Rational. Rhapsody

IBM

IBM® Rational® Rhapsody® TestConductor Add On

User Guide

Rhapsody®

IBM® Rational® Rhapsody® TestConductor Add On

User Guide

Release 2.8.2

License Agreement

Trademarks

Contents

Content

Document Structure

This user guide is organized as follows:

- **Chapter 1, Introductio**n, provides an introduction to IBM® Rational® Rhapsody® TestConductor Add On through a high-level overview of the main features.
- **Chapter 2, Rhapsody UML Testing Profile,** describes the defined stereotypes and new terms which can be used for the definition and management of tests.
- **Chapter 3, Model-based Unit Test Definition,** explains how to create Test Architectures and how to define test cases with sequence diagrams, statecharts, flow charts, or pure code.
- **Chapter 4, Test Execution,** explains how to build and execute a test configuration.
- **Chapter 5, Test Management,** guides you through the process of creating and editing the entire test suite.
- **Chapter 6, Upgrading old TestConductor Test Cases,** describes the process of upgrading of existing test definitions from older TestConductor versions.
- **Chapter 7, Advanced Test Definition,** describes the powerful features of sequence diagram test case definition like ordering, parameter mapping, activation conditions, etc.
- **Chapter 8, Failure Analysis,** explains how to analyze the source of a possible failure (after you have made design extensions and modifications).
- **Chapter 9, Using TestConductor from Eclipse,** explains how to use TestConductor when working with Rhapsody in Eclipse platform integration.
- **Chapter 10, Using TestConductor from Rational Quality Manager,** explains how to create test scripts and test cases in RQM for executing test cases with TestConductor.
- **Chapter 11, TestConductor Rhapsody Plugins,** describes how to use additional TestConductor plugins for Rhapsody.

Contacting IBM® Rational® Software Support

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

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

For Rational software product news, events, and other information, visit the IBM Rational Software Web site.

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

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

What software versions were you running when the problem occurred?

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

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

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

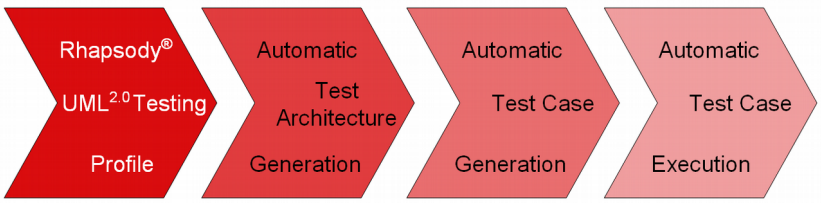
Conventions

The following table lists the conventions used in the Rhapsody documentation.

Style	Description
command1 > command2	The greater-than (>) symbol leads you through the steps in a menu or key sequence. For example, **Add New > Package** means that you should first select **Add New**, then select **Package** from the **Add New** submenu.
Bold type	Bold type indicates items that you should select, such as buttons or checkboxes in dialog boxes. For example: Click **Apply**
Italic type	*Italic type* is used for emphasis, titles of referenced documents and new terms.
`Courier type`	`Courier type` is used for file names and directory paths, user input, and code-related items such as instance names and properties.
<filename>	Angle brackets surround variable names that you should replace with actual names. For example, you should replace <filename> with the actual name of a file.

Introduction

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



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

In the Rhapsody Testing Environment the implementation of test cases 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 test cases 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. In Rhapsody in C++, Rhapsody in C, Rhapsody in Java, and Rhapsody in Ada test cases can be defined also by statecharts, flow charts (only C/C++), or pure code. Using sequence diagram related test cases, 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. TestConductor supports Rhapsody in C++, Rhapsody in C, Rhapsody in Java and Rhapsody in Ada. Limitations regarding the different languages can be found in the chapter Restrictions.

