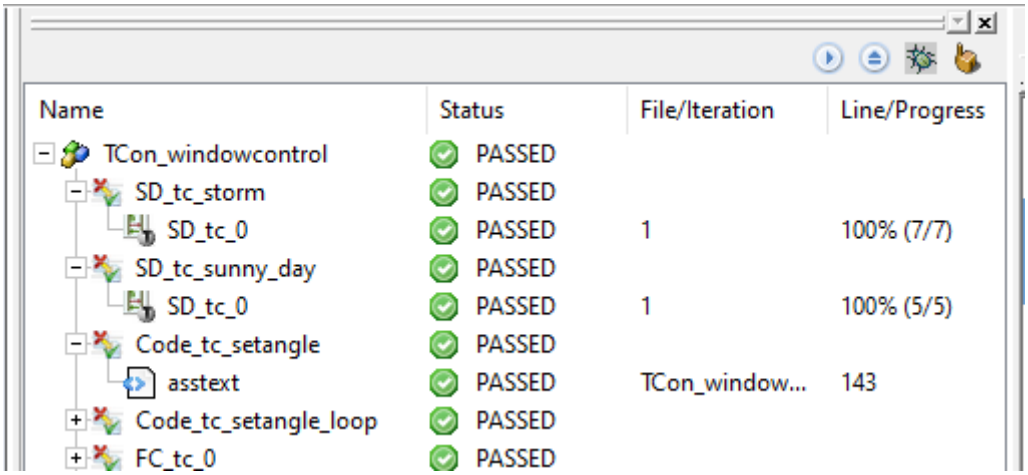


Figure 14: Test Execution Window for TestContext Execution

By default, the test application is restarted for each TestCase belonging to the TestContext. Hence, each TestCase tests a scenario or SUT reaction from beginning with the start-up initialization. Optionally, this behavior can be changed by unchecking property `TestConductor::TestContext::TestContextExecution_RestartExecutable` on the TestContext. Using this property, the test application is started only once and then all the TestCases belonging to the TestContext are executed without restarting the test application. Note, that the order of TestCases is important if all TestCases are executed without restart, since the subsequent TestCases will start in configurations of the SUT that were reached by the preceeding TestCases. If the TestCases are not independent of the particular SUT state, then modifications of the SUT or of the TestCase specifications will have a severe impact on all test execution results.

Stopping Test Execution

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

Execution Timeout for all TestCases of a TestContext

The testing profile defines no overall-timeout for TestPackage or TestContext execution. A timeout can currently be only defined for individual TestCases This can be done via the property

`TestConductor::TestCase::ExecutionIdleTimeout`for the respective TestCase or as a predefinition on a TestPackage - affecting all TestCases in theTestPackage, for which the property is not locally overridden with a different value..

If a timeout is defined and the TestCase does not terminate automatically within <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 "*Error*" in the result report.

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:

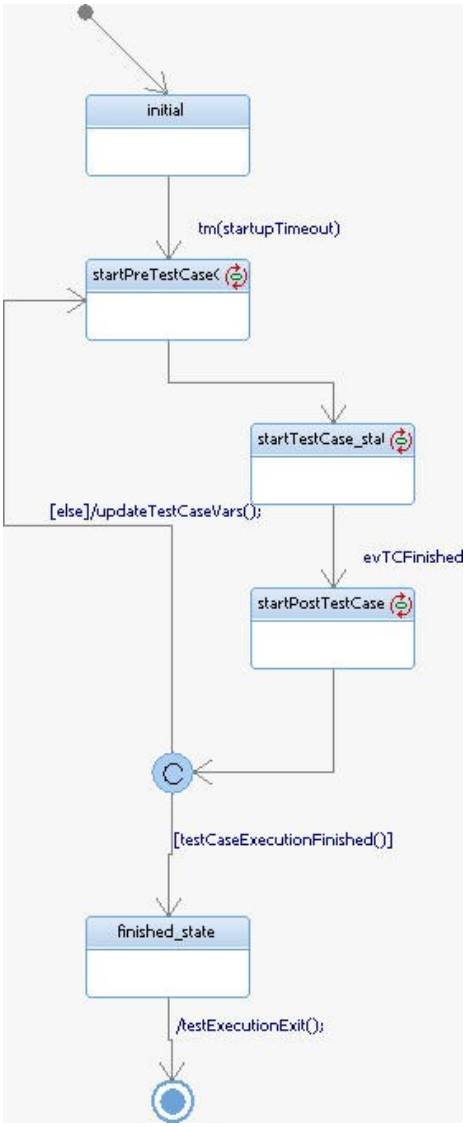


Figure 15: Unmodified TestCase Scheduler Statechart

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:

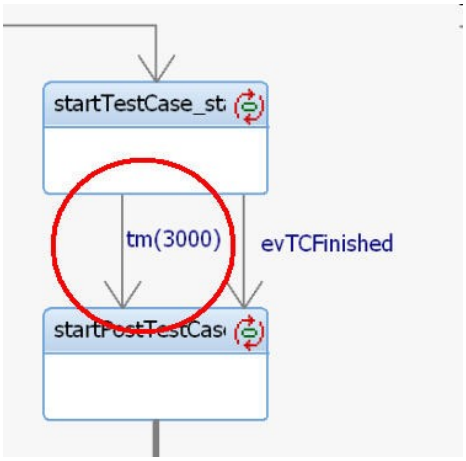


Figure 16: Introducing TestCase Timeout Transition (detail of previous figure)

Normally, a terminating TestCase gives back control to the TestContext, either by just returning as for Code- and FlowchartTestCases or by explicitly invoking the TestContext member operation `finishTestCase()`. This operation executes an assertion indicating normal termination of the actually executed TestCase and sends an `evTCFinished` event to the TestCaseScheduler. After starting a TestCase, the TestCaseScheduler waits this event[36] in order to perform some optional post processing and then to terminate the test application (or to start the next TestCase, if `TestConductor.::TestContext::TestContextExecution_RestartExecutable` was set to False). By adding a timeout triggered transition as depicted in figure 16, the TestCaseScheduler will enter its state startPostTestCaseOp_state after 3000ms, even if the TestCase is still active[37].

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.

[36]If the TestCase got stuck and the test application is not killed from external, then the TestCaseScheduler will wait here forever.
[37]Since the assertion of the TestContext member operation finishTestCase() was not executed, this indication of normal termination is missing in the assertion results. Missing termination indication will be treated as error by the result evaluation.

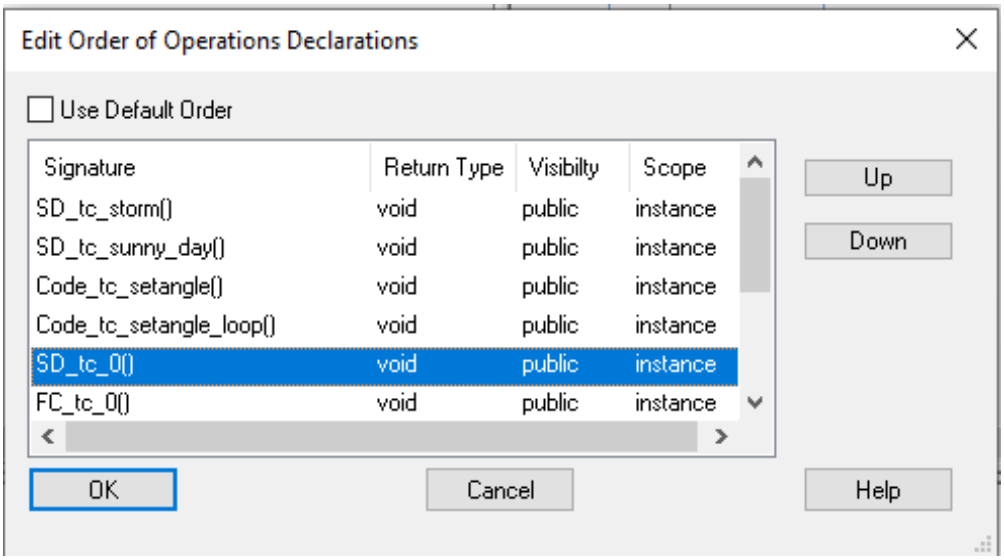


Figure 17: Ordering of TestCases

Per default the TestCases are sorted and executed in alphabetical order.
Besides serving user convenience, the order of TestCases is particularly important for TestContext execution, if property `TestConductor::TestContext::TestContextExecution_RestartExecutable` was set to False on a TestContext.

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.[38]

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.

[38]Note that with the property `TestConductor::Settings::ReportLocation` (see Settings – General Properties on page 128) a user can specify a dedicated report location)

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[39].

- 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.
- 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 the test application.
- 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

TestConductor does not support a dedicated execution timeout concept for the execution of entire TestPackages. We refer to the treatment of timeouts for TestCases (cf. section Execution Timeout on page 86) and for TestContexts (section Execution Timeout for all TestCases of a TestContext on page 89).

[39]Update, build and execute require an adequate code generation configuration to be set active. TestConductor can activate a code generation configuration automatically on invoking update, build or execute on a TestCase/TestSet/TestContext/TestPackage. The rules for code generation configuration activation are described in section TestConfiguration Dependency/Determining the code generation configuration to be used on page 78.
Since TestPackages can be nested and a TestPackage can contain multiple TestArchitectures, the actions described in this section can involve multiple TestArchitectures and, hence, multiple code generation configurations.
In particular, if a TestPackage contains multiple TestArchitectures, the rules for code generation configuration activation play an important role, since different configurations will be activated subsequently to perform an update, build and execute on such a TestPackage grouping several TestArchitectures.

Test Execution Report for TestPackage

When executing a TestPackage, an execution report for each TestContext and TestCase that has been executed is created and added to the model.

Organizing TestCases in TestSets

TestCases can either be extecuted individually or together with all other TestCases belonging to a TestContext or a TestPackage, respectively, by executing a TestContext or a TestPackage.
Moreover, there exist various workflows and use-cases, in which one wants to execute a dedicated set of TestCases, without either executing them explicitly one-by-one or executing an entire TestContext or TestPackage in one sweep. For example, some of the TestCases in one TestContext may not be ready to execute yet. Or TestCases shall simply be grouped according to certain tested features or individual operations even though they test the same SUT and belong to the same TestContext.
TestSets permit the user to group TestCases non-exclusively into subsets of the TestCases in one TestContext.

A TestContext can have arbitrary many TestSets. A TestSet can be added to a TestContext by invoking “Add New->TestingProfile->TestSet” from the context menu on the TestContext.

A TestSet defines a set of TestCases by stereotyped dependencies on the selected TestCases, i.e. dependencies with the new term `<<TestSetElement>>`. To add such a dependency, invoke “Add New->TestingProfile->TestSetElement” from the context menu on a TestSet. This menu item will invoke a small dialog, that offers selection of the particular TestCase. A TestSet may only refer to TestCases of the same TestContext to which the TestSet belongs, referring to TestCases from other TestContexts will be treated as error when attempting to update the TestSet for build and execution.

Optionally, TestSets can also refer to a dedicated code generation configuration of the TestArchitecture. If no such code generation is referred to by a <<TestSetDedicatedConfiguration>> dependency, the TestSet will be updated and executed with the active or activated code generation configuration according to the activation rules (cf. TestConfiguration Dependency/Determining the code generation configuration to be used on page 78).
Hence, TestSets can be used to execute a set of TestCases with different alternative code generation configurations as well as to execute the set always with some dedicated configuration settings.

For TestSets, no particular ordering can be determined. Property `TestConductor::TestContext::TestContextExecution_RestartExecutable` is ignored for TestSet execution, the test application will be restarted for each TestCase of the TestSet.

Note, that update, build and execute for a TestPackage will update, build and execute the TestContexts in the TestPackage by recursively inspecting the TestPackage hierarchy. Update, build and execute on a TestPackage will not consider TestSets in this hierarchy.

TestSet Execution

Starting Test Execution

Execution of a TestSet is kind of batch execution of all TestCases belonging to the TestSet TestSet. All TestCases of the TestSet will be executed subsequently.

- Right-click on the TestSet and select 'Update TestSet' from the context menu**.** This updates all necessary driver and StubOperations derived from the defined sequence diagram TestCases belonging to the TestSet.
- Right-click on the TestSet and select 'Build TestSet' from the context menu**.** This re-generates the necessary code for all elements of the TestArchitectures and starts the compile and link process of the test application.
- Right-click on the TestSet and select 'Execute TestSet' from the context menu**.** This starts the execution of all TestCases belonging to the TestSet.

Executing a TestSet is almost like the execution of a TestContext, except for leaving out the TestCases not belonging to the TestSet.

Stopping Execution

The **abort icon** in the test execution window aborts the execution of the TestSet.

Execution Timeout

TestConductor does not support a dedicated execution timeout concept for the execution of TestSets. We refer to the treatment of timeouts for TestCases (cf. section Execution Timeout on page 86).

Test Execution Report for TestSet

After the execution of all TestCases belonging to the TestSet, the execution report is written into an HTML file. This file is added to the TestSet as a controlled file.[40] A report for each TestCase that has been executed is 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, TestSet 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.

[40]Note that with the property `TestConductor::Settings::ReportLocation` (see Settings – General Properties on page 128) a user can specify a dedicated report location)

Specification of Invariants and Verification of Invariants during Test Execution

Orthogonal to the definition and execution of individual TestCases, TestConductor supports the definition of invariant expressions and their verification w.r.t. the individual TestCases or for all test executions of an entire TestContext.

Verification of invariants is supported for C++, C and Java when using the assertion based testing mode. Verification of invariants can be enabled by checking the tag "`EnableInvariantVerification`" on the code generation configuration (cf. Tags of the <<AssertionBasedTestingConfiguration>> and <<TestingConfiguration>> Stereotypes on page 69) used for testing.

Please note, when using invariant check functions in flowchart or statechart based TestCases, the result verification (see Performing result verification for TestCase execution on page 80) cannot relate the assertions of the invariant check function to the verification file of the diagram and reports an error. This means, if invariant verification is used in flowchart or statechart based TestCases, result verification should be switched off.

Optionally, an additional profile providing helpers for easier definition of invariant check functions can be loaded into the model. The '`InvariantCheckProfile`' can be found in the Share/Profiles directory of your Rhapsody installation. You can either add it by 'File->Add Profile to Model' or 'File-Add to Model' menu entry of Rhapsody – in the opening file selector dialog choose package-file "`Share\Profiles\InvariantCheckProfile\InvariantCheckProfile_rpy\InvariantCheckProfile.sbsx`" to be added to the model.
On loading the profile package, Rhapsody will find the associated helpers-file of the profile and offer additional invariant checking related helpers in the context menu of the browser.

Defining and using invariant check functions

Invariant check functions can be created in a test context by right clicking the test context in the browser and then invoking '`Add New ->Testing Profile -> Invariant Check Function`' from the context menu.

The check(s) can then be implemented in the body of the function using the language specific TestConductor assert macro or function (for a list of available assertions cf. appendix TestConductor Assert Macros (C/C++) and TestConductor Assertion Functions (Java) on page 200ff):

- `RTC_ASSERT_NAME` for C++ and C
- `TestConductor.ASSERT_NAME` for Java

These assertions allow defining a string message and an expression. The string message is shown in the test execution window and in the html test result once for each execution of the assert and allows identifying the assert and can be used to visualize data from the assert. The expression is evaluated for the result of the assert.

The body of an invariant check function should consist of one or more assertions[41], each checking an individual expression whenever the invariant check function is invoked.

[41]When using the 'Invariant Check UI' for defining and editing invariant check functions, only one assertion per invariant check function is supported by the editor. If more than one assertion in the body of an invariant check function are used, the editor of the features dialog should be used instead of the 'Invariant Check UI'.

After defining an invariant check function it can be applied by one or more test cases by adding an <<AppliedInvariantCheck>> dependency from the invariant check function to the TestCase. Alternatively the dependency can be drawn to the TestContext, meaning the invariant check function is applied by all test cases of the TestContext.

The Testing Profiles offers also invariant check matrices or invariant check tables as a convenient way to manage which invariant check functions are applied by which TestCases. Right click on the owning TestPackage of the TestContext and select 'Add New -> Testing Profile -> Invariant Check Matrix/Table', then adjust the scope of the created element and open it. The matrix or table shows all invariant check functions and their <<AppliedInvariantCheck>> dependencies and allow adding or removing <<AppliedInvariantCheck>> dependencies from invariant check functions to individual TestCases or to the entire TestContext.

When executing TestCases the results of the executed assertions from the applied invariant check functions is shown in the TestConductor test execution UI and in the html TestResult. If such an assertion fails then the TestCase is failed and the TestCase terminates. A witness scenario shows the location of the failed assertion by coloring the preceding message in red, this indicates when the failure happened but it does not necessarily imply the red message has directly caused the failure: A failed check of an an invariant could also be caused by internal activity in the SUT not related to the message.

The verification of invariants is based on an instrumentation of the driver and stub operations and of the code of the SUT[42], ensuring a high frequency of invocations of the applied invariant check functions both in the test harness and in the SUT.

The instrumentation is performed after the code has been generated by Rhapsody during the build of the test application. When testing external code, i.e. code that has been generated outside of Rhapsody, then only the driver and stub operations are instrumented with calls of the applied invariant check functions.

Helpers provided by the invariant check profile

For easier definition the body of invariant check functions, TestConductor contains a helper profile with a Java plugin providing additional context menu helpers which are available after adding the profile to a model:

- Copy Attribute Check – applicable to Attribute, ValueProperty
  When invoking "Copy Attribute Check", the helper generates an assertion to compare the attribute with a particular value and puts it into the copy/paste buffer of the system. The generated string can be pasted into the body of an invariant check function and modified, like editing the information string of the assertion or adjusting the expected value of the attribute.

- Copy In-State Check – applicable to State, AcceptEventAction, AcceptTimeEvent, CallBehavior, CallOperation, ActivityFinal, ObjectNode
  When invoking "Copy In-State Check" the helper generates an assertion to check if the statechart (or activity) currently is in the state (or action etc.) and puts

If the Invariant Check UI is invoked on an invariant check function with more than one assertion, the editor issues an error 'UI does only support one assertion in an InvariantCheckFunction' and refuses to edit the selected function.

[42]The invariant check function associated with a TestCase by <<AppliedInvariantCheck>> dependency is invoked in all driver operations, all stubs and observation instrumentations as well as on invoking any operation of the SUT and on taking any transition of the SUT statechart or control flow of SUT activity.

it into the copy/paste buffer of the system. The generated string can be pasted into the body of an invariant check function and modified.

- Invariant Check UI – applicable to InvariantCheckFunction

Please note, this helper supports creating basic checks or a starting point for defining complex checks. It is not intended create all kinds of checks fully automatically and it will not create a valid check expression for all circumstances. The user needs to inspect and adjust the expressions generated by the helper.
For example, if two orthogonal states own same named sub states, then the "Copy In-State Check" helper will generate the same expression for both of the sub states. Also, the "Copy Attribute Check" helper does not support all kinds of modifiers to access an attribute.

The 'Invariant Check UI' helper of this profile offers a UI to define an invariant check function. After invoking "`Invariant Check UI`" on an invariant check function the UI opens and allows to define the body of the function in two separate edit fields.

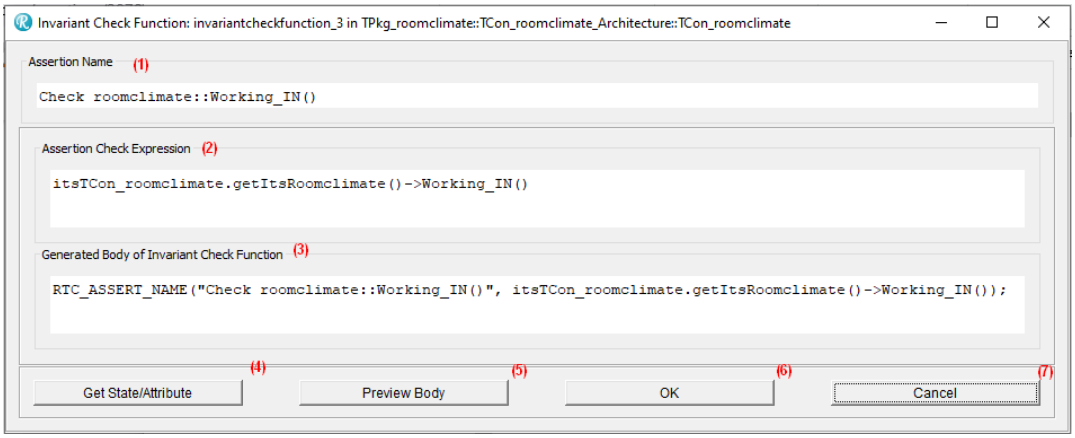


Figure 18: Invariant Check UI

Below "`Assertion Name`" (1) the name of the assertion can be entered, i.e. an identifying string for the test execution UI and for the test execution result.