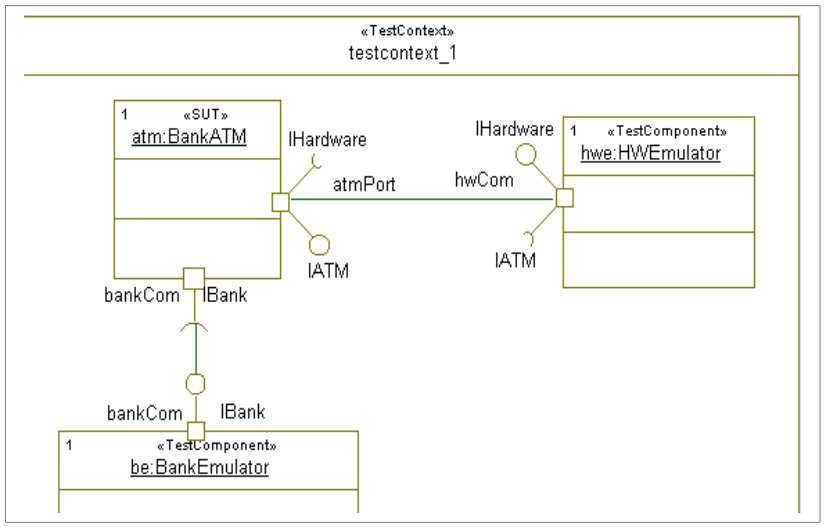
This document focuses on the animation based testing mode, which is applicable to Rhapsody in Java, Rhapsody in Ada, Rhapsody in C++ and Rhapsody in C. When using Rhapsody in C++ or Rhapsody in C++ the assertion based testing mode is recommended which is explained in details in the “TestConductor_User_Guide.pdf” document.

Rhapsody UML Testing Profile

The Rhapsody UML 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.

Automatic Test Architecture Generation

The *automatic test architecture 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.



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

- Creation of a new test package
- Creation of a new test context including
    1. System under test (“SUT”)
    2. Test components
    3. Links between SUT and test components

Test Case Definition

A *test case* represents the smallest element that can be defined and executed by TestConductor. A test case describes a sequence of input stimuli and expected behavior, in

order to verify a certain functional behavior of a system under test. Test cases can define both, black box and white box behavior.

TestConductor supports several ways to define test cases:

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

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

Test Case Execution

TestConductor is a *test case execution engine* and represents the second stage of the Rhapsody Testing Environment. It enhances the testing capabilities by not only executing the automatically generated test architecture, but it also offers a test execution analysis with respect to the expected results. If the test case 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

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 test architecture. Therefore, the project with the system under test must be in a state such that you can compile and run the test architecture, just as you must do to use the interactive simulation capabilities of Rhapsody. If you are using TestConductor with testing mode “AnimationBased” (property TestConductor.Settings.TestingMode), you must compile the code of at least the test components with animation instrumentation.

Note: For Rhapsody in Ada, make sure that you rebuild Rhapsody’s framework before using TestConductor. To rebuild the framework, select “Build framework” from Rhapsody’s code menu (after opening an Ada model). You only have to rebuild the framework once.

Note: If you are using TestConductor with testing mode “AnimationBased” (property TestConductor.Settings.TestingMode), make sure that you have compiled and linked an executable component with animation instrumentation.

Note: If you are using TestConductor with testing mode “AnimationBased” (property TestConductor.Settings.TestingMode), make sure that the properties `CG::Operation::Animate`, `CG::Operation::AnimateArguments`, `CG::Event::Animate`, and `CG::Event::AnimateArguments` of those messages used for test execution based on sequence diagrams are switched on. Otherwise they are not animated and cannot be tested with TestConductor. Ensure this for the properties of these relevant messages, and also for their parent class and package properties.

This guide uses sequence diagrams that are included (or have to be additionally created) in the CashRegister sample. The chapter *Advanced Test Definition* uses sequence diagrams from the PBX sample. Both samples do not provide step-by-step information.

Rhapsody
UML Testing Profile

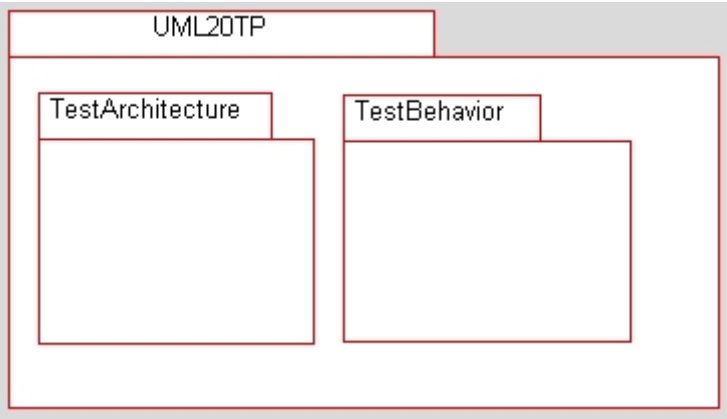
The *Rhapsody UML Testing Profile* is based on the official UML Testing Profile. It contains new terms and stereotypes that can be utilized for model testing artifacts in Rhapsody. A couple of elements defined in the UML Testing Profile are presently not part of the Rhapsody Testing Profile. However, 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 not addressed by the UML Testing Profile.

For further information on the Rhapsody UML Testing Profile please refer to the *TestConductor Tutorial*, where depict examples on the Rhapsody Testing Profile are provided. Hence, it is recommended to utilize the TestConductor Tutorial for training purposes prior to going into further detail in this document.

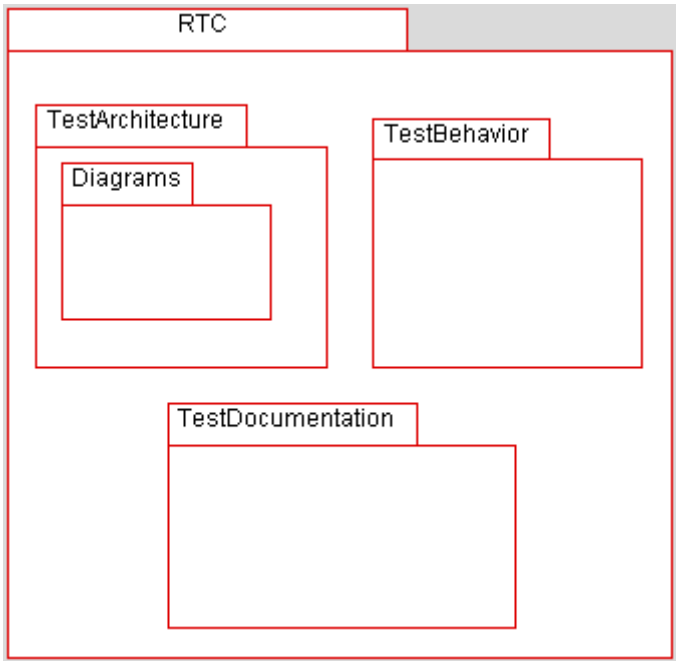
Structure Overview

The Rhapsody Testing Profile is prearranged in three major packages with additional sub-packages and the *TestingProfile* stereotype.

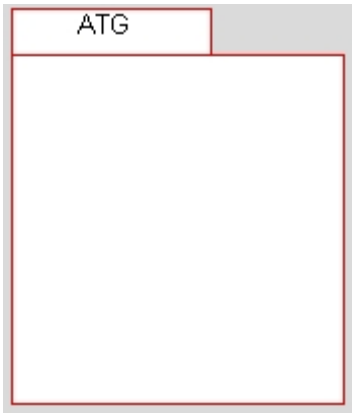
- Rhapsody UML Testing Profile (UML20TP)
    1. *TestArchitecture*
    2. *TestBehavior*



- Rhapsody TestConductor (RTC)
    1. *TestArchitecture*
    2. *TestBehavior*



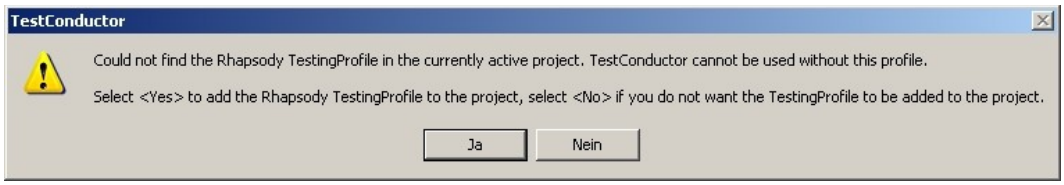
- Automatic Test Generation (ATG)



Adding the Testing Profile automatically

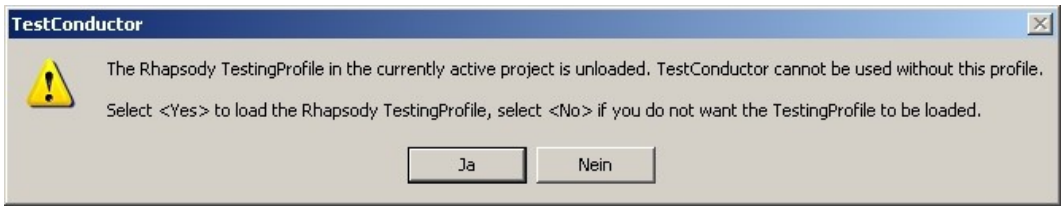
The first usage of any TestConductor functionality automatically adds the Rhapsody Testing Profile to a model. For example this can be done by choosing the Rhapsody menu entry **Tools > TestConductor**.

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



Select **Yes** to add the Rhapsody Testing Profile to the model. Select **No** to abort this process.