Below "`Assertion Check Expression`" (2) the check expression of the assertion can be entered; when clicking the `Get State/Attribute` button (4), the helper generates a check condition for the selected element and adds it to the assertion expression at the end of the current content. Sub-expressions, i.e. boolean terms, can be combined using boolean operators like `&&`, `||`, `!` and parenthesis ( ) can be used for grouping. The UI does not perform a syntax check of the expression.

The title of the UI shows the name of the currently edited invariant check function and the navigation expression to it.

When clicking the `Ok` button (6), the helper generates the complete body for the invariant check function including the assert macro/function and writes it to the body of the invariant check function, overwriting any existing content, and closes the UI.

When clicking the `Cancel` button (7), the UI closes without changing the invariant check function.

When clicking on the `Preview Body` button (5), the helper generates the complete body for the invariant check function including the assertion macro/function and writes it to the generated `Body of Invariant Check Function` field.

"Copy In-State Check" helper and "Copy Attribute Check" helper as well as the Get State/Attribute button of the Invariant Check UI compute navigation expressions from the test context to attributes or states according to the test architecture denoted by the currently active code generation configuration.

Computing Model Coverage during Test Execution

(see also Sample-models:

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

)

When executing TestCases, i.e., either individual TestCases, a TestSet, 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 <<AssertionBasedTestingConfiguration>> stereotype (the <<TestingConfiguration>> stereotype, respectively) provides tag "ComputeModelCoverage" for switching on model coverage measurement.

If the tag is switched on for such a stereotyped code generation configuration, then each TestCase, TestSet, TestContext and TestPackage execution for this configuration will generate a model coverage report in addition to the execution report. In contrast to code coverage, model coverage results will be generated for each individual TestCase even though an entire TestSet, TestContext or TestPackage is executed. The model coverage report will be added to the model in the same location as the execution report[43]. 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 user-defined *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 Scope for Model Coverage on page 101.).

For C++ and SysML, also token oriented activity realizations of operations are considered in model coverage.

All vertexes and transitions contained in a behavioral diagram are considered. If a coverage item is marked as 'covered' this means that the corresponding code generated for

[43]Note that with the property TestConductor::Settings::ReportLocation (see section Settings – General Properties on page 128) a user can specify a dedicated report location)

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[44]. The coverage information is according to the model view, there is no information about how much of the user code has been traversed, but only information whether a particular model element was used or activated or not. For a code view with detailed information about the coverage of the generated and the user code you need to use code coverage (cf. section Computing Code Coverage on page 109).

For flowchart realizations of operations, a dedicated model coverage of the flowchart realization can be measured. Operations realized using flowcharts will be treated differently from ordinary operations. Rhapsody offers code generation of flowcharts for C and C++/SysML models. The code generation generates highly efficient operation bodies from flowcharts, only considering decisions and actions with associated statements. Empty actions, as well as control flow among them will be optimized away in order to obtain a concise realization of the modelled functionality. Control flows and decisions are not directly represented in the generated operation bodies, but define the structure of the generated code in terms of for, while, if-then-else and goto statements.
Accordingly, TestConductor leaves away empty actions without associated statements from the model coverage or marks them as not measurable in the report, respectively. Model coverage does not deduce control flow coverage of flowcharts, but measures the coverage of the essential actions only in the flowchart.

Please note: If local variables are used in a flowchart of an operation, then the instrumentation by TestConductor adds function calls before the local variable declaration. This causes compile errors when using the C89 or C90 standards (for example, VS 2012 uses C89 per default) so in this case model coverage cannot be computed.

ModelCoverage of hierarchical states

In statecharts, states can be plain states or hierarchical states. A hierarchical state contains a subchart either on-page or off-page. A plain states is covered according to the intuition if it was entered via some transition. For hierarchical states the situation is different. Since a hierarchical state can potentially be entered and exited without taking all transitions or entering all states in its subchart, ModelCoverage distinguishes partial and full coverage of hierarchical states. A hierarchical state is measured as uncovered, if it was not entered at all. It is partially covered, if some but not all of the elements in its subchart were covered. It is fully covered only if all the model elements of its subchart were covered.

A similar distinction is made for sub activities of state oriented activities.

ModelCoverage of states with internal transitions

In statecharts, plain states as well as hierarchical states can be source or target of transitions. If a particular transition is taken in an execution, its source state is exited and the target state of the transitions is entered. Exiting and entering states is observed and measured by ModelCoverage measurement.
In addition to such explicit transitions, statecharts also offer internal transitions in states. Internal transitions don't leave and reenter a state, but allow for processing events while staying in the state.
TestConductor measures internal transitions in model coverage. A basic state with internal transitions will be measured as uncovered if the state is not entered at all. The basic state

[44]For some statechart and activity elements which are directly dependent of other elements Rhapsody does not generate a code representation. For these elements TestConductor applies a set of dependency rules to derive the coverage.

will be measured as partially covered, if the state is entered, but not all of its internal transitions are taken. The state will be measured as fully covered, if the state is entered and all its internal transitions are taken.

Choosing the Coverage Scope for Model Coverage

For determining the coverage scope for model coverage, TestConductor offers three tags on the tags tab of the features dialog of `<<AssertionBasedTestingConfiguration>>` code generation configurations (or `<<TestingConfiguration>>`code generation configuration, respectively):

- CoverageScopeIncludeParts: By default, the global SUT objects and SUT parts of the TestContext as well as their parts will be instrumented for model coverage. TestComponents and TestContext are by default not considered - this can be changed by tag “CoverageScopeIncludeTestArchitecture” (see next item). In some cases, it might e desired to compute model coverage only for the global SUT objects themselves or, respectively, only for the direct SUT parts of the TestContext, without taking their parts into consideration. If tag “CoverageScopeIncludeParts” is unchecked in the tags tab of the features dialog for the respective code generation configuration, model coverage instrumentation will only affect the direct parts and global objects, respectively, without their internal parts, according to the set of model elements pre-selected by “CoverageScopeIncludeTestArchitecture”. The default setting of tag “CoverageScopeIncludeParts” ensures, that always the entire hierarchy of internal objects of the set of model elements chosen for model coverage scope is regarded.
- CoverageScopeIncludeTestArchitecture: By default, only the model elements belonging to the SUT are considered in model coverage. TestComponents and TestContext will not be considered in model coverage. If also TestComponents and TestContext shall be considered in model coverage, tag “CoverageScopeIncludeTestArchitecture” has to be checked.
- “CoverageScopeIncludeInheritance”: Base classes and inheritance are by default not regarded by model coverage. If this is desired, checking “CoverageScopeIncludeInheritance” will extend the scope of model coverage, such that also all generalizations and realizations of model elements in the coverage scope will be regarded by model coverage.

Despite choosing the coverage scope using these three tags 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

In Rhapsody 9.0.1 and up, TestConductor does no longer require Rhapsody animation to determine the coverage of model elements. If tag “ComputeModelCoverage” is checked in the tags tab of the features dialog of the active `<<AssertionBasedTestingConfiguration>>` code generation configuration or

`<<TestingConfiguration>>` code generation configuration, respectively. TestConductor will instrument the generated source code with additional instrumentation code for model coverage measure.

There are some elements for which the Rhapsody code generation does not generate explicit representations in the source code (e.g. transition fragments in transition chains with junction connectors only the flattened transitions can be identified in the source code, the code of other transition fragments is included in the code block of the flattened transition). For these scenarios TestConductor applies a set of dependency rules to derive the coverage of these elements from the coverage of elements that can clearly be identified in the generated source code.

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?”

)

This section regards RequirementCoverage measurement based on ModelCoverage and traceability dependencies of model elements on requirements. Besides this ‘dynamic’ approach, TestConductor also supports defining and maintaining obligations for requirement based testing by so called TestObjectives (cf. Linking TestCase to Requirements on page 120).

Computing Requirement Coverage for TestCases and TestContexts

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

Requirement coverage measurement is enabled by setting both tags `ComputeModelCoverage` and `ComputeRequirementCoverage` on the `<<AssertionBasedTestingConfiguration>>` code generation configuration or `<<TestingConfiguration>>` code generation configuration, respectively. Computing requirement coverage is supported for requirements directly in the Rhapsody model and also for so called remote requirements from DOORS Next Generation (DNG) or another OSLC based requirement management tool.

Precondition for measuring requirements coverage by individual TestCases, TestSets 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.

TestConductor regards such dependencies (or OSLC links when using remote requirements) in order to calculate requirement coverage based upon model coverage information. The user can define the stereotypes on dependencies 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. When using this option dependencies without any of the listed stereotypes are not considered for computation of requirement coverage.

TestConductor provides also two properties for configuring requirement coverage scope for TestConductor. The packages (and their sub-packages), from which requirements shall be regarded in requirement coverage calculation can be defined using property `'ModelBasedTesting::Settings::RequirementCoverageRequirementsScope'` of the code generation configuration. Multiple packages can be defined using a comma separated list of fully qualified package paths, e.g. `"RequirementsAnalysisPkg::RequirementsPkg::SecSysReqs,TestPkg::RequirementsPkg::SecSysTestReqs"`. When using this option requirements located outside of the listed packages (or their sub-packages) are ignored.
Property `'ModelBasedTesting::Settings::RequirementCoverageRegardedTags'` (on code generation configuration) can be used to specify names and values of tags applied on requirements for selecting requirements to be regarded in coverage calculation. Multiple pairs of tag name and value can be specified using a comma separated separated list, e.g. `"RequirementType=functional,RequirementType=additional"`. When using this option requirements without any of the listed pairs of tag names and values are ignored.

Also the model elements scope for requirement coverage calculation can be specified, i.e. restricted. Packages (and their sub-packages) for which model elements shall be regarded in requirement coverage calculation can be specified in property `'ModelBasedTesting::Settings::RequirementCoverageModelElement`

sScope' of the code generation configuration. A comma separated list can be used for multiple packages, e.g. "DesignSynthesisPkg::SecSysControllerPkg::SecSysController,ActorPkg::CardReaderEntry". When using this option model elements located outside of the listed packages (or their sub-packages) are ignored.

Property 'ModelBasedTesting.Settings.RequirementCoverageExcludedMetaClasses' of code generation configuration can be used to specify tags of model element meta classes to be excluded from requirement coverage consideration. Multiple meta classes can be excluded from consideration using a comma separated list, e.g. "Attribute,Class,Event".

TestConductor distinguishes three grades of requirement coverage by TestCases:

- **full coverage**

  All source model elements of relevant dependencies on a particular requirement (w.r.t. specified dependency stereotypes, excluded metaclasses of model elements and requirement model element scope) are fully covered by a TestCase, TestSet or TestContext. The TestCase, TestSet or TestContext then `fully` covers the requirement. For dependencies of hierarchical states (or subcharts) and basic states with internal transitions all relevant model elements in the (hierarchical) state must be covered by model coverage to regard this dependency for full requirement coverage. If at least one relevant model element in the hierarchical state is not covered by model element coverage, the requirement will only be partially covered.

- **partial coverage**

  At least one but not all source model elements depending on a particular requirement (w.r.t. specified dependency stereotypes, excluded metaclasses of model elements and requirement model element scope) are covered by a TestCase, TestSet or a TestContext. The TestCase, TestSet or TestContext then only `partially` covers the requirement. Also if hierarchical states or basic states with internal transitions are not fully covered, the referenced requirement will only be measured partially covered.

- **uncovered**
  No source model element depending on a particular requirement (w.r.t. specified dependency stereotypes, excluded metaclasses of model elements and requirement model element scope) is covered by a TestCase, TestSet or TestContext. The TestCase, TestSet or TestContext then does not at all cover the requirement.

Transitivity of Dependencies (Refinement of model elements and requirements)

Property 'ModelBasedTesting::Settings::RequirementCoverageTransitivityOfDependencies' allows the user to switch support of transitivity (exploiting refinement relations of model elements and refinement relations of requirements) on or off. (Default: off)

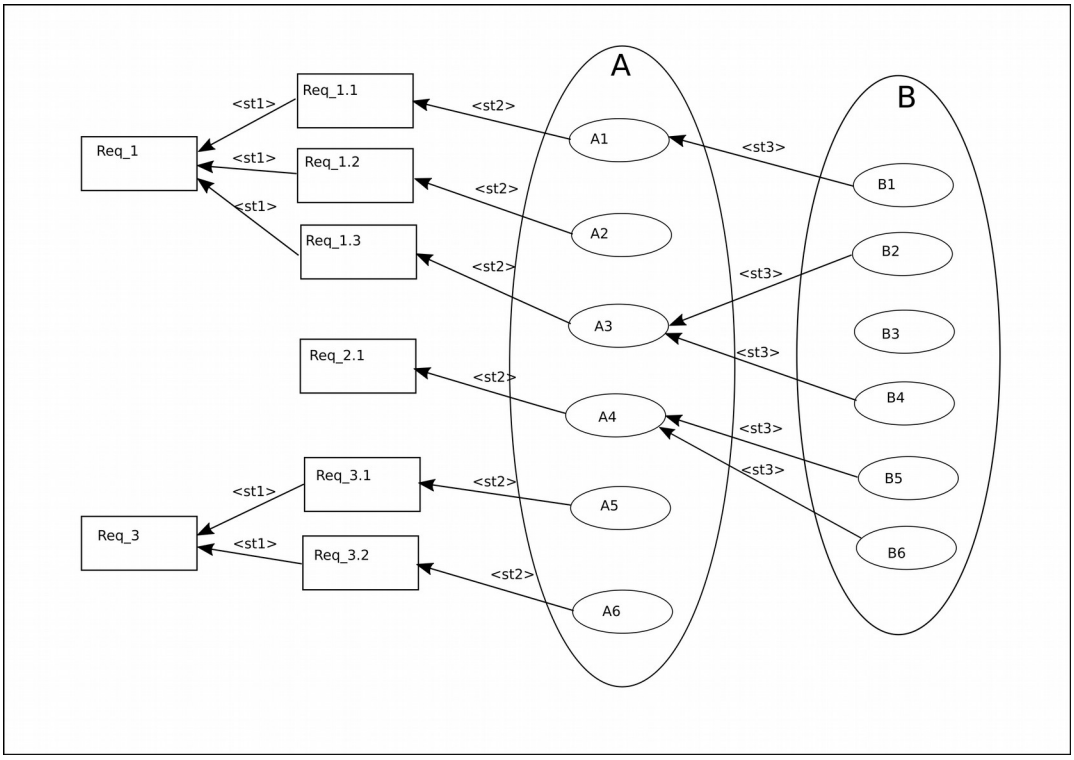


Figure 19: Transitivity of Dependencies (Refinement of model elements and requirements)

The figure above shows an application for the refinement of requirements and model elements: boxes denote requirements in the model, arrows denote stereotyped dependencies. For example requirement `Req_1` is referenced by Requirements `Req_1.1`, `Req_1.2` and `Req_1.3` using dependencies with stereotype `st1`.
Ellipses annotated with `A1`, `A2`,… denote individual model elements, such as statechart states, transitions, operations, etc. Inthe figure, e.g. `A3` has a dependency stereotype `st2` on requirement `Req_1.3`. `A3` itself is referenced by `B2` and `B3` using dependencies stereotyped `st3`. For simplicity, we use an homogenous set of '*requirement refinement*' dependencies (stereotyped `st1`), a homogenous set of '*satisfy*' dependencies (stereotyped `st2`) and a homogenous set of '*model refinement*' dependencies (stereotyped `st3`) in this example. The set of stereotypes to be regarded by the algorithm is determined by property `ModelBasedTesting::Settings::StereotypesForDependenciesToRequirements` (cf page 103).
Let `A` and `B` denote code generation scopes with individual model elements `A1`,..,`A6` and `B1`,…,`B6`.
TestCases are executed w.r.t. to a code generation scope, i.e. the code generation configuration using which the test application has been build only regards model elements selected in the scope of its code generation component. When a test application is build with scope `A`, only model elements in this scope are regarded in model element coverage. In particular, model elements of code generation scope `B` are in general not regarded in model element coverage when executing a TestCase with code generation scope `A`.
Note that code generation scope and requirement coverage scopin general differ: while code generation scope is determined by the code generation component, requirement coverage scope by default comprises of the entire model unless restricted using properties

ModelBasedTesting::Settings::RequirementCoverageModelElementsScope (cf page 104) and ModelBasedTesting::Settings::RequirementCoverageModelElementsScope (cf page 104).

Example 1: If transitive evaluation of dependencies is switched off, requirement coverage of Req_1.1, Req_1.2 and Req_1.3 can only be achieved by executing a TestCase for code generation scope A. A1 refers to Req_1.1, A2 refers to Req_1.2 and A3 refers to Req_2.3. Hence, requirement coverage of Req_1.1, Req_1.2 and Req_1.3 depend on model coverage results for A1, A2 and A3, respectively. Requirement Req_1 can not be covered at all, since it is not directly referenced by any model element regarded by model coverage.

Switching on transitive evaluation of dependencies allows for consideration of requirement coverage also for Req_1 (if stereotype st1 belongs to the regarded stereotypes according to ModelBasedTesting::Settings::StereotypesForDependenciesToRequirements). Requirement Req_1 is only fully covered by a TestCase or a TestContext, if the requirements Req_1.1, Req_1.2 and Req_1.3 are as well fully covered by this TestCase or TestContext (if Req_1 is in the requirements scope according to ModelBasedTesting::Settings::RequirementCoverageRequirementsScope (cf page 103)). If at least one of the requirements Req_1.1, Req_1.2 and Req_1.3 is only partially covered, also Req_1 will only be partially covered. Req_1 will be uncovered if requirements Req_1.1, Req_1.2 and Req_1.3 are entirely uncovered.

Executing a TestCase for code generation scope B with transitive evaluation of dependencies switched off will not yield a meaningful requirement coverage, since none of the model elements in B directly refer to any requirement.

But if transitive evaluation of dependencies is switched on, the refinements of A1 by B1 and of A3 by B2 and B4 are considered during the requirement coverage calculation. E.g. requirement Req_1.3 is only fully covered by a TestCase if both model elements B2 and also B4 are covered by this TestCase (if stereotypes st2 and st3 belong to the regarded stereotypes according to ModelBasedTesting::Settings::StereotypesForDependenciesToRequirements and A3 does not belong to the requirement coverage scope according to ModelBasedTesting::Settings::RequirementCoverageModelElementsScope (cf page 104)). Note that A3 does not belong to the code generation scope B and can hence not appear in the model coverage. If A3 belonged to the requirement coverage scope, then all requirements referenced by A3 could at most be partially covered since A3 does not contribute to requirement coverage directly but only indirectly via transitive evaluation w.r.t. model elements in scope B referring to A3.
Property ModelBasedTesting::Settings::RequirementCoverageModelElementsScope determines the base set of model elements. Starting in this set chains of dependencies will be transitively evaluated for requirement coverage.

If transitive evaluation of dependencies is switched off, also the dependencies of requirements Req_1.1, Req_1.2 and Req_1.3 on requirement Req_1 will not be considered. But if transitive evaluation of dependencies is switched on requirement Req_1 will be considered due to the dependencies of requirements Req_1.1, Req_1.2 and Req_1.3 on Req_1.

Example 2: A4 has a st2 dependency to requirement Req_2.1. B5 and B6 have both a st3 dependency to the model element A4. If the property ModelBasedTesting::Settings::RequirementCoverageTransitivityOfDependencies is set and ModelBasedTesting::Settings::RequirementCoverageModelElementScope is set to scope B, then the requirement Req_2.1 will be fully covered by a TestCase, if the TestCase covers all of the model elements B5 and B6. If the TestCase covers only one of the model elements B5 and B6, then the requirement Req_2.1 will only be partially covered.

Example 3: Model element A5 has a st2 dependency on requirement Req_3.1, which has in turn a st1 dependency on requirement Req_3. Model element A6 has a st2 dependency on requirement Req_3.2, which in turn has a st1 dependency on requirement Req_3.
If property ModelBasedTesting::Settings::RequirementCoverageTransitivityOfDependencies is set and requirement Req_3 is in the requirement scope, then requirement Req_3 will be fully covered by a TestCase, if this TestCase fully covers both requirements Req_3.1 and Req_3.2. Req_3 will only partially be covered by this TestCase, if requirement Req_3.1 or Req_3.2 is only partially covered.