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

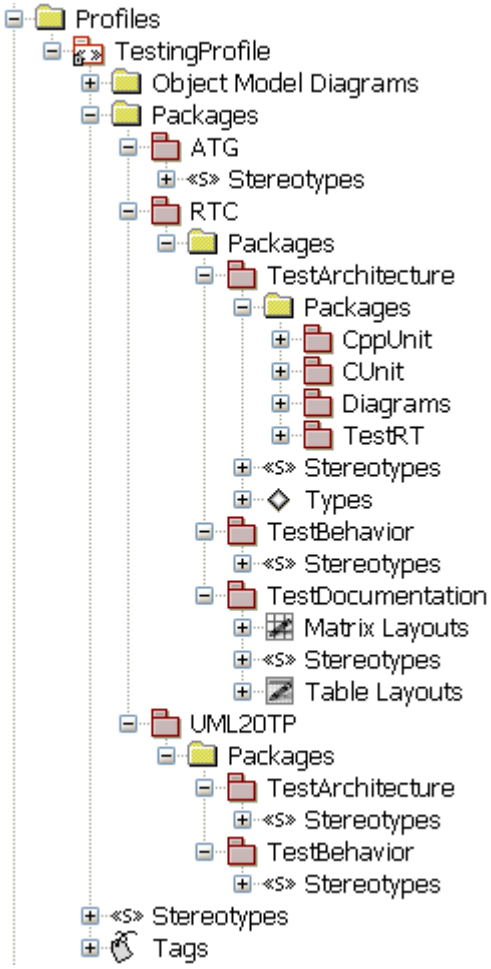


Select **Yes** to load the Rhapsody Testing Profile to the model. Select **No** to abort this process.

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

Select **OK**. After removing the existing profile with name TestingProfile from the model redo the action to start Rhapsody TestConductor.

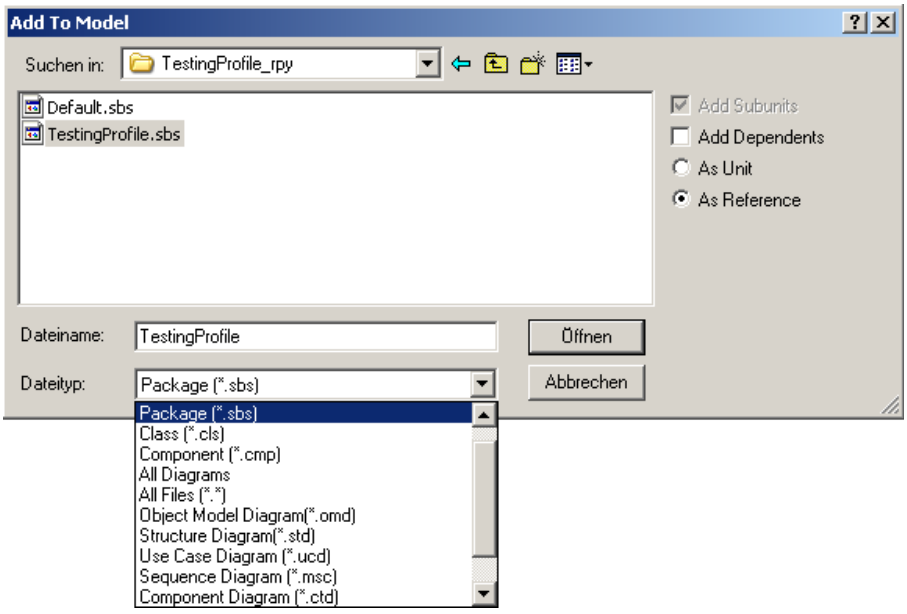
Once the Rhapsody Testing Profile has been loaded into a Rhapsody project by starting TestConductor the Rhapsody browser window will contain the above stated testing profile packages and its individual sub-packages as shown in the following picture.



Adding the Testing Profile manually

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

- Open your project in Rhapsody
- Select the menu item **File > Add Profile to Model…**
- Select the following **Data Type**: ‘Profile (*.sbs)’



- Select in Rhapsody installation folder: `'...\Share\Profiles\TestingProfile\TestingProfile_rpy\TestingProfile.sbs'`
- Press **Open** to add the Rhapsody Testing Profile to the model.

Functional Specification

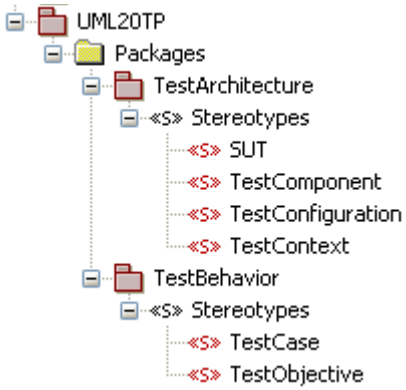
The *functional specification* of the Rhapsody Testing Profile shall be explained by means of its structure stated in the previous chapter *Structural Overview*.

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*

as shown in below picture.



TestArchitecture Package

The *TestArchitecture package* consists of the stereotypes

- *SUT*
- *TestComponent*
- *TestConfiguration*
- *TestContext*

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 test components, the test context or the system environment (ENV).

A *test component (TestComponent)* is a class of a test system. The *test component objects* (*TestComponentInstances*) realizes partially the behavior of a test case. An instance of a test component may have a set of interfaces which are used to communicate via connections with other test component 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 *stub operations (StubOperations)* which are able to generate necessary “stub” return values.

The *test configuration (TestConfiguration)* is a dependency to a code generation configuration. Depending on this configuration the code for the complete test context including its test cases can be generated, built and executed.

A *test context (TestContext)* describes the context in which test cases are executed. A test context is responsible for defining the structure of the test system, i.e., which test component objects and which SUT objects exists and how they are interconnected. The test component instances and SUT objects are normally parts of a test context. Since test cases are operations of a test context, a test case can access both the test component instances and also the SUT objects.

TestBehavior Package

The *TestBehavior package* contains two stereotypes named

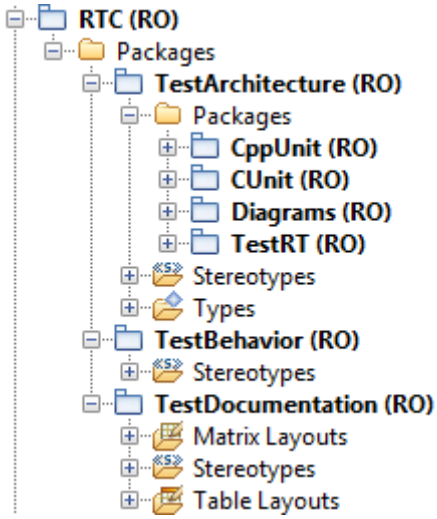
- *TestCase*
- *TestObjective*

A *test case (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 test case is an operation of a test context (described above).

A *test objective (TestObjective)* is a named element describing what should be tested. It is associated to a test case.

TestConductor (RTC) Package

The RTC package consists of three major packages: *TestArchitecture*, *TestBehavior* and *TestDocumentation* as shown in below picture.



TestArchitecture Package

The *TestArchitecture package* contains the stereotypes:

- Subpackage *CppUnit*
  - *CppUnitConfig*
  - *CppUnitContext*
- Subpackage *Cunit*
  - *CUnitConfig*
  - *CUniContext*
- Subpackage *Diagrams*
  - *TestContextDiagram*
- *AUTOSAR_RTE*
- *AUTOSAR_RTEInstance*
- *Arbiter*
- *ArbiterInstance*
- *ControlArbiter*
- *instantiated*
- *usedSUTObject*
- *usedTestComponentObject*
- *NoConsoleApp*
- *ParameterTable*
- *replacement*
- *greyboxreplacement*
- *greyboxinstancereplacement*
- *instancereplacement*

- *filereplacement*
- *scheduled*
- *Scheduler*
- *SCTCInstance*
- *stubbed*
- *Stub*
- *TestActor*
- *TestFile*
- *TestComponentInstance*
- *TestComponentObject*
- *TestingConfiguration*
- *TestPackage*
- *TestParameter*
- *TestLink*
- *use_ParameterTable*
- *use_replacement*
- *use_greyboxreplacement*
- *use_greyboxinstancereplacement*
- *use_instancereplacement*
- *use_filereplacement*
- *TestSUT*
- *TestSUTObject*

Subpackages *CppUnit* and *CUnit* contain stereotypes for the integration of CppUnit and CUnit testing with Rhapsody.