Requirement Coverage Results

Other than model coverage and code coverage results, requirement coverage results are not presented as a html report, but are available as part of the model directly in the browser. Below a RequirementCoverageResult model element, the requirement coverage information is represented by a set of dependencies with tags:

- <<consideredTestCase>> (stereotyped dependency):
  dependencies on each TestCase contributing to the owning requirement coverage result.
- ExecutedElement (new term on dependency):
  link to the TestCase/TestSet/TestContext/TestPackage that has been executed to obtain the owning requirement coverage result.
  Note that merged requirement coverage results (cf. Merging requirement coverage reports on page 175 ff) do not have this dependency.
- Coverage information (dependencies on requirements):
  The essential requirement coverage information (for definition of coverage grades see page 104) is represented using dependencies termed[45]:
  - fully: for all requirements fully covered by the set of considered TestCases a dependency termed <<fully>> on the requirement is added to the Requirement Coverage Result Report of the TestCase or TestContext.
  - partially: for all requirements partially covered by the set of considered TestCases a dependency termed <<partially>> on the requirement is added to the Requirement Coverage Result Report of the TestCase, TestSet or TestContext.

[45] <<fully>>, <<partially>> and <<uncovered>> are *new term* stereotypes – applicable to dependency.

- `uncovered`: for all requirements which are uncovered by the set of considered TestCases a dependency termed <<`uncovered`>> on the requirement is added to the
  Requirement Coverage Result Report of the TestCase, TestSet or TestContext.

In case of remote requirements above mentioned termed dependencies do not directly point to the remote requirement, instead they point to a proxy element in the model (RemoteRequirementRef) with an OSLC trace link to the remote requirement.

Each of these `fully`, `partially` and `uncovered` dependencies provides detailed information about the considered, covered, uncovered and unregarded model elements contributing to this partial result:

- **considered_mes**: list of considered/relevant model elements for this dependency, i.e.:
  - the model elements which belong to the model element scope (cf property `ModelBasedTesting::Settings::RequirementCoverageModelElementsScope`) and
  - are related with the particular requirement by dependencies stereotyped according to property `ModelBasedTesting::Settings::StereotypesForDependenciesToRequirements` (default <<`trace`>> or <<`satisfy`>>).
- **covered_mes**: list of covered model elements (subset of considered_mes) that contribute to the particular grade of coverage of the particular requirement.
- **uncovered_mes**: list of uncovered model elements (subset of considered_mes) which were needed to cover the respective requirement but are not covered by the TestCase/TestSet/TestContext/TestPackage.
- **unregarded_mes**: list of model elements (subset of considered_mes) which remained unregarded for certain reasons, e.g.:
  - because their metaclass has been excluded from requirement coverage computation (property `ModelBasedTesting::Settings::RequirementCoverageExcludedMetaClasses`) or
  - because they are not supported by requirement coverage, such as for example attributes or ports.

- TestConfiguration (new term on dependency): dependency on the active code generation configuration at creation time of the result. Note that merged requirement coverage results (cf. Merging requirement coverage reports on page 175 ff) do not have this dependency.
- Tags:
  - **covered_modelelements**: List of full pathnames of model elements covered by the set of considered TestCases
  - **date**: creation date/time of the requirement coverage result.

- **regarded_excludedmetaclasses**: contents of property `ModelBasedTesting::Settings::RequirementCoverageExcludedMetaClasses` at creation time of the result.
- **regarded_modelelementscope**: contents of property `ModelBasedTesting::Settings::RequirementCoverageModelElementsScope` at creation time of the result.
- **regarded_requirementscope**: contents of property `ModelBasedTesting::Settings::RequirementCoverageRequirementsScope` at creation time of the result.
- **regarded_requirementtaglist**: contents of property `ModelBasedTesting::Settings::RequirementCoverageRegardedTags` at creation time of the result.
- **regarded_stereotypes**: contents of property `ModelBasedTesting::Settings::StereotypesForDependenciesToRequirements` at creation time of the result.
- **regarded_transitivityofdependencies**: value of property `ModelBasedTesting::Settings::RequirementCoverageTransitivityOfDependencies` at creation time of the result.

Representing requirement coverage results using model elements directly in the model instead of presenting the result in a html report allows for accessing and processing the information in e.g. tables and matrices. The TestingProfile provides some predefined matrix and table layouts for concise result representations and aggregation (see section TestDocumentation Package on page 197 ff).

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, TestSets and TestContexts (for C and C++ only). In order to turn on code coverage, the tag "`ComputeCodeCoverage`" of the `<<TestingConfiguration>>` code generation configuration 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 a 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.

The same as for model coverage, the coverage scope of code coverage measure can be determined by the tags CoverageScopeIncludeParts, CoverageScopeIncludeTestArchitecture and CoverageScopeIncludeInheritance. For details, see previous section Choosing the Coverage Scope for Model Coverage on page 101.

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".

In contrast to model coverage, a code coverage result will be generated only for the entire TestSet or TestContext if 'Execute TestContext', 'Execute TestSet' or 'Execute TestPackage' is executed. In order to obtain a code coverage result for an individual TestCase, the TestCase needs to be executed separately by 'Execute TestCase'.

(See also document <Rhapsody install>/Doc/pdf_docs/CodeCoverage_Limitations.pdf for details regarding code coverage measurement).

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) != 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 != 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 ^ 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)* ..., 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 >= 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:

i	Specification i > 5	Implementation i >= 5
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 <expr1> <relop> <expr2> where <expr1> and <expr2> are arbitrary

expressions and <relop> is a relational operator (<, <=, ==, !=, >=, >).
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: (expr1) – (expr2) > 1 (operation became true)
- P2: (expr1) – (expr2) < -1 (operation became false)
- P3: (expr1) – (expr2) == -1 (boundary check)
- P4: (expr1) – (expr2) == 0 (boundary check)
- P5: (expr1) – (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 (only on Windows, TestConductor does not support rhapsodycl.exe on Linux).

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 TestSet, a TestContext or a TestPackage.

TestConductor logs in the file “cl.log” in the project folder the command line actions together with the result[46] (SUCCEDED, FAILED or ERROR[47] for actions, TRUE, FALSE or ERROR 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.

Note: When using rhapsody.exe also the -hiddenUI option can be used to run Rhapsody and TestConductor with a hidden user interface. This is supported for Windows and Linux.

Examples:

- `"<full Rhapsody path>\rhapsody.exe" -cmd=open <path to Rhapsody samples>\CppSamples\TestConductor\CppTestActions\CppTestActions.rpy -cmd=call "rtc update_build_execute TPkg_Calc::TCon_Calc_Architecture::TCon_Calc::SD_tc_0" -cmd=save`
  updates, builds, and then executes the TestCase “SD_tc_0” of the sample model CppTestActions. After test execution the model is saved, Rhapsody is not terminated.
- `"<full Rhapsody path>\rhapsody.exe" -cmd=open <path to Rhapsody samples>\CppSamples\TestConductor\CppTestActions\CppTestActions.rpy -cmd=call "execute TPkg_Calc::TCon_Calc_Architecture::TCon_Calc" -cmd=save`
  executes the TestContext `"TCon_Calc"` of the sample model CppTestActions. After test execution the model is saved, Rhapsody is not terminated.
- `"<full Rhapsody path>\rhapsody.exe" -cmd=open <path to Rhapsody samples>\CppSamples\TestConductor\CppTestActions\CppTestActions.rpy -cmd=call "rtc build_execute TPkg_Calc" -cmd=save`
  builds and executes the TestPackage ”TPkg_Calc” of the sample model CppTestActions. After test execution the model is saved, Rhapsody is not terminated.

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 TestConducror commands are invoked by specifying

- `-cmd=call "rtc TC_COMMAND TC_ELEMENT"`

[46]In the format <command> <parameter> : <result>.
[47]After the keyword ERROR a description about the problem will be given in parentheses.

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

Examples (we assume that rhapsodycl.exe is already started and the model has been opened):

- `"> -cmd=call "rtc update_build_execute TPkg_Calc::TCon_Calc_Architecture::TCon_Calc::SD_tc_0"` updates, builds, and then executes the TestCase "`SD_tc_0 `" of the sample model CppTestActions.
- `"> -cmd=call "execute TPkg_Calc::TCon_Calc_Architecture::TCon_Calc"` executes the TestContext "`TCon_Calc`" of the sample model CppTestActions

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" on page 81 ff, "TestSet Execution" on page 95 ff, "TestContext Execution" on page 88 ff and "TestPackage Execution" on page 93 ff.

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)

Providing Compiler Builtin-Type Information for User Defined Types

For code generation and build it is often sufficient to just make types visible in the Rhapsody model. E.g. adding an external package (Property CG::Package::UseAsExternal **checked on the package**) or using an external type defined in a profile (or type with property CG.Type.Generate **unchecked**) and then to add language types with empty declaration or typedef types with undefined details as for example `uint16_t` with appropriate settings of properties {Lang}`_CG::Type::{In,Out,Inout,TriggerArgument}`. If `stdint.h` is included as standard header by the code generation configuration or by particular classes, the used compiler will recognize occurences of the 'external' type in the code by the type's name and compile and build the code without complaints.

From the model perspective and from the model transformations which TestConductor performs in order to generate e.g. driver operations and stubs, such an external type lacks necessary or at least important definition details. TestConductor only ‘fully’ knows the predefined types and the language dependent predefined types as well as user defined types that are defined on their basis as arrays, records, typedefs and so on. A partially defined ‘external’ type lacks information about cast compatibility and access rules, e.g. which relational operations are applicable. For example, an integer based type can in general be compared using “==” etc. whereas pointer based types need e.g. `memcmp()` or other compare operations.
Unfortunately, it can not be requested that such external types have to be completely defined: compiler builtin types such as `int` or `long` have compiler dependent sizes and will vary from compiler to compiler. Therefor, TestConductor supports a mechanism that allows for code generation configuration specific additional type informations. By adding a TypeMappingC or TypeMappingCpp, respecitvely, to the code generation configuration, depending on a particular (external) type, information about the compiler builtin base type of the external type can be provided.
From the code generation configuration invoke the context-menu. Invoke ‘Add New->TestingProfile->TypeMappingC’, or ‘Add New->TestingProfile->TypeMappingCpp’. A dependency dialog opens to select the target of the type mapping. In the features dialog of the type mapping, on the ‘Tags’ tab, tag TestArchitecture::TypeMappingC or TestArchitecture::TypeMappingCpp can be set from a drop-down list of predefined builtin types.

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 Rhapsody Publishing Engine (see section Generating Test Reports with IBM Engineering Lifecycle Optimization - Publishing on page 132) for report generation regarding requirement coverage by TestCases according to TestObjective relations.

Note, that TestObjectives linkage to requirements only works for requirements in the model. OSLC links currently only allow for using stereotypes derive, refine, trace and verify.

Besides establishing traceability from TestCases to requirements using TestObjectives or other stereotyped dependencies, TestConductor also supports RequirementCoverage measurement based ModelCoverage and traceability dependencies from model elements to requirements (cf. Computing Requirement Coverage on page 102 ff).

TestConductor Dialog

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

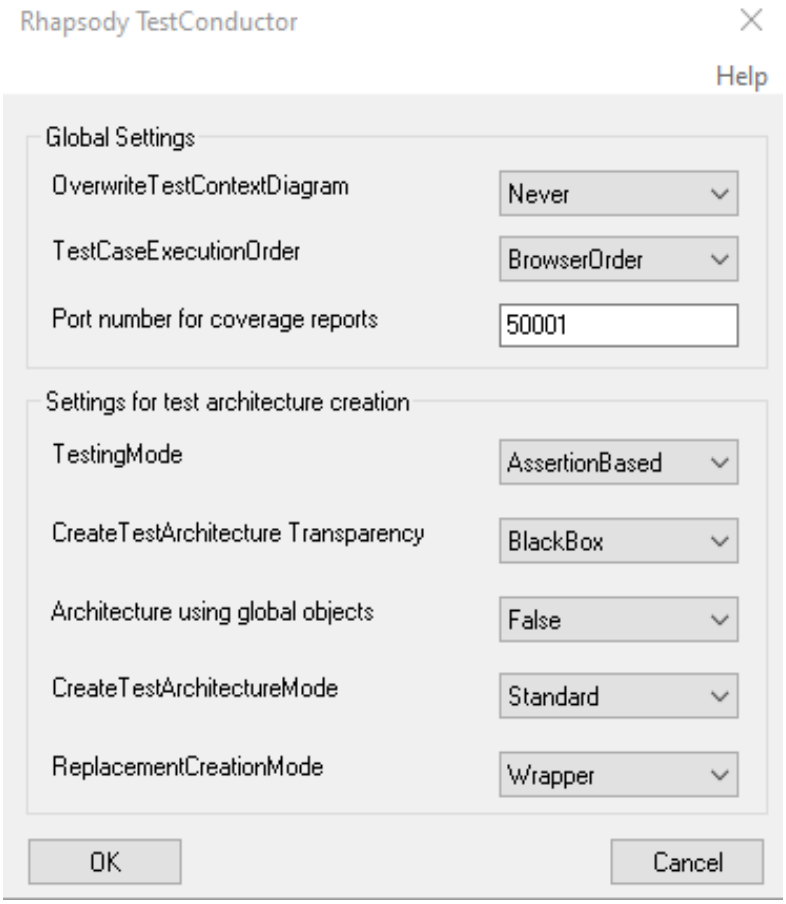


Figure 20: TestConductor Main Dialog

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 section **Settings – General Properties** on page **126ff.**

- Global Settings
  - Overwrite TestContextDiagram = {Always,Never,AskUser}
    Lets the user define how TestConductor shall proceed when updating the TestArchitecture. The TestContextDiagram may have user changes, such as colouring or arrangement. By default, a new TestContextDiagram will be created.
  - TestCaseExecutionOrder = {BrowserOrder,DeclarationOrder}
    By default the TestCases of a TestContext, TestSet or a TestPackage will be executed in the order in which they appear in the browser.
    Note, that the browser order can be user-defined.
    Alternatively, the order can be determined by the order in which the TestCases were defined.

- Port number for coverage reports = default:50001
  ModelCoverage reports allow for highlighting elements in the Rhapsody model for names listed in the report. This highlighting is based on java script in the report, connected via tcp/ip to the Rhapsody incarnation. When running multiple Rhapsody instances, the connection between report and the respective Rhapsody incarnation have to use different ports.
  On opening a report in a browser, thhe port connection is established for the port defined in this setting (note, that the used port can also be changed in the opened report).

- Settings for test architecture creation

  - TestingMode = {AssertionBased,AnimationBased}
    Lets the user chose the TestingMode for which a TestArchitecture will be created. This document only concerns the AssertionBased TestingMode. For information regarding the legacy AnimationBased TestingMode, we refer the user to the RTC_User_Guide.pdf document in the Doc/pdf_docs/legacy folder of the Rhapsody installation.

  - CreateTestArchitecture Transparency = {BlackBox,GreyBox}
    Determines with which transparency the TestArchitectures will be created (cf. sections Black Box Testing and Grey Box Testing on page 45 ff).
    Note, that this setting only affects the way, a new TestArchitecture will be created. It has no effect on existing TestArchitectures.
    Default is BlackBox.

  - Architecture using global objects - {False,True}
    The default TestArchitecture instantiates the SUT and TestComponentInstances as parts of the TestContext. This is suuitable for testing classes.
    When dealing with global objects (e.g. implicit classes or C-singletons), the objects can nor be instantiated in a TestContext, but have to be referred as global objects (cf. TestArchitecture chapter, section Using Objects on page 40).

  - CreateTestArchitectureMode = {Standard,Advanced}
    By default, TestConductor creates a TestArchitecture according to the rules described in section TestArchitectures on page 20 ff. A standard decision will be taken, if there is a choice, e.g. between inheriting TestComponent and using a replacement. Advanced TestArchitecture lets the user take such decision interactively. For TestComponents, it can individually be decided, whether to use wrapper or stub replacements or to prefer replacements over inheriting TestComponents when there is a choice. Furthermore, for the code generation configuration of the newly created TestArchitecture, an existing configuration can be selected from which all overridden properties, tags and settings will be taken over (cf. section Automatic TestArchitecture Generation on page 31 ff.).

  - ReplacementCreationMode = {Wrapper,Stub}
    For replacements created by TestArchitecture creation, it can be preselected whether these are created as copies of the replaced classes, or if abstraction will be applied on the copies (cf. section ReplacementsReplacements. on page 23 ff).

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.

The properties of Subject TestConductor are organized in Metaclasses

- SDInstance – the properties of metaclass SDInstance are used only in the animation based testing mode. These properties are irrelevant for assertion based testing
- Settings – properties of metaclass Settings are mainly relevant for TestArchitecture creation and can be defined on project-level in the TestConductor main dialog. These properties are explained in section Settings – General Properties on page 126ff.
- TestContext – properties of metaclass TestContext
  - affect SD TestCase creation for composite SUTs,
  - are aimed at setting a common pre- and post- TestCase-execution operation
  - allow for selecting the execution method for all TestCases of the TestContext.,

  The properties of metaclass TestContext are explained in section TestContext Properties on page 130.
- TestCase – most of the properties of metaclass TestCase are irrelevant for assertion based testing mode. Only very few configuration options affecting individual TestCases were considered useful.The majority of configuration options will affect all TestCases executed with one code generation configuration can be set on the respective configuration in the tags of stereotype `<<AssertionBasedTestingConfiguration>>` or `<<TestingConfiguration>>`, respectively (cf. section Tags of the <<AssertionBasedTestingConfiguration>> and <<TestingConfiguration>> Stereotypes on page 6969ff.).

  The relevant properties of metaclass TestCase are explained in section TestCase Properties on page 132ff.

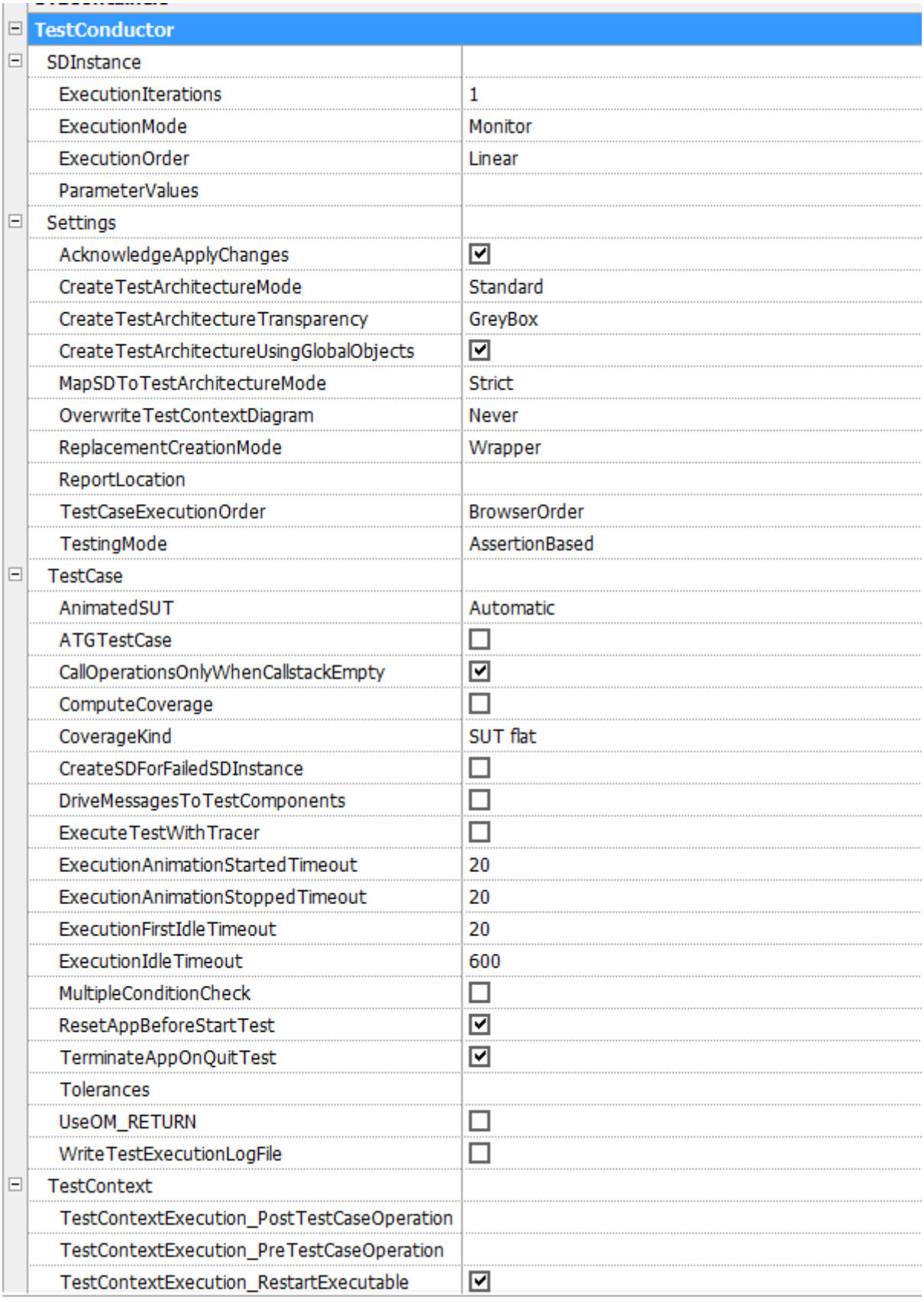


Figure 21: Properties - TestConductor

On the project, level, only properties of TestConductor::Settings are available. The properties of the other metaclasses are available from TestPackage-level on.