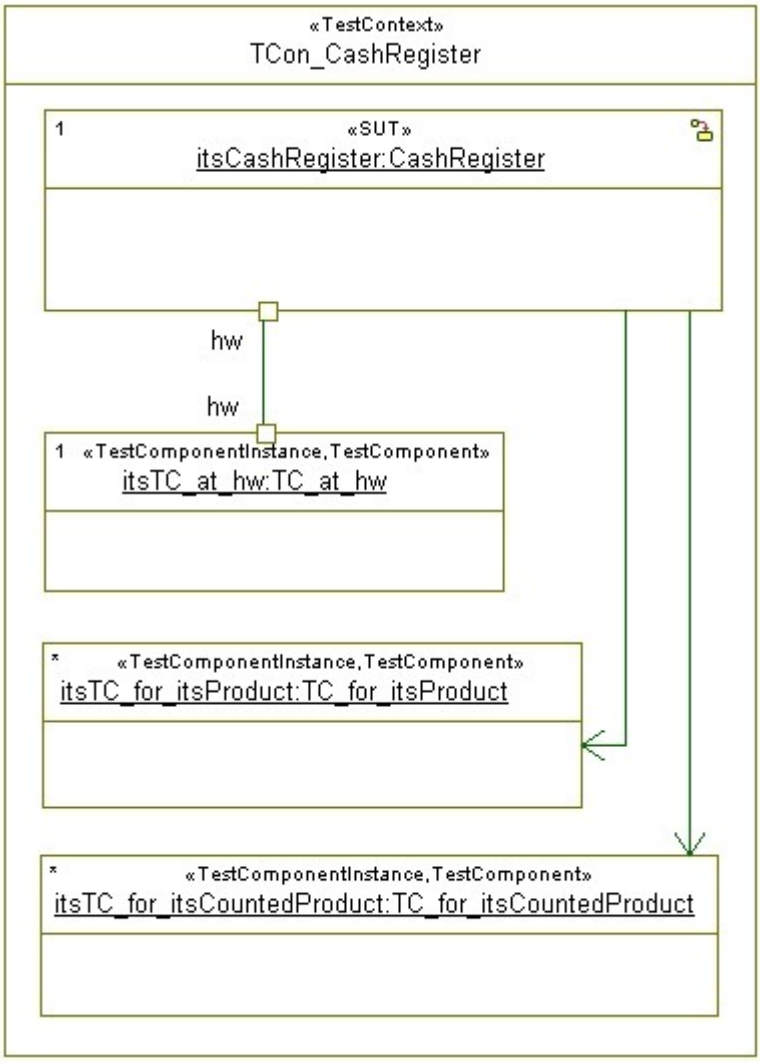
Stereotype *CppUnitContext* can be applied to a class and sets some properties for CppUnit testing integration. You can change a test context to CppUnitContext – and vice versa - by right-clicking a test context and secting “Change to > CppUnitContext”.

Stereotype *CppUnitConfig* can be applied to a configuration and provides a set of tags for customization of the CppUnit testing integration with Rhapsody.

Stereotype *CUnitContext* can be applied to a class and sets some properties for CUnit testing integration. You can change a test context to CUnitContext – and vice versa - by right-clicking a test context and secting “Change to > CUnitContext”.

Stereotype *CUnitConfig* can be applied to a configuration and provides a set of tags for customization of the CUnit testing integration with Rhapsody.

Subpackage *diagrams*: A *test context diagram (TestContextDiagram)* is a structure diagram that contains the SUT instances, the test component instances and their interconnections. It is used to define the structure of the test context graphically.



The test context diagram is being generated during the test architecture generation inside the test context. It is a structure diagram stereotyped with *TestContextDiagram*.

Stereotype *instantiated* is used to label associations that are always instantiated with a valid link during runtime. TestConductor interprets associations labelled with this stereotype like links. `<<instantiated>>` asscociations 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.

Stereotype *Arbiter* is used by TestConductor for auto generated test components that control the execution of a SD test case.

Stereotype *ArbiterInstance* is used by TestConductor for test component instances that are instances of Arbiter test components.

Stereotype *ControlArbiter* is used by TestConductor to mark a dependency of a SD test case on a Arbiter test component that controls the SD test case.

Stereotype *NoConsoleApp* can be applied to configurations in order to suppress opening a console when running the application.

Stereotype *ParameterTable* is used to mark a controlled file as a parameter table definition that contains values for all external test parameters of a test context.

Stereotype *replacement* is used to mark a dependency of a test component on the original class that is replaced by the test component in the test architecture.

Stereotype *instancereplacement* is used to mark a dependency of a test component object (implicit object) on the implicit object that is replaced by the test component object in the test architecture.

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 test architecture (Rhapsody in C).

Stereotype *greyboxreplacement* is used to mark a dependency of a <<TestSUT>> on the original class that is replaced by the <<TestSUT>> in the test architecture (for Grey Box Testing).

Stereotype greybox*instancereplacement* 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 test architecture.

Stereotype *scheduled* is used to mark a dependency of a test context on a *Scheduler* test component that controls the starting and stopping of test cases of the test context.

Stereotype *Scheduler* is used to mark an auto generated test component that is used to control the activation and termination of test cases.

Stereotype *SCTCInstance* is used to mark a test component instance to be an instance of a statechart test case test component.

Stereotype *stubbed* is used to mark an operation of a test component to be stubbed, i.e., that the behavior o fthe operation has been changed for testing purposes.