Settings – 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`

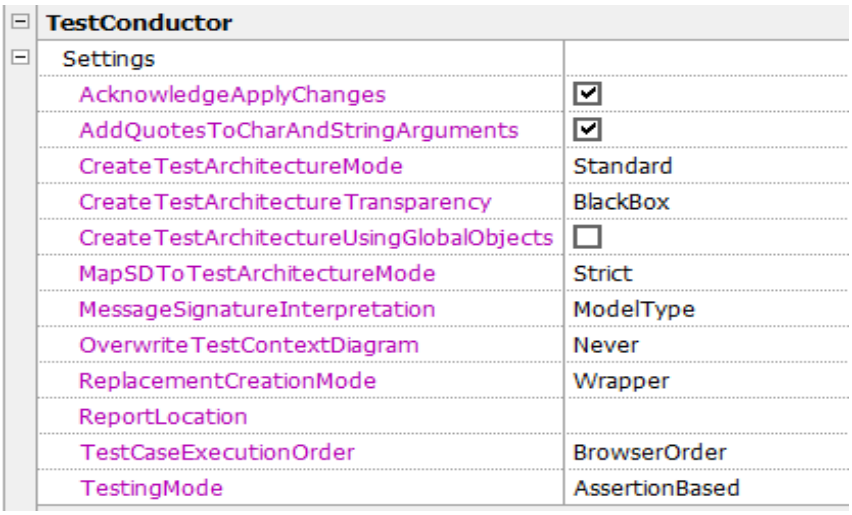


Figure 22: Properties TestConductor.Settings

`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::AddQuotesToCharAndStringArguments`

If this property is switched on (Default), TestConductor will interpret unquoted message argument values for char- and char*-arguments as if they were quoted. For e.g. char-argument x a value `0` will be interpreted as `'0'` and for char*-argument y a value `MyValue` will be interpreted as `"MyValue"`.
If property `TestConductor::Settings::AddQuotesToCharAndStringArguments` is unchecked on the TestPackage, a value `0` of char-argument x will be interpreted as numerical value 0. In order to specify character `'0'`, single quotes " must explicitly be used, For char*-argument y a value `MyValue` will be interpreted as identifier name `MyValue` instead of `"MyValue"`. In order to specify an explicit constant string value, double-quotes "" have to be used explicitly (see also Dealing with char- and char* - Arguments – Property TestConductor::Settings::AddQuotesToCharAndStringArguments . on page 148).
Unchecking the property on an individual TestCase or TestScenario will have no effect. The property is only regarded on TestPackage level.

`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.
The property only has an affect on creation of a new TestArchitecture and should be modified only on project level. In TestPackages, the property is set to the actual value of the project-level property at creation time of the TestPackage. The property should not be modified on an individual TestPackage.
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.
The property only has an affect on creation of a new TestArchitecture and should be modified only on project level. In TestPackages, the property is set to the actual value of the project-level property at creation time of the TestPackage. The property should not be modified on an individual TestPackage.

`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.
The property only has an affect on creation of a new TestArchitecture and should be modified only on project level. In TestPackages, the property is set to the actual value of

the project-level property at creation time of the TestPackage. The property should not be modified on an individual TestPackage.

`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::MessageSignatureInterpretation`

This property controls the interpretation of arguments and returns in messages with respect to operation signatures according to the realization of messages.
Possible values are “ModelType” and “CodeType”, where “ModelType” is the default. TestConductor distinguishes the model signature of operation from the code signature. Since “ModelType” treats operation arguments according to their model type and regards the direction to optimally support value specifications in the TestScenarios, “CodeType” treats operation arguments only by their resulting signature in the generated code. See Specifying Argument Values on page 146ff for more information.

`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”.

[48]`TestConductor::Settings::ReportLocation`

With this property[49] 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

[48]

[49]Property will be evaluated not only on project but also also on package level.

hyperlink will be created[50] 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:[51]

`$TESTPACKAGENAME`: Will be substituted by the name of the TestPackage[52] 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.

`$TESTSETNAME`: Will be substituted by the name of the executed TestSet. 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” form the context menu (cf. section Ordering of TestCases on page 91).

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”.

This setting does not affect any existing TestArchitecture. The property should not be modified.

[50]Hyperlink will be created only for test elements which can be written.
[51]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”)
[52]The outer TestPackage in assertion based mode

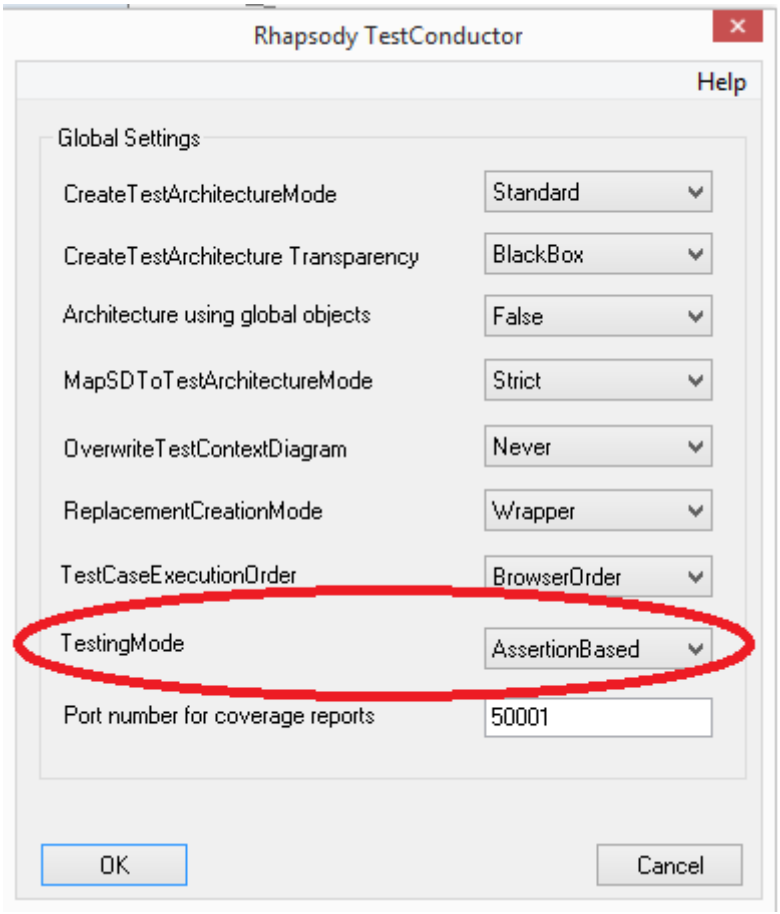


Figure 23: Setting TestingMode

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::CreateSDTestCaseWithParts`

For testing composite SUTs in GreyBox testing mode, one often want to specify the communication behavior among the parts of the composite SUT. TestConductor offers an automatic GreyBox TestArchitecture creation for one decomposition-level, i.e. representing also the parts of a composite GreyBox SUT as GreyBox elements (cf. section GreyBox TestArchitectures for classes, objects and Files on page 41 and section Grey Box Testing on page 45).

`TestConductor::TestContext::CreateSDTestCaseWithInstanceLabels`

The names of life-lines used in TestScenarios denote a navigation expression to the represented instance starting in the TestContext and reflecting the path by dot notation and allowing for indices in order to denote individual objects if instances with multiplicity are used. As a consequence, life-line names can grow very long and can become difficult to read for the user. Since the naming scheme is necessary to identify the life-line with individual parts or objects in the TestArchitecture, names can not simply be abbreviated. Labels can be used to show user defined labels instead instead of the interpreted names in the life-line heads. Unfortunately, applying label-mode to a TestScenario also shows labels instead of names for the messages. Label-mode prevents messages from being easily edible right in the diagram (to edit a message in label-mode, the features dialog on the message has to be used).
A manual solution to the problem is to set the label to be shown instead of the name for each individual life-line using the 'Display Options...' dialog on the respectibe life-line.
Property `TestConductor::TestContext::CreateSDTestCaseWithInstanceLabels` affects the creation TestScenarios by 'Create SD TestCase' and 'Create TestCase from Scenario' by defining a short-name label for each life-line in the created TestScenario and setting the life-line's display options appropriately. The names of the life-lines are still set according to the needs and can be inspected and modified in the features dialog, but instead of the lengthy name, the shorter label is displayed in the TestScenario editor.
The display options dialog can can be used to override the effect on the life-lines individually.

`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. By default, the property is checked.
Note, that executing all TestCases in one run of the application, will start each TestCase in a state of the application that was reached by its predecessor TestCase. Thus, order of TestCases is crucial and modifying one TestCase could affect all its successors. Do only uncheck this property when you exactly understand what you are doing.

`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 (cf. section Test scheduling with <<Scheduler>> TestComponents on page 36).

`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 (cf. section Test scheduling with <<Scheduler>> TestComponents on page 36).

TestCase Properties

Most properties in metaclass TestCase are irrelevant for the assertion based testing mode. Almost all configuration options that affect TestCase execution, can be defined using the tags of the stereotype on the active code generation configuration, using which the TestCases will be executes (cf. section Tags of the <<AssertionBasedTestingConfiguration>> and <<TestingConfiguration>> Stereotypes on page 69ff).

We here only list the properties of metaclass TestCase that are relevant for assertion based testing mode.

`TestConductor::TestCase::ExecuteWithTracer`

For TestCases that have been build with animation instrumentation, i.e. the active code generation for the execution has animation instrumentation switched on, the execution is performed with the tracing feature of Rhapsody's animation enabled.

`TestConductor::TestCase::ExecutionIdleTimeout`

Using this property a TestCase specific individual maximal duration for the execution can be specified. The default f the property is 600 seconds. If e.g. 20 seconds are specified in the property, the TestCase will be aborted if the execution of the TestCase does not terminate within 20 seconds after starting the TestCase.

Generating Test Reports with IBM Engineering Lifecycle Optimization - Publishing

IBM Engineering Lifecycle Optimization - Publishing 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 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 > Publish > Report using user-specific template…**

  Note: For the correct interpretation of new term elements in the Testing Profile, IBM Engineering Lifecycle Optimization - Publishing needs to be started with a trailing "NewTermsOn" in the URL of the Rhapsody REST API. Because of this, when using other templates before or after TestRequirementCoverage.dta, the Rhapsody client must be restarted. Otherwise Rhapsody will start IBM Engineering Lifecycle Optimization - Publishing with the previously used URL.

- Select the 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 IBM Engineering Lifecycle Optimization - Publishing 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 IBM Engineering Lifecycle Optimization - Publishing Document Studio is described in the 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\RPE\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 TestPackage execution has finished. This is the preferred way of executing TestPackages via the TestConductor API.

Applicable to TestSet elements

- “Update TestSet”
- “Clean TestSet”
- “Build TestSet”
- “Execute TestSet”

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

- “Execute TestSet Sync”

  Performs synchronous TestSet execution, i.e., the function returns only after the execution of the TestSet has finished. This ensures that subsequent TestConductor API commands are only performed after the TestSet execution has finished. This is the preferred way of executing TestSets 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", etc. 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 (only for Rhapsody in C) or
  - the environment/system border, i.e. objects not explicitly represented in the TestScenario.

  An object represented by a life-line is determined by its name and its 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. Such navigation names can become relatively long and difficult to read. In order to display shorter names in the life-line heads, a label can be set in the features dialog of a life-line and the label can be displayed instead of the name (and classifier)[53].

- asynchronous Message – **Event**:

  - Stereotype `<<RTC_MsgInfo>>` on Messages is used by TestConductor for definition of additional informations regarding the interpretation of messages.

[53]TestConductor supports creation of TestScenarios with automatic generation of labels for life-lines and appropriate setting of the display options, see property TestConductor::TestContext::CreateSDTestCaseWithInstanceLabels on page 130.

- synchronous Message
  - **Operation**:
  - **Triggered Operation**:
  - Stereotype <<RTC_MsgInfo>> on Messages is used by TestConductor for definition of additional informations regarding the interpretation of messages.
- **dataflow Message**:
  - Stereotype <<RTC_MsgInfo>> on Messages is used by TestConductor for definition of additional informations regarding the interpretation of messages.
- **Condition**:
  - Conditions are used for various purposes in sequence diagrams. In particular, Rhapsody uses conditions to display active states of behavioral diagrams in animated sequence diagrams. But conditions can also be used by developers or testers to specify states, situations or execution conditions in a more or less free textual form. Because of the wide variety of such use cases for conditions, TestConductor does not interpret conditions but uses TestAssignments and TestConditions instead.
- **TestAction**:
  - general TestAction: A general TestAction allows entering arbitrary source code to be executed, when the TestAction is activated during TestCase execution. See section (general) TestActions, TestAssignments and TestConditions page 160.
  - Message-related TestAction: Message related TestActions can be used to override automatically generated test artifacts, as for example the declarations used in a DriverOperation, the call used in a DriverOperation, etc. See section Influencing DriverOperation and Stub generation using TestActions in TestScenarios on page 157.
    - <InitAction>
    - <PreCallAction>
    - <CallAction>
    - <PostCallAction>
    - <StubAction>
- **TestCondition**: can be used to specify a condition that will be evaluated in an assertion when the TestCondition is activated during TestCase execution. See pages Error: Reference source not found and 160.
- **TestAssignment**: can be used to assign a particular value to a (member-) variable when the TestAssignemt is activated during TestCase execution. See page 160.
- **Time-Interval** . Can be used to specify a maximum time-interval for a series of messages or specify a minimum separation time-interval between successive messages (Time-Intervals are explained in section Using Time Intervals on page 144).

- **InteractionOccurrence**: allows to reference an entire TestScenario from within another TestSceanrio. Can be used e.g. to share common sub-scenarios in multiple TestScenarios, such as system start-up sequences. InteractionOccurrences can also be used just to keep individual diagrams smaller and to organize the TestScenario specifications. See page 150.
- **InteractionOperator**: (see section Using Interaction Operators in SD TestCases on page 162). InteractionOperators are used to specify conditional sub-scenarios, repetitions, alternative or parallel (i.e. unordered) subsequences or to focus on particular occurrences of some messages.
  - `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>>` can be applied on InteractionOperators to specify additional informations for the interpretation of the oeprator (see section Using Interaction Operators in SD TestCases on page 162).

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
- Conditions

Note: TestConductor will ignore condition marks during test execution (see exception above).

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, Rhapsody in C++, Rhapsody in Java or Rhapsody for SysML.)

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.

For specification of arguments and return values of a message, see section Specifying Argument Values on page 146ff.

Ignoring Unrealized Messages

If a message shall be left unrealized intentionally, it is recommended to set a dedicated stereotype on the message to indicate, that leaving the message unrealized was not just a mistake. Messages with stereotype `<<Unrealized>>` are filtered out and ignored in update and 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.

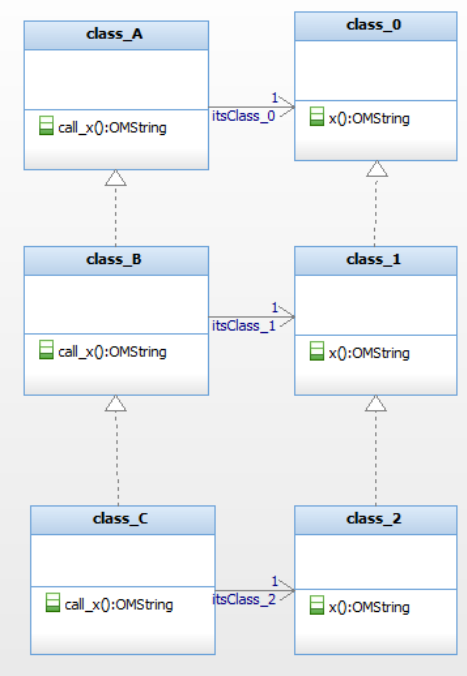


Figure 24: Example relations with inheritance

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):

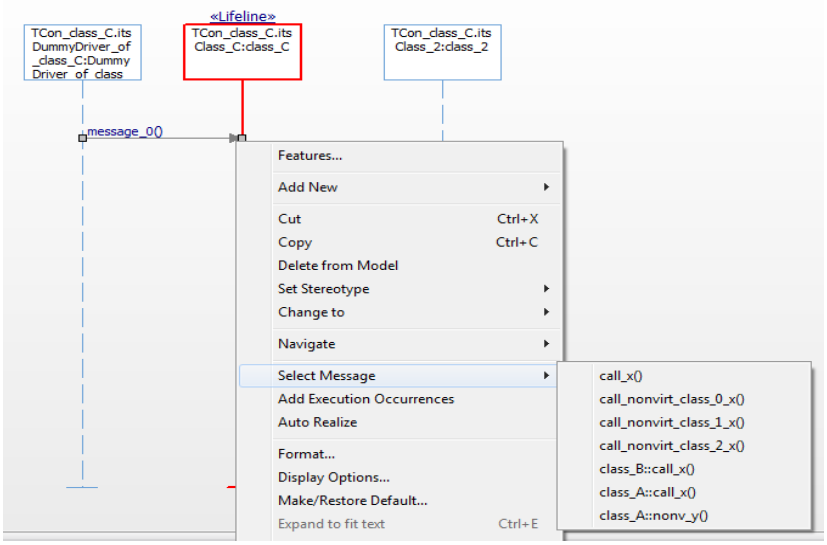


Figure 25: Select Message with virtually inherited operations

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

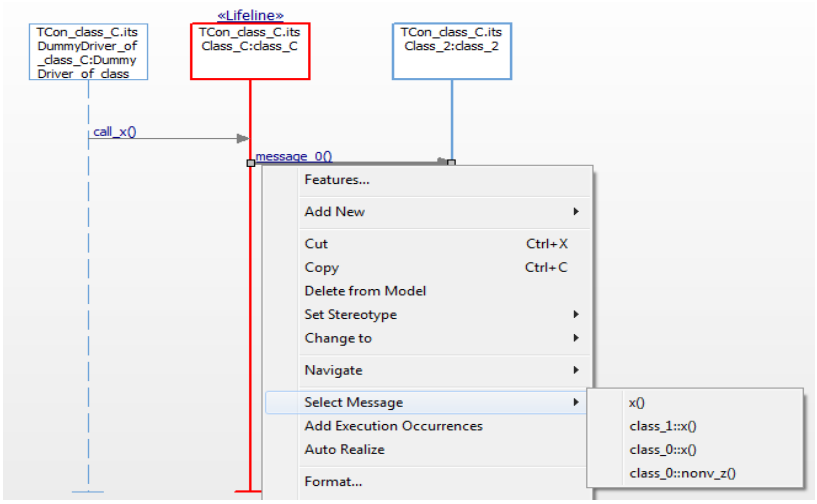


Figure 26: Select Message and Inheritance

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

- if a DriverOperation will be generated from the message – as it is the case in figure 25, 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 26, 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.

Note, that Java does not support non-virtual call as in C++. Hence, stereotypes `<<call_virtual>>` and `<<call_nonvirtual>>` do not have an effect for Rhaposody in Java.

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 45). Therefore self-messages are ignored by TestConductor and appear blue in witness TestScenarios.

In GreyBox testing (cf. section Grey Box Testing on page 45 and GreyBox TestArchitectures for classes, objects and Files on page 41) 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 23). 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.