New term *TestActor* is used for test components that have the role of an actor in the test architecture. Test actors replace actors for testing purposes.

New term *TestFile* is used for test files in the test architecture. Test files replace files of the design for testing purposes.

New term *TestComponentInstance* is used to specify instances of test components.

New term *TestComponentObject* is used to stereotype copies of implicit objects in the role of test components.

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.

New term *TestPackage* represents a package that contains testing related model elements, e.g. other test packages, test contexts or test cases. 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.

Stereotype *TestParameter* is used to mark an attribute of a test context to be a parameter that can be controlled by a testing configuration by using a parameter table.

Stereotype *use_ParameterTable* is used to mark a dependency of a testing configuration on a parameter tanle in order to specify that the testing configuration shall apply the linked parameter table for the test parameters of the test context for which the testing configuration generates code for.

Stereotype *use_replacement* is used to mark a dependency of a test component instance on a test component that is a replacement of a design class for testing purposes.

Stereotype *use_instancereplacement* is used to mark a dependency of the test context on a TestComponentObject (i.e. a greybox replacement of an implicit object used in the role of a test component).

Stereotype *use_filereplacement* is used to mark a dependency of a test context on a test file indicating that this test file is used by the test context for testing purposes.

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.

Stereotype *use_greyboxinstancereplacement* is used to mark a dependency of the test context on a TestSUTObject (i.e. a greybox replacement of an implicit SUT object).

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).

New term Test*SUTObject* is used to mark a replacement object (basically a copy of the original implicit SUT object – used only for Grey Box Testing).

Stereotype *TestLink* is a stereotype on links an connectors (SysML). `<<TestLink>>` sets a code generation property that forces generation of link initialization code for link, regardless of its location in the design/test architecture 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 test architecture although the link refers to instances anywhere in the browser hierarchy.

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>>`.

TestBehavior Package

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

- *CodeCoverageResult*
- *CoverageResult*
- *ModelCoverageResult*
- *DefaultOperation*
- *DefaultTriggeredOperation*
- *DriverOperation*
- *RTC_InstInfo*
- *RTC_MsgInfo*
- *RTC_OperatorInfo*
- *RTC_RefInfo*
- *SDInstance*

- *StatechartTestCase*
- *StubbedOperation*
- *StubOperation*
- *TestAction*
- *TestAssignment*
- *TestCondition*
- *TestResult*
- *TestScenario*
- *Unrealized*
- *WitnessScenario*

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 moode.

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 testing configuration.

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.

A *default operation (DefaultOperation)* defines the default behavior of an operation of a test component. A test case in which the behavior of this operation is not explicitly specified uses this default behavior in the current test case execution.

A *driver operation (DriverOperation)* is an operation of a test component which is able to inject input stimuli to the SUT objects. It is generated automatically by TestConductor for the test component class that calls a message of a SUT object defined in a sequence diagram. During execution of the test case, TestConductor calls the driver operation, and as a result the test component stimulates the SUT as it is described in the used sequence diagram.

The stereotype *RTC_InstInfo* contains two tags *RTC_IgnoreSCBehavior* and *RTC_Monitor*. When adding this stereotype to an instance line of a test scenario, 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 state chart behavior of the tagged instance. If the tag RTC_Monitor is set, TestConductor just monitors all messages starting from the tagged instance.

The stereotype *RTC_MsgInfo* contains tags *RTC_Monitor*, *RTC_Receiver, etc*. When adding this stereotype to a message in a test scenario, the user can set these tags. If the tag RTC_Monitor is set, the tagged message is just monitored by TestConductor. If the tag RTC_Receiver is set, the tagged value is used as the real receiver instance of the tagged message. If the tag *RTC_DriverCallCode* is set, TestConductor generates the string contained in this tag instead of the standard call code TestConductor generates for driver operations. If the tag *RTC_InitCode* is set, TestConductor generates the string contained in this tag instead of the standard init code TestConductor generates for driver operations. If the tag *RTC_MsgId* is set, the specified string is used to reference the message in macros *RTC_ASSERT_SD_NAME*. If the tag *RTC_StubBodyCode* is used, TestConductor generates the string contained in this tag instead of the standard stub code TestConductor

generates for stub operations. For further information please read the chapter User Defined Driving Operation Calls at page *208*.

The stereotype *RTC_RefInfo* is used internally for unique identification of messages in sequence diagrams which are referenced by other sequence diagrams.

A *sequence diagram instance (SDInstance)* represents one instance of a test scenario. When using a sequence diagram for testing purposes, several parameters must be defined that influence the behavior of a test case. A combination of a sequence diagram with such a set of parameters forms a sequence diagram instance.