Stereotype SelfMessageRealizationInParts

(cf. Samples:

- TestCase 'SD_tc_gb_observation_record' in TPkg_Coffeemachine_GB::TCon_Coffeemachine_Architecture::TCon_Coffeemachine
  - of Sample Samples/CppSamples/TestConductor/TestingCookbook/CppCompositeCoffeeMachine_RAL
  - or in Samples/CppSamples/TestConductor/TestingCookbook/CppCompositeCoffeeMachine_wo_ports

)

TestConductor automatically introduces replacements for one decomposition level when creating a GreyBox TestArchitecture (cf. section GreyBox TestArchitectures for classes, objects and Files on page 41 ff).
This enables support for TestScenario specifications with individual 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 a 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 SUT life line as representing a composite class or object and its parts also in TestScenarios: for (self-)messages to such a life-line, the realization has to be either a member operation or reception of the composite class or object itself or a member operation or reception of one of its parts. By default, the Sequence Diagram Editor offers only member operations and receptions of the class or

object represented by the life line[54]. To enable also selection of member operations and receptions of parts of the composite SUT as message realizations, stereotype `<<SelfMessageRealizationInParts>>` has to be set on the respective TestScenario.
Member operations and recptions of part classes will be offered for selection as realization in the 'Select Message' context menu on messages and in the features dialog of messages after setting stereotype `<<SelfMessageRealizationInParts>>` on the respective TestScenario.
TestConductor will add observation instrumentation to the part class replacements[55] according to the advanced realization information enabled by stereotype `<<SelfMessageRealizationInParts>>`.

Even though the stereotype name emphasizes self-messages, the stereotype is also needed sometimes for specifying the real receiver of a message when GreyBox testing a composite SUT with delegation ports: When sending an event to a GreyBox SUT, which delegates the event via a non-behavioral port to one of its parts (delegation port), TestConductor can not infer where to add the observation instrumentation, if the composite SUT itself also has a reception declaration for this event. It then depends on the formal message realization whether the instrumentation is added the SUT or to its part. By default the realization of an event message is the event itself. Setting stereotype `<<SelfMessageRealizationInParts>>` on the TestScenario allows for specifying the reception of the part's class as realization of the message. TestConductor then will instrument the event consumption of the part's class for observation of this event message instead of instrumenting the event consumption of the composite SUT[56].

Note that realization of messages can only be chosen from class or interface members but can not be set to members of individual instances using the features dialog or the "Select Message' context menu item. To enable specification of an individual receiver part for a message, tag `GBSUT_ObserverInstance` of stereotype `<<RTC_MsgInfo>>` can be used in combination with `<<SelfMessageRealizationInPart>>`. While the realization of the message still determines the particular member operation or reception of the part's class that realizes the message, tag `GBSUT_ObserverInstance` determines the individual receiver instance as navigation expression.

Using Time Intervals

(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

[54] as well as inherited operations and receptions, of course.
[55] cf. section Replacements on page 23 ff and GreyBox TestArchitectures for classes, objects and Files on page 41 ff.
[56] The same ambiguosity may arise if an operation is invoked via delegation port, but the composite also has an (obsolete) operation with matching name and signature. Then this ambiguosity can be solved using stereotype `<<SelfMessageRealizationInParts>>` accordingly.

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.

Time delays will be specified with the time interval notation in sequence diagrams. 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.

Examples:

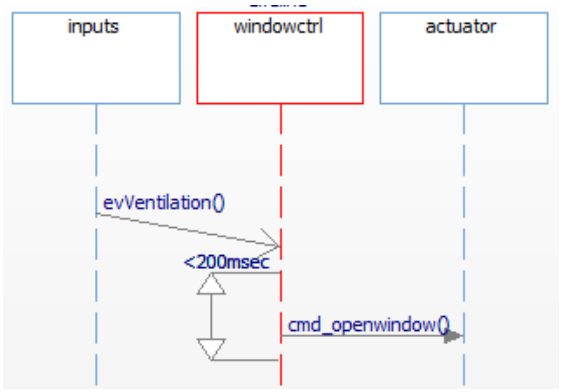


Figure 27: Example of an upper bound time interval for a reaction (bounded liveness)

In figure 27, an upper time limit for the reaction of the SUT is specified. The message '`cmd_openwindow()`' shall be sent to the actuator within 200msec after receiving the '`evVentilation()`' message from inputs. If the time-interval ranged over more than one message, then it would specify, that all the messages in the range of the time-interval had to be observed before the time bound expired.

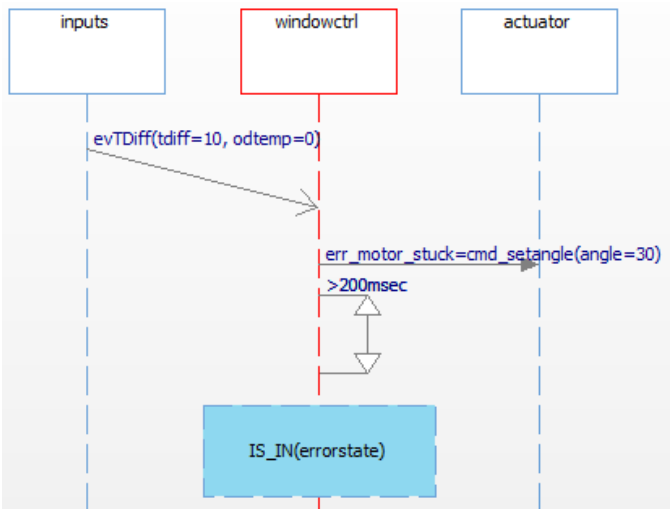


Figure 28: Example for a minimal time distance (timed separation)

Figure 28 shows an example for a minimal time distance specification. After receiviving message '`evTDiff(diff=10,odtemp=0)`' from inputs, windowctrl shall send message '`cmd_setangle(angle=30)`' to actuator. For the specification of a return for the message to a TestComponentInstance, TestConductor will generate an appropriate stub, s.t. '`err_motor_stuck`' will be returned. This return leads to a state change in windowctrl. The example specification shall check, whether the state change takes place as desired. But the TestCondition testing the active state shall not be activated immediately after the call of 'cmd_setangle()' in the actuator. The TestCondition shall be evaluated at earliest more than 200msec after the call of 'cmd_setangle()' in the actuator.

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[57], TestConductor will display the actual argument value in the witness TestScenario (cf. page 169) 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.

[57]TestingConfiguration tag `rtc_assert_handling` has to be set to `by_string`

TestConductor will issue a warning on updating TestCase/TestSet/TestContext/TestPackage for messages with unspecified argument values.

For individual argument values, also don't care '*' can be used (cf. page 151). 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 151). A dedicated syntax supports specification of input and output value for `InOut` arguments separately (cf. page 149).

ModelType vs. CodeType – MessageSignatures in TestScenarios

By default, argument specifications in TestScenarios adhere to the model type signatures of the operations and receptions as they are displayed in the Rhapsody browser. Model signatures often differ from the signatures in the generated code. Code signatures in general depend on various properties. E.g. an operation `f()` with argument `x` of type `ty` and direction `InOut` appears in the browser as `f(ty x)`, whereas the code signature of the generated function `f()` will look like for example

`void f(ty &x)`, or `void f(ty *x)` in C or C++, mainly dependent on property {LANG}`_CG.Type.InOut` of type `ty`.

For argument value specifications adhering to the model type message signature, TestConductor will generate appropriate variable declarations and initializations to provide a code type consistent driver or stub for the message with values according to the specified arguments.

These transformations conveniently support test specifications from a model perspective. From a rather code oriented perspective, the model type oriented view and the involved transformations are sometimes counter intuitive. They do not permit specification of arguments according to the code signatures directly, In particular, pointer and reference arguments can often not be directly provided in the TestScenario specifications without using dedicated TestAcions (cf. section Influencing DriverOperation and Stub generation using TestActions in TestScenarios on page 157).

Users have different perspectives on models. In many use cases, the model itself and hence the model type view is in the focus of interest, code generation is viewed as prerequisite of execution or simulation of the model. In other use cases, the model is seen as artefact on the way to obtaining generated code. Here the generated code is the main focus of interest and thus test specifications refer to code signatures rather than referring to model sinatures.

In order to support also TestScenario specifications referring to code signatures, TestConductor offers property `TestConductor::Settings::MessageSignatureInterpretation = { ModelType, CodeType }` on TestPackages. The property has to be set on the outer TestPackage `TPkg_<SutName>` and is by default set to `ModelType`. The property affects the entire TestArchitecture in `TPkg_<SutName>` including all TestCases.

For the impact of this property for argument specification see section Specification of Out and InOut Argument Values on page 149.

Note, that different treatment of arguments according to ModelType or CodeType message signature interpretation mainly affects Rhapsody in C and C++. For Rhapsody in Java ModelType and CodeType interpretation will make a difference only in very rare cases, due to the capabilities of Java itself. Java passes all non-primitive data types by reference,

but passing is always be value, i.e. for all non-primitive types reference is passed by its value (copy). All primitive types generally passed by value. Hence, only for a very few cases, using InOut or Out arguments may make sense at all

Dealing with char- and char* - Arguments – Property `TestConductor::Settings::AddQuotesToCharAndStringArguments`

Rhapsody omits quoting of char- and char*-arguments in animation and animated sequence diagrams. In order to support generating TestCases from animated scenarios, and for historical reasons, TestConductor also doesn't require char- and char*- message arguments to be quoted.
This leads to ambiguosities how to interpret e.g. char argument "`x=0`". TestConductor by default treats this argument as '0' (= 0x30) instead of numeric value 0.
The same also applies to char* arguments. TestConductor interprets by default argument specification e.g. "`x=&globalstring`" as character array"`&globalstring`" instead of `<address of global variable globalstring>`.

For influencing the interpretation (in particular for *avoiding interpretation*) of char- and char*-message arguments and for taking them '*as specified*', property `TestConductor::Settings::AddQuotesToCharAndStringArguments` can be unchecked (Default: checked) on the TestPackage. This property setting has no effect when applied to individual TestCases or TestScenarios but is only regarded on the entire TestPackage.
If the property is unchecked, the user is responsible for quoting character values with single quotes, e.g. '0' for specifying the character 0 (=0x30). Unquoted character will then be interpreted as identifiers or as numbers.
Char*-arguments have then to be specified explicitly as e.g. "`MyValue`" if the argument value string "`MyValue`" shall be used. Unquoted char*-argument `MyValue` will then be interpreted as identifier name `MyValue`.

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 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/TestSet/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.

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 (e.g. to force declaration and usage of an ‘osc_ret’ variable to keep the returned value for later use in DriverOperations, cf. page 151). 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 151).

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[58] {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.

[58]where {Lang_CG} is either C_CG or CPP_CG, depending on the language setting of the model.

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[59] 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

```
<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[60]

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)

Note: if property TestConductor::Settings::MessageSignatureInterpretation is set to CodeType on the outer TestPackage TPkg_<SutName> , directions of arguments will not be treated in the way described above, but only the resulting code type of the argument is considered (cf. section ModelType vs. CodeType – MessageSignatures in TestScenarios on page 147). In particular, no dedicated assertions and stubs for InOut and Out arguments will be generated. If needed, such assertions and stubs can be specified using <StubAction> Testctions (cf. section Influencing DriverOperation and Stub generation using TestActions in TestScenarios on page 157)

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,

[59]depending on whether the message is drawn from a TestComponent to SUT or the other way round.
[60]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.

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 170.

Don't care values

In some cases one might not be interested in checking actual parameter values. If

- 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.

Note, that in DriverOperations no ‘osc_ret’ varaible will be declared and the return of the driven call will not be stored in a variable, if no return value is specified in the message. In order to force the declaration of an adequate ‘osc_ret’ variable and keeping the returned value in this variable for later use (e.g. in a <PostCallAction>), ‘*’ specification for the return value can be used.

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 thy appear in messages for which DriverOperations or Stubs will be generated:

- for a driven message[61], 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[62], i.e. e.g. `msg3(x=[0..4])` TestConductor will check whether argument `x` of the message is in the specified range.
- 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 55.

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:

[61]Message from a TestComponent to SUT with in-argument x.
[62]Message from SUT to a TestComponent with in-argument x.

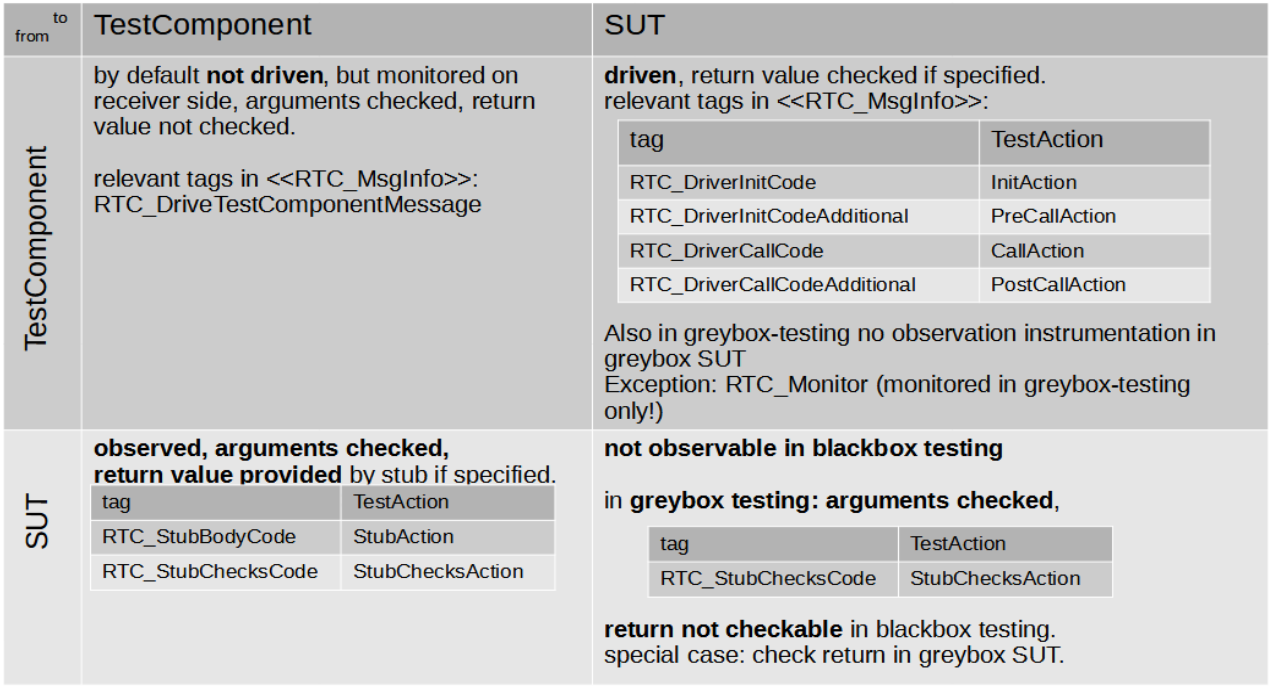


Figure 29: Interpretation of Messages by TestConductor

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 55).

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 refering to the message in the TestScenario.
- DriverOperation generation can also be influenced using the tag `RTC_ImediateEvaluation` of the `<<RTC_MsgInfo>>` stereotype on messages. By default, TestConductor will wait for a minimal delay before driving a message (realized by tm(1) in the arbiter generated from a TestScenario specification), i.e. in particular TestConductor separates two successive DriverOperations by this minimal delay. By checking tag `RTC_ImediateEvaluation` of the `<<RTC_MsgInfo>>` stereotype on a message to be driven, TestConductor can be forced to drive the message without adhering to this minimal delay.

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 tag `RTC_StubBodyCode` of the `<<RTC_MsgInfo>>` stereotype on messages
- using <StubAction > TestActions referring to the message in the TestScenario.

The default implementation of argument checks generated by TestConductor for messages from SUT to TestComponent or GreyBox SUT replacement may be overwritten and customized by the user. Overriding the default arguments check can be achieved by either

- using tag `RTC_StubCheckCode` of the `<<RTC_MsgInfo>>` stereotype on messages
- using <StubChecksAction> TestActions referring to the message in the TestScenario.

Influencing DriverOperation and Stub generation using `<<RTC_MsgInfo>>` tags

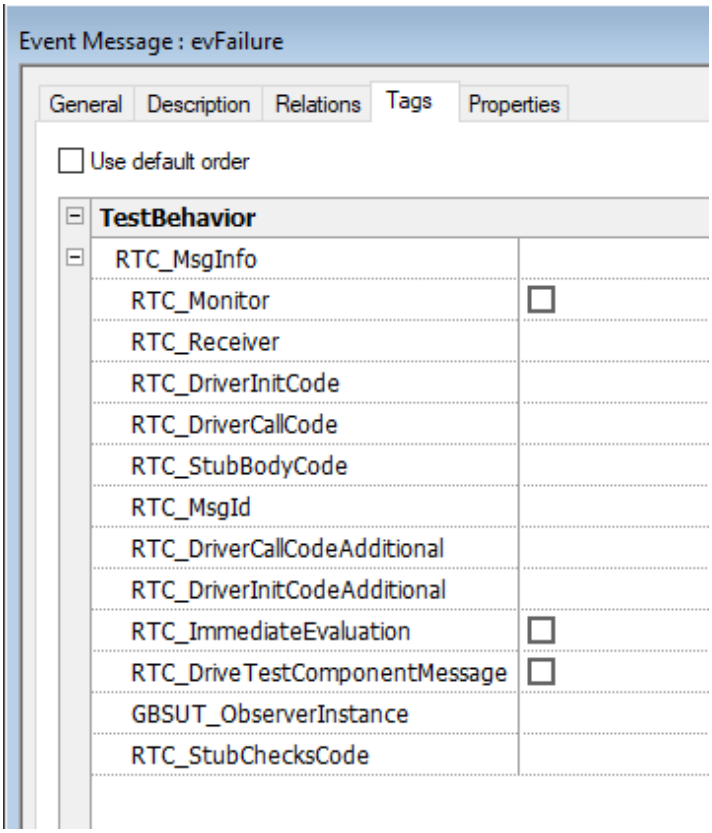


Figure 30: Tags of Stereotype `<<RTC_MsgInfo>>`

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`,
`TestBehavior::RTC_MsgInfo::RTC_StubChecksCode`

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.

Tag `GBSUT_ObserverInstance` serves for realising a message in a particular part instance for GreyBox Testing (cf. section Stereotype SelfMessageRealizationInParts on page 143).

Tag `RTC_ImmediateEvaluation` affects the TestConductor behavior for driving messages (see section User Defined DriverOperations on page 153 for details).

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 and RTC_StubChecksCode

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 157). 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.

For messages from SUT to TestComponents or GreyBox SUT replacements, arguments of the actual call have to be compared to the values specified in the message in the TestScenario specification. TestConductor generates assertion code for this argument check into the respective `<<StubOperation>>` operation for the related operation, or into the processEvent-operation of the TestComponent or GreyBox SUT replacement for the related event, respectively. Tag `RTC_StubChecksCode` (or using the corresponding <StubChecksAction> TestAction, cf. page 157) can be used to override the auto generated argument check code with a user defined code block.

Deleting <<RTC_MsgInfo>> Tags (User Defined Driver and Stubs)

TestConductor regards the tags RTC_DriverInitCode , RTC_DriverCallCode , RTC_DriverCallCodeAdditional , RTC_DriverInitCodeAdditional , RTC_StubChecksAction 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 one 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 155).

<InitAction> TestAction must be placed on sending TestComponent life-line *directly*[63] 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 155).
  <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 156).
  <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 156).
  <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. returning specific values. The code contained in the stub action is handled as the tag "RTC_StubBodyCode" of stereotype <<RTC_MsgInfo>> (cf. section "RTC_StubBodyCode and RTC_StubChecksCode" on page 156).
  <StubAction> TestAction must be placed on receiving TestComponent life-line *directly* after message reception.

- <StubChecksAction>: For messages from SUT to TestComponent or GreyBox SUT replacements, TestConductor generates a code block checking the actual arguments of the message against the specified values into the respective stub- or observation operation. This auto generated argument check can be overridden using <StubChecksAction> TestAction (or, alternatively, using tag "RTC_StubChecksCode" of stereotype <<RTC_MsgInfo>> (cf. section "RTC_StubBodyCode and RTC_StubChecksCode" on page 156).
  <StubChecksAction> TestAction must be placed on receiving TestComponent (or GreyBox SUT replacement) life-line *directly* after the respective message reception.

In order to add a TestAction to a sequence diagram TestCase:

[63]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.

- 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>` , `<StubChecksAction>` 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.
  Note, that TestConductor offers a separate small editor for editing TestActions. The editor can be invoked on a TestAction, TestCondition or TestAssignment from the context menu "`TestConductor->TestAction Editor`"

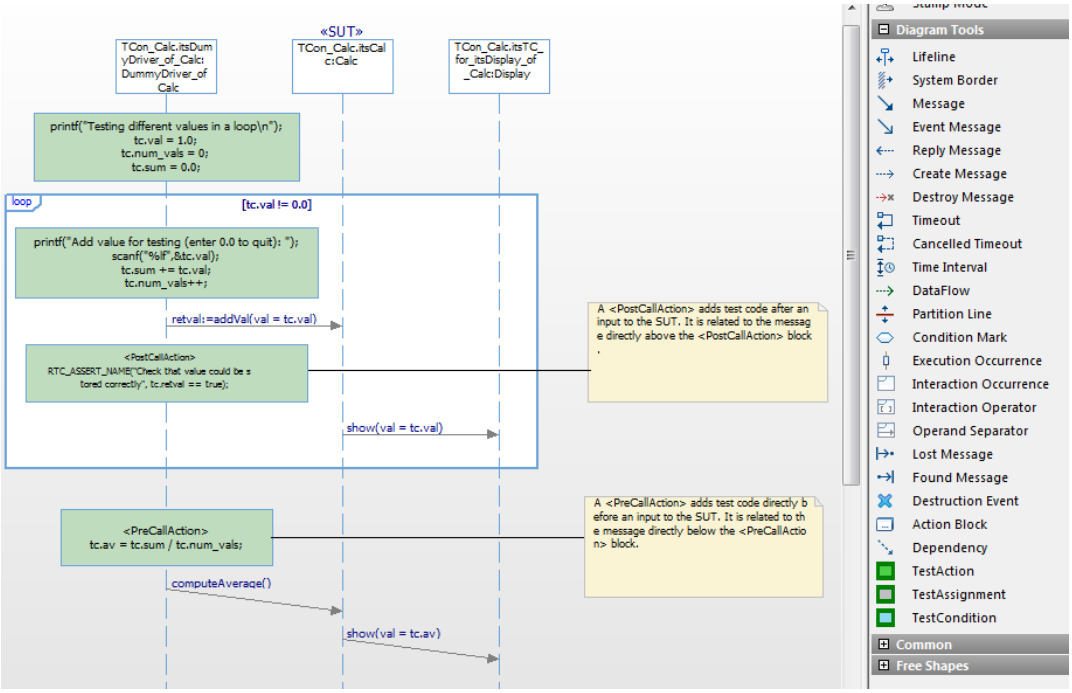


Figure 31: Using TestActions

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>:

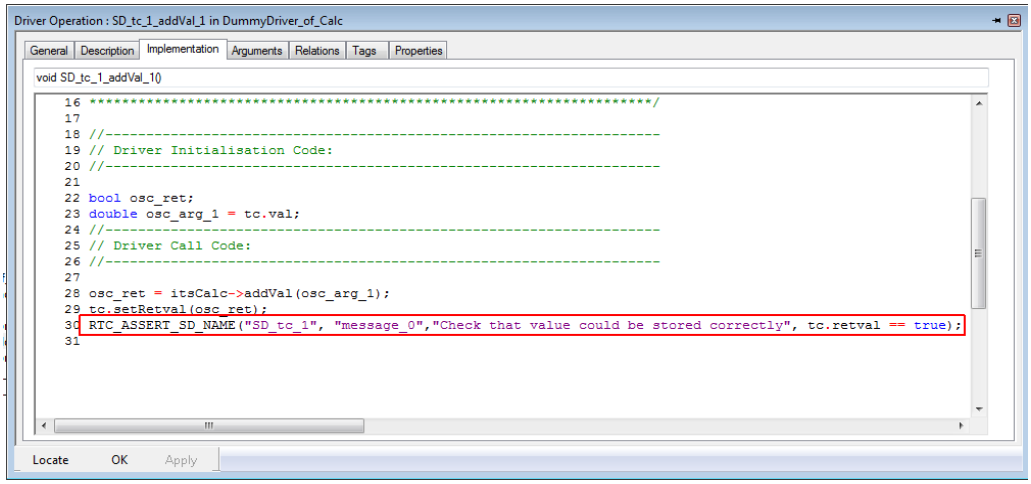


Figure 32: Assertion generated from <PostCallAction> TestAction

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 in 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>`, `<StubChecksAction>` 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 59).

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 59)**.**

(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.

)

Messages and InteractionOperators are powerful concepts to specify communication among life-lines of a TestScenario, but often there is a need to perform actions or initiate test specific helper functionality that can not easily be captured by messages only.
For this purpose, TestConductor offers general TestActions as a tool to execute arbitrary user defined code at a partticular instance of time during a test execution. TestActions are treated as regular parts of the specified sequence. A driver operation will be generated to the TestComponent or TestContext representing the life-line on which the TestAction is located. These driver operation will contain the unmodified contents of the TestAction and will be invoked within the sequence of occurences defined by the TestScenario.

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 assertions (cf. pg 200) - 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, etc.

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
      | !check_op | partidentifier.check_op
      | !partidentifier.check_op
check_op ::= IS_IN(stateidentifier)
name ::= identifier | partidentifier.identifier
relop ::= == | > | >= | <= | < | !=
```

value will not be interpreted by parser

By default, TestConductor waits a minimal delay, before a TestAction or TestAssignement is executed or, repectively, a TestCondition is evaluated. To avoid this minimal delay, e.g. for using a TestAction for incrementing a loop InterationOperator's counter or to evaluate a TestCondition between two succeeding reactions of a SUT, stereotype `<<SynchronousTestAction>>` can be applied on the respective TestAction, TestAssignement or TestCondition. In the separate small TestAction Editor, a drop down box 'Activation' offers 'delayed' and 'immediate'. This drop down box corresponds to adding and removing stereotype `<<SynchronousTestAction>>`, respectively.

The stereotype has no effect on <InitAction>, <PreCallAction>, <CallAction>, <PostCallAction>, <StubChecksAction> and <StubAction> TestActions.

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.
  The alternatives are interpreted as in a nested
  if(condition1) {}
  else if (condition2) {} ...

Instead of a specific condition, ‘else’ can be used for the last operand of an ‘alt’ operator.
Note, that ‘else’ condition is only understood correctly by TestConductor, if the condition contains only the word ‘else’ with no white spaces.

- `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 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.

For Interaction Operators, their kind, conditions of the operands and their activation, TestConductor offers a separate “InteractionOperator Settings” editor, which can be invoked from the context menu “TestConductor->InteractionOperator Settings” on Interaction Operators.
In this editor, drop down “Activation” with its possible choices “delayed” and “immediate” corresponds to setting and unsetting tag `RTC_ImmediateEvaluation` (of stereotype `<<RTC_OperatorInfo>>`), respectively.

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`”.

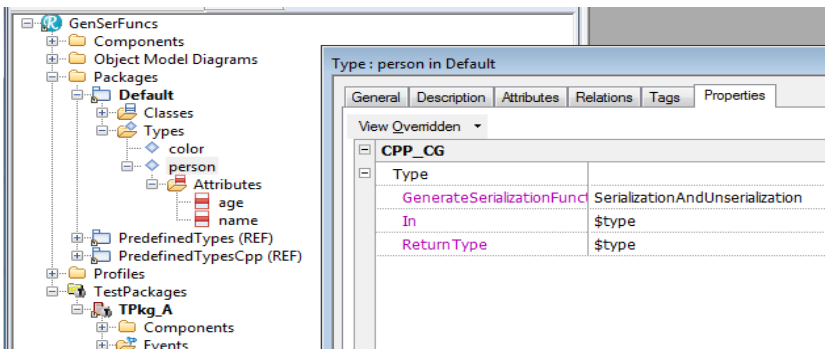


Figure 33: {Lang}_CG.Type.SerializationAndUnserialization

If this property is set to `SerializationAndUnserizalization` for a user defined type, automatically generated serialization and unserialization functions can be used for specifying argument values for message arguments of that type:

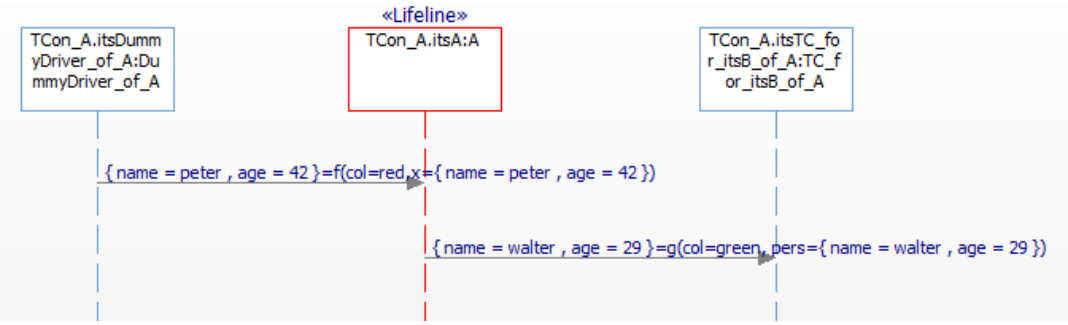


Figure 34: Using Serialization

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.

Testing untriggered initial behavioral

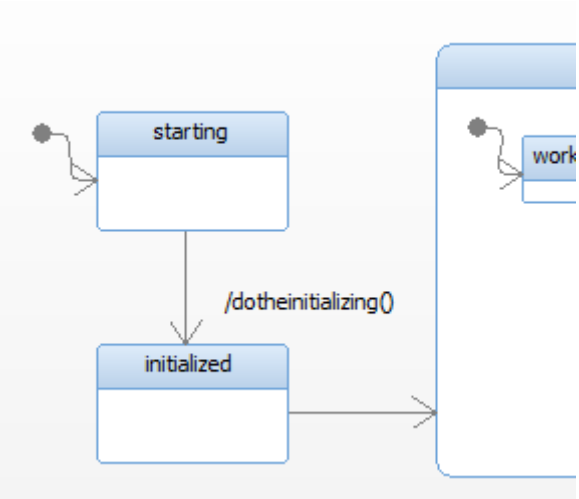


Figure 35: Statechart with untriggered initial behavior

Figure 35 shows a statechart with untriggered initial behavior. A SD TestCase for this statechart can not reliably refer to ‘dotheinitializing()’, since the TestCase and the statechart of the SUT will be started concurrently and it is not determined whether the TestCase is already started when the SUT will execute ‘dotheinitializing()’.
A simple solution for this problem is to add an additional event to the statechart of the SUT, which is normally self-triggered by the statechart depending on some compiler define. If this define is provided by the code generation configuration, then the statechart will wait on this event. Without the define, the statechart will behave as before except for the additional event appearing automatically in the execution.

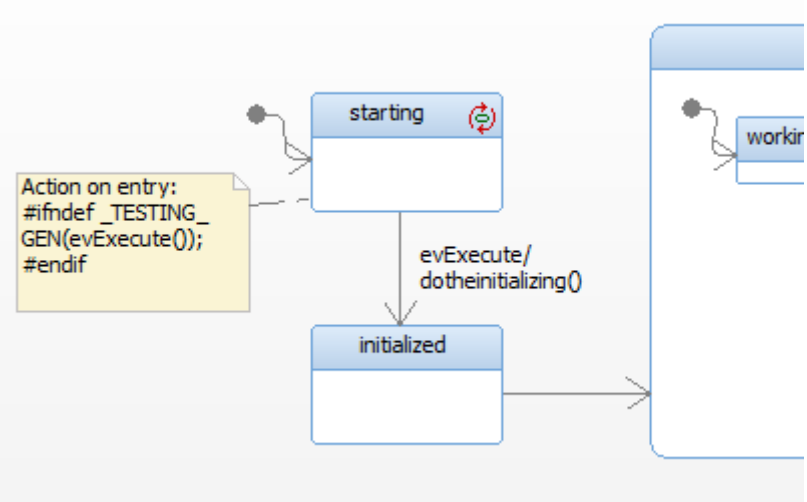


Figure 36: Simple solution for testability of untrigged initial behavior

If /D”_TESTING_” is added to compiler switches[64] of the code generation configuration of the TestArchitecture, TestCases now can explicitly start the statechart and hence ensure that also the initial behavior can be tested[65]:

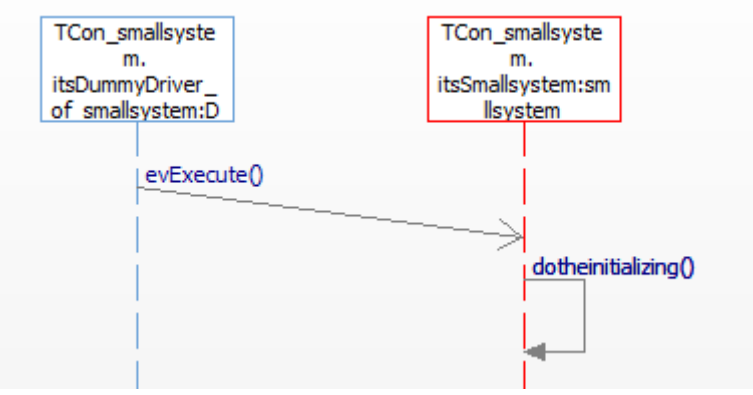


Figure 37: GreyBox SD TestCase testing initial behavior

[64]Example for MSVC
[65]Example for GreyB

For Java, a similar approach could be applied. Due to the lack of a preprocessor and of compiler defines, the approach could for example involve a static class with a static attribute which has to be set differently for building the test application and for building the design application.
Of course, the intervention into the model will have a stronger impact on the model than the presented solution for C and C++.

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[66].

Note, that existing witnesses will not be automatically updated for a new run of a TestCase. If after some changes in the TestScenario, the TestCase still failed, a new witness has to be generated by 'Show as SD' in order to obtain a new witness.

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.

[66]provided that TestingConfiguration tag `rtc_assert_handling` is set to `by_string`.

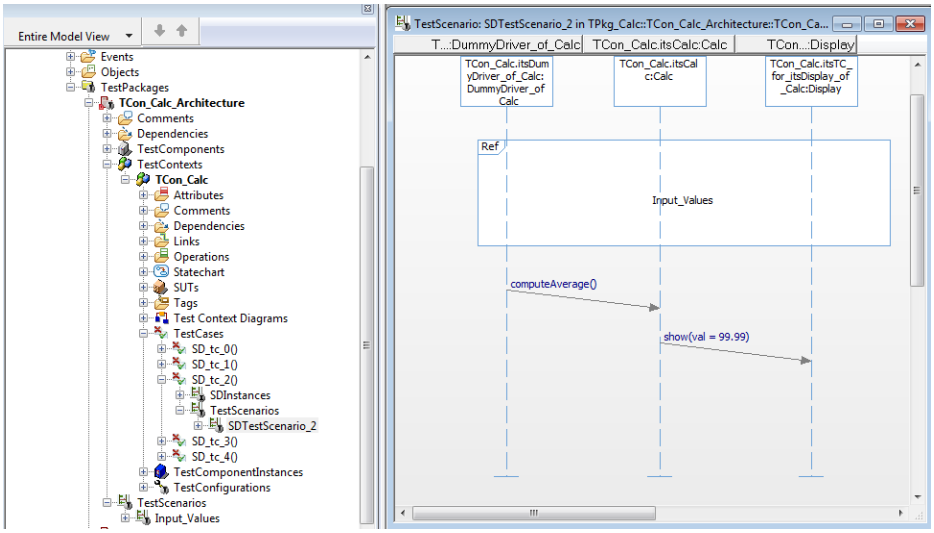


Figure 38: TestScenario with InteractionOccurrence

The TestCase shown in figure 38 can be found in sample model Samples/CppSamples/TestConductor/CppModelCodeCoverage.

Execution of the TestCase failed, 'Show As SD' is invoked:

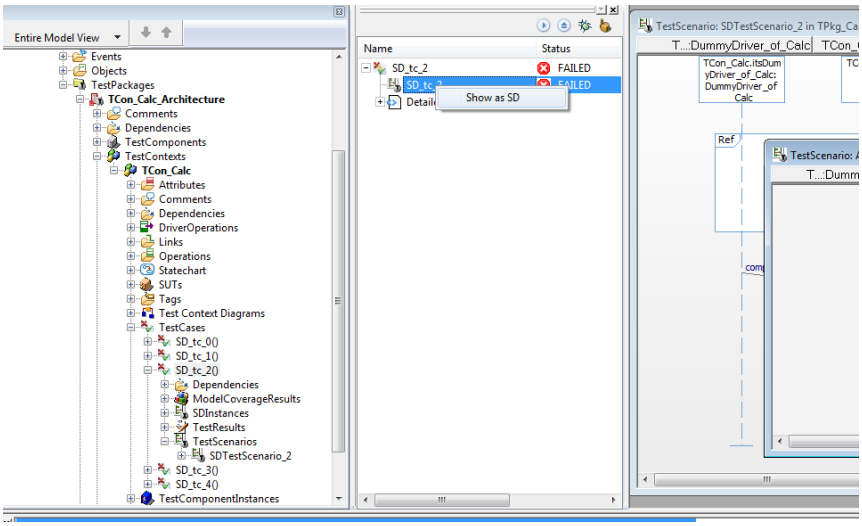


Figure 39: Invocation of 'Show As SD'

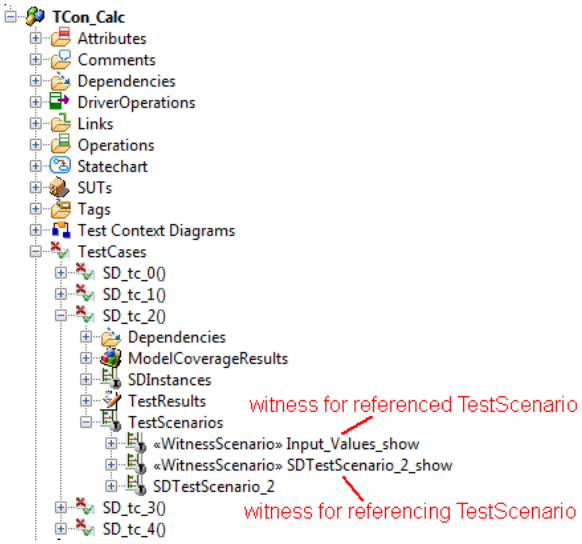


Figure 40: Witnesses for referencing and refererenced TestScenario

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 87.

Result Verification

see section Performing result verification for TestCase execution on page 80.

Merging Coverage or Execution Results

TestConductor offers functionality to merge existing coverage or execution results. Execution, model and code coverage results are represented by html reports as controlled files in the model. Several of these result reports can be merged into one combined result report.

Requirement coverage results are represented directly in the model using dependencies and tags. Also for requirement coverage results merging of results into a combined requirement coverage result is supported by TestConductor.

Merging test execution reports

This function can be invoked using the menu helper 'Merge Test Execution Results'. The helper is available on TestPackages, TestContexts, TestSets and TestCases and supports multi selection. After invocation, the helper collects all test execution results of the TestCases below the selected test element(s) and merges them into one combined test execution result which is added to the model.
The combined result is added to the joint parent TestPackage of the selected test elements (if exist) or to a TestPackage 'MergeTestExecutionResults' if the joint parent of the selected test elements is the project itself.

When collecting the execution results of the included TestCases, also results at a customized result location (see page 128) are considered (even if they are located outside of the selected test elements). The collection of merged test execution results included existing and expected but not existing results (see below).

The combined result contains summary information for the merged results, including the total number of included TestCases and results and the overall verdict of the merged results (which is the worst verdict of the included results; passed < failed < inconclusive < error). Also, the number of passed, failed, inconclusive or error result verdicts are listed. If TestSets are selected, a TestCase is considered once for each TestSet including this TestCase (which means, one TestCases can be considered more than once if it is included in multiple selected TestSets).

Below the summary, the details of the merged execution results are presented. This section lists the results with their verdicts and also the test elements and CG configurations of their corresponding test architecture. TestSets are listed only if they have been selected. For TestSets, it is also shown if they have a dedicated CG configuration or not. The items in this section of the html report contain links which can be used for navigation to the model elements. For details of this functionality, see Traceability of Coverage Items on page 102.

Rules for collecting the expected execution results: Only results of TestCases are considered. For each TestCase, execution results for each CG Configuration of its test architecture are expected to exist. If a TestSet with a dedicated CG Configuration is selected, then only results for this CG Configuration are considered (and expected). If a

selected TestSet does not have a dedicated CG Configuration, then the same rule as for other kinds of test elements is applied.
If any expected execution result does not exist, then its verdict is inconclusive.

Merging model coverage reports

This function can be invoked using the menu helper 'Merge Model Coverage Reports'. The helper is available on TestPackages and on individual ModelCoverageResults 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.
If one TestPackage is selected, the combined report is added to this TestPackage. If multiple TestPackages or individual ModelCoverageResults 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.

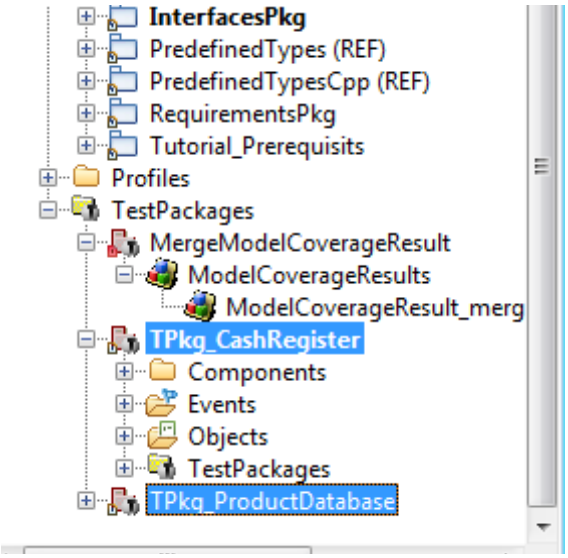


Figure 41: Merging ModelCoverage Results

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).

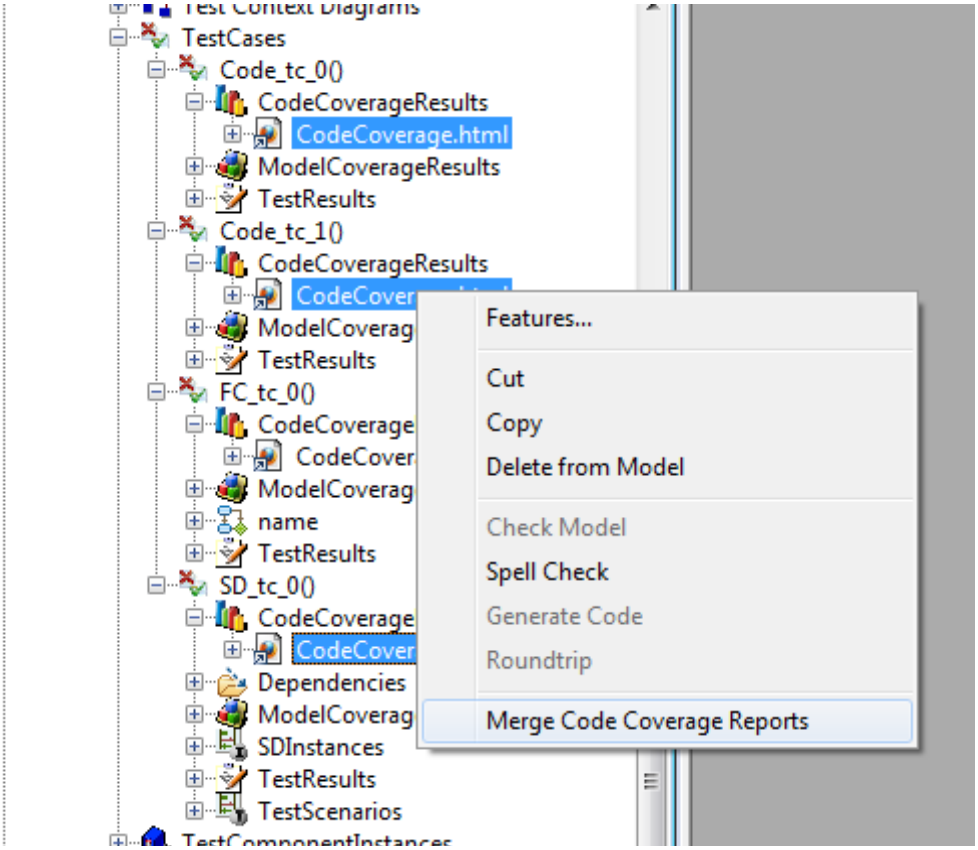


Figure 42: Merging CodeCoverage Results

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 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 Plugin for IBM Engineering Test Management

TestConductor TestCases can be referenced and executed from IBM Engineering Test Management. A detailed description how to integrate IBM Engineering Test Management and TestConductor can be found

In the document “ETMTestConductorAdapter_HowTo.pdf” in <Rhapsody installation>/Doc/pdfbooks.

To improve the integration between TestConductor and IBM Engineering Test Management, this plugin introduces the possibility to directly create and link IBM Engineering Test Management TestScripts while working with Rhapsody and TestConductor. An additional Helper 'Create ETM TestScript' is available which is applicable on TestCases, TestContexts and TestPackages.

After running the helper, the user has to specify the IBM Engineering Test Management server to connect to, user login and password for the server as well as the ProjectArea where the TestScript should be created.

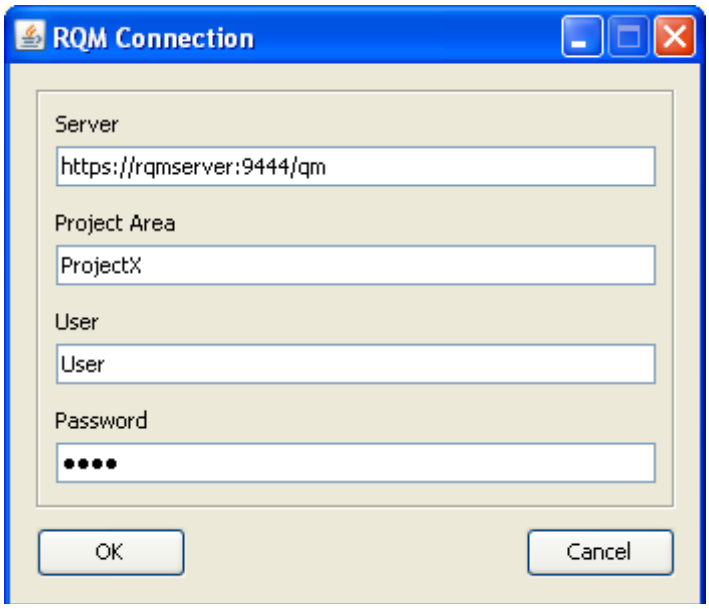


Figure 43: IBM Engineering Test Management Connection

After that, a 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 adaptions have to be done manually.

In IBM Engineering Test Management, the TestScript can now be executed using the TestConductor Adapter as described in the document "ETMTestConductorAdapter_HowTo.pdf"

Also a Hyperlink to the newly created TestScript is added automatically underneath the model element for which the helper has been called. Following the Hyperlink, the TestScript can be opened directly from Rhapsody.

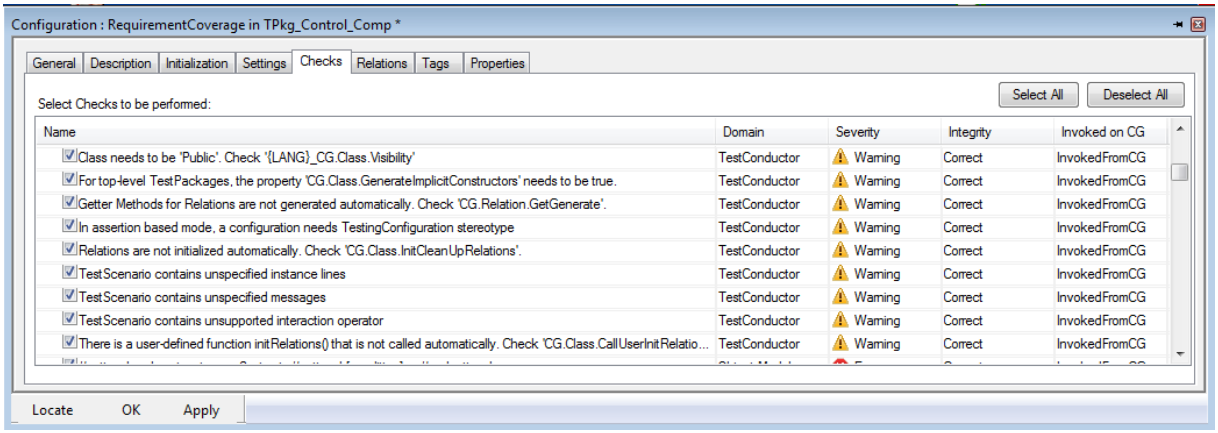
TestConductor Check Model Plugin

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 `<<AssertionBasedTestingConfiguration>>` or `<<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 180
    1. TestArchitecture
    2. TestBehavior
- Rhapsody TestConductor (**RTC**) – detailed information on page 183
    1. TestArchitecture
        1. CppUnit (no longer supported)
        2. CUnit (no longer supported)
        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.

- ***TestSet***
  A TestSet represents a collection of TestCases belonging to the same TestContext. TestSets are used to define sets of TestCases, which can be updated, build and executed at once by applying the respective command on the TestSet (cf. section Organizing TestCases in TestSetson page 394 ff). TestSets non-exclusively define subsets of the TestCases of one TestContext by selecting the respective TestCases using stereotyped dependencies (see stereotype <<TestSetElement>> on page 189). Optionally, a TestSet can determine using which code generation configuration it will be updated, built and executed. Therefore, a TestSet can own a stereotyped <<TestSetDedicatedConfiguration>> dependency on a code generation configuration of the TestArchitecture (cf. stereotype <<TestSetDedicatedConfiguration>> on page 189).
  If such a dependency exists and depends on a valid code generation configuration, then the referenced code generation configuration will automatically be activated on updating, building and executing the TestSet.

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 183)
  The TestArchitecture package contains further sub packages:
  - ***TestRT*** – stereotypes and types for using Test Realtime with TestConductor – support for TestRT integration with TestConductor is deprecated.
  - ***CppUnit*** – definitions for integration of the CppUnit testing framework with TestConductor – support for CppUnit integration with TestConductor ended with Rhapsody 8.4.
  - ***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 – support for CUnit integration with TestConductor ended with Rhapsody 8.4.
  - ***SDReuse*** – see section Creating Sequence Diagram TestCases from existing Scenarios using an explicit instance mapping on page 63 ff for a detailed explanation of the new terms defined in this sub package.
- *TestBehavior* (page 191)
- *TestDocumentation* (page 197)
- TestViewsAndQueries (page 199)

TestArchitecture Package

The *TestArchitecture package* contains the stereotypes:

- Sub package *CppUnit – CppUnit-Testing is not supported anymore.*
  Sub packages CppUnit and CUnit contain stereotypes for the integration of CppUnit and CUnit testing with Rhapsody.

  Note: Support for integration of TestConductor with CppUnit or CUnit ended with Rhapsody 8.4.

  - **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 not supported anymore.*
  Sub packages CppUnit and CUnit contain stereotypes for the integration of CppUnit and CUnit testing with Rhapsody.

  Note: Support for integration of TestConductor with CppUnit or CUnit ended with Rhapsody 8.4.

  - **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* (cf. section Creating Sequence Diagram TestCases from existing Scenarios using an explicit instance mapping on page 63 ff.)

  - ***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.

- **AnimationBasedTestingConfiguration**
  Stereotype for code generation configuration. Similar to stereotype TestingConfiguration for usage in animation based testing mode.

- **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.

- **AssertionBasedTestingConfiguration**
  Stereotype on code generation configuration. Base definitions of tags for assertion based testing mode (cf section Tags of the <<AssertionBasedTestingConfiguration>> and <<TestingConfiguration>> Stereotypes on 69 and figure Figure 9: Hierarchy of the code generation configuration stereotypes defined by the TestingProfile on page 69 for the inheritance relations of stereotypes regarding the testing mode).

- **AssociatedReplacementsPkg**
  Stereotype on dependency. For Rhapsody in Java, each TestArchitecture is associated with a TestPackage containing the replacement classes and objects used as TestComponents, TestActors and as GreyBox SUTs. The TestContext has a `<<AssociatedReplacementsPkg>>` dependency on the associated replacements TestPackage.

- **ControlArbiter**
  Stereotype ControlArbiter is used by TestConductor to mark a dependency of a SD TestCase on an Arbiter TestComponent that controls the SD TestCase.

- **ETPTargetTestingConfiguration**
  (cf section Tags of the <<AssertionBasedTestingConfiguration>> and <<TestingConfiguration>> Stereotypes on 69 and figure Figure 9: Hierarchy of the code generation configuration stereotypes defined by the TestingProfile on page 69 for the inheritance relations of stereotypes regarding the testing mode)

- **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.

- **MultiplicityConstantMap**
  Stereotype `<<MultiplicityConstantMap>>` is used for a model element in the TestContext which defines concrete values of symbolic multiplicity parameters (typically defines used for instance- or port- or association multiplicity specifications in the design, such as ‘NUMBEROFINSTANCES’) as a list of tags. Each tag must have the same name as the respective symbolic multiplicity parameter and defines the concrete value to be used by TestConductor for interpretation of the symbolic parameter (cf. section Multiplicity of Relations on page 29 and section Multiplicity of Relations on page 59, respectively).

- **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.
  Please see also Usage of hstart.exe to hide console application windows on page 208 regarding the helper tool used for this option on Windows machines.

- **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.

- **greyboxfilereplacement**
  Stereotype <<greyboxfilereplacement>> is used to mark a dependency of a greybox file (TestSUTFile) on the file that is replaced by the greybox file 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).

- **SCArbiter**
  Stereotype on Class. Used to mark a TestComponent to be the owner of a statechart defining a statechart TestCase.

- **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.

- **original**
  Stereotype marking original user defined processEvent() operation (RiC++ OXF, SXF), reactive consume event operation (RiC OXF) or root state dispatch event operation (RiC MXF, SMXF).
  For obsevation instrumentation of event receptions, TestConductor hooks into event processing in the individual class or object using mechanisms provided by the particular framework. To preserve already existing user defined hook functionality using the same framework mechanisms, TestConductor stereotypes the user defined hook <<original>> and adds a call to this <<original>> hook at the end of the observation hook (by default the appropriate framework event consumption is called at the end of the observation hook).

- **referenceObject**
  In RiC++, for instances of embeddable classes, one instance can be configured to be just a reference to another instance. If SUT or a TestComponentInstance shall refer to an instance, that already was instantiated before instantiating the TestContext, stereotype referenceObject can be set on the instance in the TestContext. Instantiation will then be turned of for this instance and the assignment to the instance identifier has to be realized in the constructor call of the TestContext.

- **referenceObjectPointer**
  For instances of non-embeddable classes, one instance can be configured to be just a pointer to another instance. If SUT or a TestComponentInstance shall refer to an instance, that already was instantiated before instantiating the TestContext, stereotype referenceObjectPointer can be set on the instance in the TestContext. Instantiation will then be turned of for this instance and the assignment to the instance identifier has to be realized in the constructor call of the TestContext.

- **referenceSameObject**
  When using referenceObjects or referenceObjectPointers in a TestArchitecture, it is helpful to document this methodology. To mark document the references among instances, it is recommended to add a <<referenceSameObject>> dependency on the referencedObject to the referring object.

- **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.

- **TestCounter**
  Stereotype to mark attributes as test artifacts. TestConductor introduces a set of counters in TestComponents and GreyBox SUTs which are used to count message occurrences along life-lines of TestScenarios during test execution. Based on these counters, the order and positions, as well as the number of calls of the specified messages are measured and compared to the TestScenario specification. These counters are temporary artifacts, they are introduced by Update TestCase/TestContext/TestPackage/TestSet and are removed by Clean TestPackage. Stereotype `<<TestCounter>>` is used for identification of such counters.

- **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 <<AssertionBasedTestingConfiguration>> and <<TestingConfiguration>> Stereotypes on 69 and figure Figure 9: Hierarchy of the code generation configuration stereotypes defined by the TestingProfile on page 69 for the inheritance relations of stereotypes regarding the testing mode)

- **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.

- **TestSetDedicatedConfiguration**
  Stereotype <<TestSetDedicatedElement>> is used to mark a dependency of a TestSet (see stereotype <<TestSet>> on page 181 and cf section Organizing TestCases in TestSets on page 94 ff). This optional dependency may depend on a code generation configuration of the same TestArchictecture to which the TestSet belongs. If a TestSet refers to a valid code generation configuration via a <<TestSetDedicatedConfiguration>> dependency, this dedicated configuration will automatically be activated on updating, building and executing the TestSet.

- **TestSetElement**
  Stereotype <<TestSetElement>> marks a dependecy of a TestSet on a TestCase belonging to the same TestContext as the TestSet itself. Using <<TestSetElement>> dependency the TestSet defines the set of TestCases that are grouped in the TestSet (cf section Organizing TestCases in TestSets on page 94 ff).

- **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_greyboxfilereplacement**
  Stereotype <<use_greyboxfilereplacement>> is used to mark a dependency of the TestContext on a replacement file (i.e. a greybox file replacement of a file in the design).

- **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.

- **TestArtifactContainer**
  To prevent the user design model from changes or modifications, TestConductor only writes to TestPackages, TestComponents, TestSUTs, e.t.c. Non-stereotyped model elements must not be affected by TestConductor. Stereotype <<TestArtifactContainer>> allows TestConductor to write to an Actor, Class or Package which is neither a protected user element, nor a test artifact with a strictly defined meaning.

- **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).

- **TestSUTFile**
  New term `<<TestSUTFile>>` is used to mark a replacement file (basically a copy of the original file – used only for Grey Box Testing).

- **TargetTestingConfiguration**
  Stereotype inheriting from and extending stereotype `<<TestingConfiguration>>`. Used for proxy based TestConductor support for executing TestCases on target hardware and simulators (for details see document Testing_with_TestConductor_on_a_small_target.pdf in the Doc/pdf_docs directory of the Rhapsody installation. cf also section Tags of the <<AssertionBasedTestingConfiguration>> and <<TestingConfiguration>> Stereotypes on 69 and figure Figure 9: Hierarchy of the code generation configuration stereotypes defined by the TestingProfile on page 69 for the inheritance relations of stereotypes regarding the testing mode).

- **WSTTargetTestingConfiguration**
  Stereotype inheriting from and extending stereotype `<<TargetTestingConfiguration>>`. Used for TestConductor support for executing TestCases on target hardware and simulators by Embedded UML Studio from SodiusWillert (cf section Tags of the <<AssertionBasedTestingConfiguration>> and <<TestingConfiguration>> Stereotypes on 69 and figure Figure 9: Hierarchy of the code generation configuration stereotypes defined by the TestingProfile on page 69 for the inheritance relations of stereotypes regarding the testing mode).

TestBehavior Package

The *TestBehavior pack*age is composed of a number of stereotypes like:

- **AppliedInvariantCheck**
  Stereotype (new term) on dependency of an InvariantCheckFuntion on a TestCase or TestContext. Using `<<AppliedInvariantCheck>>` dependencies, the user can define which InvariantCheckFunctions will be applied to which TestCase or TestContext executions (see section Specification of Invariants and Verification of Invariants during Test Execution on page 96).

- **InvariantCheckFunction**
  Stereotype (new term) on functions of the TestContext. `InvariantCheckFunctions` define assertions that will be checked frequently during test executions. An `InvariantCheckFunction` states that its particular expressions will be fullfilled globally and independently of the test specification during entire computations of the test application (see section Specification of Invariants and Verification of Invariants during Test Execution on page 96).

- **call_virtual**
  determines a message in a TestScenario to be a virtual operation call (see section Virtual Call vs Nonvirtual Call (Rhapsody in C++) on page 140).

- **call_nonvirtual**
  determines a message in TestScenario to be a non-virtual call (see section Virtual Call vs Nonvirtual Call (Rhapsody in C++) on page 140).

- **consideredTestCase**
  Stereotype on dependency. Used in RequirementCoverageResults to refer to the TestCases considered in requirement coverage computation.

- **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 (see section Computing Code Coverage on page 109 ff).

- **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**
  Stereotype on dependency. Used to establish a navigable relation from TestContext or TestCase to a CodeCoverageResult when reports are stored in a non-default location **(**see property `TestConductor::Settings::ReportLocation` (page 128) in section Settings – General Properties on page 126 ff).

- **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.
  (see section Computing Model Coverage during Test Execution on page 99 ff).

- **ModelCoverageResultRef**
  Stereotype on dependency. Used to establish a navigable relation from TestPackage, TestContext or TestCase to a ModelCoverageResult when reports are stored in a non-default location **(**see property `TestConductor::Settings::ReportLocation` (page 128) in section Settings – General Properties on page 126 ff).

- **RequirementCoverageResult**
  Stereotype on comment. Below this comment a requirement coverage result is organized as a sub-tree of model elements in the browser.
  Requirement coverage is based on Model coverage and can be enabled using tag **`ComputeRequirementCoverage`** on the TestingConfiguration for assertion based testing mode (see section Computing Requirement Coverage on page 102).

- **RequirementCoverageResultRef**
  Stereotype on dependency. Used to establish a navigable relation from TestContext or TestCase to a RequirementCoverageResult when reports are stored in a non-default location **(**see property `TestConductor::Settings::ReportLocation` (page 128) in section Settings – General Properties on page 126 ff).

- **ExecutedElement**
  New term on dependency. Used in CodeCoverageResults, ModelCoverageResults,

RequirementCoverageResults and TestResults to refer to the TestPackage/TestContext/TestCase for which they were generated.

- **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.
  For driven messages, tag RTC_ImmediateEvaluation defines how the message will be driven by the arbiter generated from the TestScenario. By default, the arbiter statechart waits for a minimal timeout before driving the next message. This allows the scheduling algorithm of the framework to schedule another statechart between two subsequently driven messages. If tag RTC_ImmediateEvaluation is set on a message to be driven, the arbiter statechart will not have to wait for a timeout before driving the respective message and, hence, no event of another reactive of the same task can be dispatched between e.g. two subsequently driven messages.
  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 153.
By default, TestConductor only generates DriverOperations for messages from TestComponents to the SUT. For messages from TestComponent to TestComponent, no DriverOperations are generated and such messages are not driven in the test execution. Tag “RTC_DriveTestComponentMessage” can be checked to change this behavior for individual messages.

- **RTC_OperatorInfo**
  Stereotype applicable on InteractionOperator (cf. section Using Interaction Operators in SD TestCases on page 162 ff).
  The execution semantics of Interaction Operators can be adapted by using the stereotype `<<RTC_OperatorInfo>>` and the tag `RTC_ImmediateEvaluation`. By default the arbiter statechart generated from a TestScenario will wait a minimal timeout before the operator is activated, i.e. before operand conditions are evaluated.This is 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, e.g. to observe the immediate reaction of the SUT on a driven message. To support this behavior, stereotype `<<RTC_OperatorInfo>>` can be applied on the Interaction Operator and tag `RTC_ImmediateEvaluation` of the stereotype can be checked.

- **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**
  applicable to comment. Used as root element for requirement coverage document generation using Publishing Engine.

- **considerForCoverage**
  Normally, the coverage scope is automatically computed w.r.t. to the tags "CoverageScopeIncludeParts", "CoverageScopeIncludeInheritance" and "CoverageScopeIncludeTestArchitecture" of the `<<TestingConfiguration>>` code generation configuration (cf. Tags of the <<AssertionBasedTestingConfiguration>> and <<TestingConfiguration>> Stereotypes on page 69 and section Choosing the Coverage Scope for Model Coverage on page 101). 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. When working with replacements, the dependency should refer to the replaced element, i.e. to the original element in the user model.

- **considerNotForCoverage**
  Normally, the coverage scope is automatically computed w.r.t. to the tags "CoverageScopeIncludeParts", "CoverageScopeIncludeInheritance" and "CoverageScopeIncludeTestArchitecture" of the of the `<<TestingConfiguration>>` code generation configuration (cf. Tags of the <<AssertionBasedTestingConfiguration>> and <<TestingConfiguration>> Stereotypes on page 69 and section Choosing the Coverage Scope for Model Coverage on page 101). 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. When working with replacements, the dependency should refer to the replaced element, i.e. to the original element in the user model.
  Note: `<<considerNotForCoverage>>` overrides the default scope in case of a conflict, i.e. excludes an element even though it belongs to the default coverage scope, according to the settings.

- **SynchronousTestAction**
  Stereotype to affect representation of general TestAction, TestAssignment and TestCondition.
  By default, TestConductor always waits a minimal delay before executing or evaluating a general TestAction, TestAssignment or TestCondition. By stereotyping such an element `<<SynchronousTestAction>>`, TestConductor omits the minimal dely and executes or evaluates the affected element immediately after the preceding action or observation.

- **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
  - <StubChecksAction> - RTC_StubChecksCode

  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 160 and Influencing DriverOperation and Stub generation using TestActions in TestScenarios on page 157 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 160 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 160 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**
  Stereotype on dependency. Used to establish a navigable relation from TestPackage, TestContext or TestCase to a TestResult when reports are stored in a non-default location **(**see property `TestConductor::Settings::ReportLocation` (page 128) in section Settings – General Properties on page 126 ff).

- **TestScenario**
  The stereotype *TestScenario (TestScenario)* contains two tags *RTC_ActivationCondition* and *RTC_SDParameters*. 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_SDParameters 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 140).

- **WitnessScenario**
  TestScenario witnesses obtained by 'Show as SD' (cf. section Failure Analysis using Witness Scenarios on page 169) or generated due to `<<TestingConfiguration>>` tag `CreateWitnessScenariosForFailedSDTestCase` (cf. section Tags of the <<AssertionBasedTestingConfiguration>> and <<TestingConfiguration>> Stereotypes on page 69) are stereotyped `<<WitnessScenario>>`.

- **IgnoreSuperclassOperations**
  For C++, it is possible using referenceObjectPointer instances to use separate life-lines for inheriting classes and their base-classes in TestScenarios, in order to specify communication along the inheritance hierarchy. IgnoreSuperclassOperation (applicable to TestCase) forces TestConductor to not accept message realizations in base classes on the life-line of a specialization in order to avoid mixing up virtual and non-virtual calls.

- **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 53 and 54, respectively) may produce not compilable code under some circumstances. I 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 pack*age 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:

- **InvariantCheckMatrix**
- **TestRequirementMatrix**
- **TestRequirementResultMatrix**
- **TestSetMatrix**
- **InvariantCheckTable**
- **TestResultTable**

- **TestCaseResultTable**
- **TestSetTable**
- **RequirementCoverageTable**

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.
A TestRequirementMatrices provides information about 'static' TestObjective relations in the scope of the matrix

A *TestRequirementResultMatrix* shows in an array view the execution state of requirement coverage by TestCases. The left hand side of the array shows all existing RequirementCoverageResults. The upper side shows all the requirements. The cells contain three kinds of entry, showing if a RequirementCoverage result fully or partially covers the requirement or does not cover the requirment at all,
A TestRequirementResultMatrix provides information about the 'dynamically' computed coverage relation according to model element coverage by TestCases and traceability information among model elements and requirements (see section Computing Requirement Coverage on page 102 ff for configuration capabilities of requirement coverage measure).

A *TestCaseResultTable* shows in a table form the owning TestPackage, the TestCase name, its verdict(s) (if existing – TestCases have no TestCaseResult and no verdict unless executing them) and the configuration(s) for which the TestCase has been executed.

A *TestResultTable* shows in a table form the existing TestCaseResults and their current result values (verdicts). The left column of the table shows all existing TestCaseResults. The right column shows the current TestCase verdicts. Note, that not executed TestCases are not regarded in this table, since they do not have an TestCaseResult.

A RequiremenCoverageTable is organized by requirements as rows of the table. Each row shows for a particular requirement by which model element the requirement is referenced, by which TestCases or TestContexts or TestPackages the requirement is fully and partially, respectively, covered and by which by which TestCases or TestContexts or TestPackages the requirement is not covered at all. The last column shows which TestCases refer to the requirement by TestObjectives.

InvariantCheckMatrix and InvariantCheckTable can be used to document but also to define, which InvariantCheckFunction has to be regarded for which TestCases. (see Specification of Invariants and Verification of Invariants during Test Execution on page 96 ff for details).

TestSetMatrix and TestSetTable are used to visualize and modify TestSets in a TestArchitecture A TestSetMatrix shows the TestSets in the from-scope of the matrix view as rows and the TestCases in the to-scope of the matrix view as rows. Cell elements in the matrix view are <<TestSetElement>> dependencies. The matrix can not only be used to visualize the TestSets in the scope, but also to easily add or remove <<TestSetElement>> dependencies by modifying the cells of the matrix.
A TestSetMatrix does not show the <<TestSetDedicatedConfiguration>> dependencies of

TestSets. In order to visualize and modify these, a TestSetTable can be used (cf section Organizing TestCases in TestSets on page 94 ff).

All these matrices and and tables can be used as is for documentation purposes. By copying the matrix- or table-layout to a package in the model and modifying the layouts, the layouts can be adapted to the individual needs of the user or project, respectively. Note, that for all the matrix- and table-layouts in the TestingProfile, there is also a stereotype (new term) definition on the respective matrix- and table-view. This allows the user to simply add the desired view with preselected layout by just invoking e.g. “Add New->TestingProfile->TestSetMatrix” from the context menu of a TestPackage, without first adding a view and then defining the layout manually.

TestViewsAndQueries

This sub-package of the TestingProfile provides some predefined model queries that can be applied to search the model and, in particular, to restrict the browser view to customized views.
Queries provided by the TestingProfile are:

- TestCases
- TestConfigurations
- TestContexts
- TestResults
- TestSets

By default, the Rhapsody model browser shows the ‘Entire Model View’. Views can be changed using the drop down in the upper left corner of the browser window.
When the TestingProfile is loaded, the drop down selection offers the predefined views listed above. These views can easily be modified and adapted to the users needs by copying a query to another location in the model (outside the read-only profile) and then adjusting the query criteria according to the needs.
By adding additional queries, customized views can be created very easily.

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 *48* and in chapter *TestCase Definition with Flow Charts on page 49* and in chapter TestCase Definition with Statecharts on page 49, 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." [67]

- RTC_ASSERT_SD_NAME_ACTUAL(`sd_instance_name, msgid, p, e, a`)[68]
  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[69].

- 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*.

[67] 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)`'.

[68]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)

[69] 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 != 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 != e2`). Otherwise the result of the assertion is *FAILED*.

TestConductor Assertion Functions (Java)

As described in chapter TestCase Definition with Code on page *48* and in chapter *TestCase Definition with Flow Charts on page 49* and in chapter TestCase Definition with Statecharts on page 49, pre-defined assertion functions are used to get results from a TestCase execution.

TestConductor defines several static assertion functions in class com.btces.testconductor.assertionbased.TestConductor. These functions are listed below. Each function 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

- ASSERT (boolean `e`)
  Assertion with default name e. The assertion is PASSED, if the result of the boolean expression is true (`e!=false`), otherwise the assertion FAILED.

- ASSERT_FATAL (boolean `e`)
  Assertion with default name. The assertion is PASSED, if the result of the boolean expression is true (`e!=false`), otherwise the assertion FAILED. If it is failed, the TestCase is aborted immediately without executing further assertions.

- ASSERT_NAME (String `n`, boolean `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!=false`), otherwise the assertion *FAILED*.

- ASSERT_NAME _FATAL(String `n`, boolean `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!=false`), otherwise the assertion *FAILED*. If it is failed, the TestCase is aborted immediately without executing further assertions.

- ASSERT_SD (String `sd_instance_name`, String `msgid`, boolean `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 (String `sd_instance_name`, String `msgid`, String `p`, boolean `e`)
  Similar to 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." [70]

- ASSERT_SD_NAME_ACTUAL(String `sd_instance_name`, String `msgid`, String `p`, boolean `e`, `T a`)
  where `T` is the type of actual value `a` for types that can be casted to `Object`. For scalar types there are dedicated overloaded functions available for: `boolean, byte, char, double, float, int, long, short`
  Similar to 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[71].

- ASSERT_TRUE (String `n`, boolean `e`)
  This assertion with name `n` is *PASSED*, if `e == true`. Otherwise the result of the assertion is *FAILED*.

- ASSERT_FALSE (String `n`, boolean `e`)
  This assertion with name `n` is *PASSED*, if `e == false`. Otherwise the result of the assertion is *FAILED*.

- ASSERT_EQUAL (String `n`, boolean `e1`, boolean `e2`)
  This assertion with name `n` is *PASSED*, if `e1 == e2`. Otherwise the result of the assertion is *FAILED*.

- ASSERT_NOT_EQUAL (String `n`, boolean `e1`, boolean `e2`)
  This assertion with name `n` is *PASSED*, if `e1 != e2`. Otherwise the result of the assertion is *FAILED*.

Using IntelliVisor for TestConductor Assert Macros/Functions

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, depending on the used programming language:

- Close Rhapsody if it is open.
- For Rhapsody in C++

[70] 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)`'.

[71] 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)`

- Copy the file CPP_C_IntellisenseTC.prp from the Rhapsody\TestConductor folder to the UserShare\Properties folder of your Rhapsody installation.
- Open the siteC++.prp file and add
  Include "CPP_C_IntellisenseTC.prp"
- Save the siteC++.prp file and open Rhapsody.

- For Rhapsody in C
  - Copy the file CPP_C_IntellisenseTC.prp from the Rhapsody\TestConductor folder to the UserShare\Properties folder of your Rhapsody installation.
  - Open the siteC.prp file and add
    Include "CPP_C_IntellisenseTC.prp"
  - Save the siteC.prp file and open Rhapsody.
- For Rhapsody in Java
  - Copy the file Java_IntellisenseTC.prp from the Rhapsody\TestConductor folder to the UserShare\Properties folder of your Rhapsody installation.
  - Open the siteJava.prp file and add
    Include "Java_IntellisenseTC.prp"
  - Save the siteJava.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 Assertions. Selecting one of the macros also shows a hint that gives you information about the parameters of the macro.

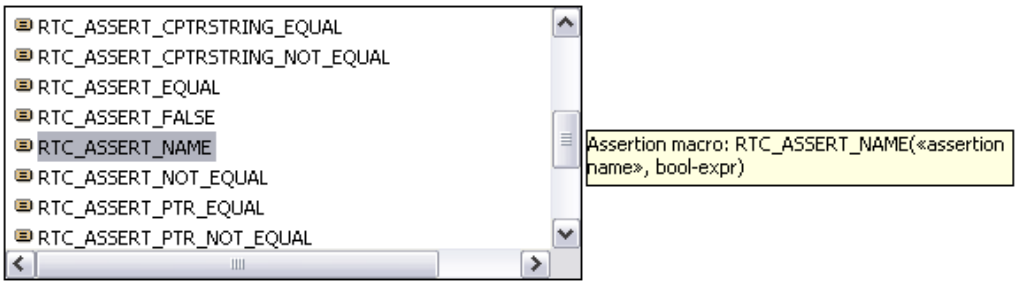


Figure 44: IntelliVisor with TestConductor Assert Macros, C++ example

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.

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.
      For language Java: 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.

- 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)”.
  For language Java: In the Action of the transition, change line “itsTCon.rtc_exit()” to “itsTCon.finishTestCase()”.

- 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'.
Since introduction of assertion based testing mode for Java (with Rhapsody 9.0.1), automatic migration is applicable also for Java models.

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.

Usage of *hstart.exe* to hide console application windows

When executing TestConductor tests the tested application and the tools responsible for the generation of the test result files are running in a console window. When executing several tests in a row the continuous opening and closing of application windows can be inconvenient and prevent interactive working on a machine while testing in the back ground at the same time. TestConductor can optionally hide the application windows by using the additional tool hstart.exe that can start other console applications with a hidden window as a starter for the test application and the result generation tools.

This helper tool *hstart.exe* is installed to the Share/etc folder of the Rhapsody installation, a small description, usage information and a link to the web site providing this tool can be found in the file hstart_ReadMe.txt in the same folder. If the option to run tests without a console window is enabled when testing on a Windows machine then TestConductor is using hstart.exe with the parameter /NOCONSOLE to start the test application and the result generation tools.

The tool hstart.exe is considered a security problem by some antivirus and security checking tools. If the option to hide the console window during test execution is not used then the tool hstart.exe can safely be removed from the installation without losing TestConductor functionality except for execution without consoles.

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.
- Stubs and Assertions generated by TestConductor are not thread safe.
- Before Rhapsody 9.0 stubbing of static member operations wasn't supported at all. Although stubbing of static member operations is now supported for Rhapsody in C++, a similar construct for Rhapsody in C by emptying property C_CG.Operation.MeDeclType for a member operation is not supported. Since static operations have no access to member attributes and there are no class static attributes in Rhapsody in C, counters for message occurrences would have to be realized and addressed as global counter variables, which is not supported by TestConductor.
- Untriggered initial behaviors can not reliably be tested by TestConductor. TestCases are started by the TestCaseScheduler, which in turn is itself a reactive class. Its statechart is initialized and started on system startup concurrently to all other statecharts in the system under test. Untriggered initial behavior, e.g. actions at default transitions are executed on starting the behaviors, whereas TestCases start stimulating and observing only after they were started by the TestCaseScheduler.
  In order to avoid such races, it is recommended to model statecharts waiting on a dedicated start event (cf. section Testing untriggered initial behavioral on page 165).
- For static inheritance, TestConductor offers specific support by lifting the concept to the modeling level (see section Special Case: Static InheritanceSpecial Case: Static Inheritance on page 39)
- TestConductor does not support stubbing of functions with variable argument lists.
- TestConductor does not support UseCases as SUT or TestComponent. Although UseCases can more or less be used like classes, i.e. defining statecharts or having operations etc., TestConductor largely ignores UseCases.
- TestConductor does not support Actors as SUT. Actors can be used in TestComponents place but can not be tested themselves.
- TestConductor offers no support for Templates as SUT or TestComponents. There is specific support of classes inheriting from templates using generalizations wit template parameter bindings (see section Special Case: Class inheriting from Template on page 38).
- There is no GreyBox TestArchitecture creation for Files in TestConductor for models in Rhapsody in C.
- RiCString is not supported as argument in messages of SD TestCases for Rhapsody in C.
- Even though TestConductor supports {Lang}_CG.Package.GenerateDirectory and CPP_CG.Package.DefineNameSpace, TestArchitecture creation may become compilcated and manual corrections and adaptions ay become necessary, since TestConductor has to regard package hierarchies when creating replacements in

the TestArchitecture. TestConductor has to model TestPackage hierarchies according to the original package hierarchies to create replacements in the same model paths as their replaced original elements.
Since code generation artefacts of elements from ‘same model paths’ will be merged in the same files by code generation, undesired effects may appear in code generation and build.

- Stubbing of ‘inline’-operations for Rhapsody in C, testing ‘inline’ operations of a GreyBox SUT: if checkbox ‘inline’ is checked in the features dialog for an operation, Rhapsody generates a macro definition in the header-file for the owning class. This means, that the body of the operation definition must not contain return statements. Hence, the body of an operation has to be defined differently wrt. usage as macro or usage as a regular operation.
  In order to instrument inline operations for observation or to stub inline operations, TestConductor unsets the inline flag and treats ‘inline’ operations as regular operations - inlining is just a compiler optimization and normally doesn’t affect the behavior of the operation.
  This approach works well for C++, but is not applicable for Rhapsody in C (cf. section Replacements on page 23). If your compiler supports real inlining, consider overriding property `C_CG::WriterTemplates::OperationAsMacro` as described in the footnote on page 25.

- GreyBox testing only regards one level of decomposition.

- Implicit parts can not be instantiated as SUT. TestArchitectures can only be created for their instantiating parent class or object.
  Implicit parts of GreyBox SUTs can not be instrumented and hence not be observed in GreyBox testing.

- Copy/Paste of TestCases and other model elements in a TestArchitecture shall always be performed only after ‘Clean TestPackage’.
  ‘Update TestCase/TestSet/TestContext/TestPackage’ generates temporary artifacts, such as DriverOperations and StubOperations, as well as dependencies among these artifacts for navigation purposes. By copying e.g. TestCases in an uncleaned TestArchitecture, also these artifacts and their dependencies will be partially copied. This brings the TestPackage into an inconsistent status and might cause problems in subsequent Update and Build.
  ‘Clean TestPackage’ will solve this inconsistency. ‘Clean TestPackage‘ before copy/pasting will avoid such inconsistencies.

- TestConductor is widely incompatible with property ‘CG.CGGeneral.GeneratedCodeInBrowser’. When ‘CG.CGGeneral.GeneratedCodeInBrowser’ is turned on, auto-generated functions appear in the browser and can’t be overridden as usual. TestConductor will then fail to generate needed artifacts, such as helper functions for the initialization of bidirectional associations involving replacements, stubbed setters for dataflows and other artifacts. When working with TestConductor, it is strictly recommended, not to use ‘CG.CGGeneral.GeneratedCodeInBrowser’.

- When using Rhapsody in C with the C89 or C90 language standard, model coverage cannot be computed if a flowchart of an operation uses local variables. The instrumentation by TestConductor adds function calls before the local variable declaration causing compile errors when using the C89 or C90 standards (for example, VS 2012 uses C89 per default).

- Verification of invariants with flowchart or statechart based TestCases is not supported by result verification.

List of Figures