Stereotype *StatechartTestCase* is used to stereotype the dependency of a statechart test case on the test component owning the statechart defining the test.

A *stubbed operation (StubbedOperation)* is an operation for which at least one test case 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 test case if either the default behavior should be executed or a specific stub-operation.

A *stub operation (StubOperation)* is a replacement of an operation of a test component class. It realizes the code for an operation call return value specified in the referenced sequence diagram. The code of the stub operation is generated automatically by TestConductor.

A *test action (TestAction)* is an action block that can be placed on life lines in TestScenarios. There are different kinds of test actions: <InitAction>, <PreCallAction>, <CallAction>, <PostCallAction>, <StubAction>. Inside these actions, one can place e.g. assertions to perform complex checks on output values (return or out arguments), or one can write code that initializes complex input data.
These kinds of TestActions correspond to the tags of RTC_MsgInfo

- <InitAction> - RTC_DriverInitCode
- <PreCallAction> - RTC_DriverInitCodeAdditional
- <CallAction> - RTC_DriverCallCode
- <PostCallAction> - RTC_DriverCallCodeAdditional
- <StubAction> - RTC_StubBodyCode

Note, that both specification techniques are mutual exclusive. If such TestActions are used in order to determine the code propulated for the respective message, the RTC_MsgInfo tags are ignored for this message.

A *test result (TestResult)* represents an outcome of an execution of a test case. It is a textual report that contains detailed information about the test case execution, e.g. if the test case has passed or failed.

The stereotype *TestScenario (test scenario)* contains two tags *RTC_ActivationCondition* and *RTC_SDParameters*. When adding this stereotype to a test scenario, 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.

Messages with stereotype *Unrealized* are filtered out and ignored during test execution. See also section Ignoring Unrealized Messages.

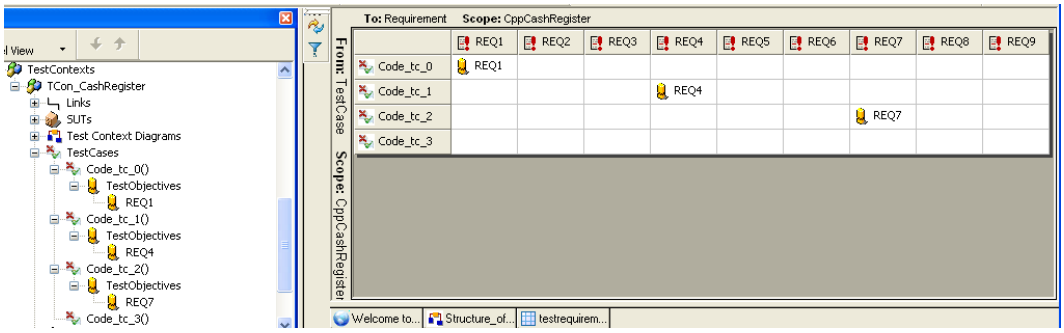
TestDocumentation Package

The *TestDocumentation pack*age contains a Matrix-Layout *TestRequirementCoverage* and a Table-Layout *TestResultTable* in order to present test information in matrix and table notation.

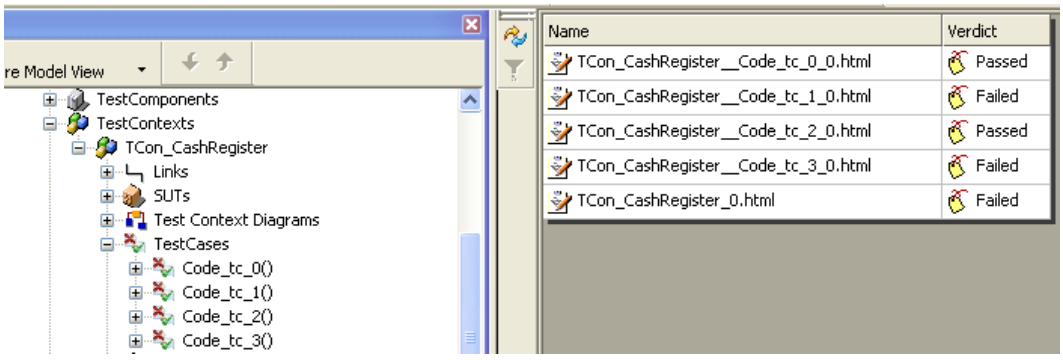
The layouts are used to define two stereotypes:

- *TestRequirementMatrix*
- *TestResultTable*

A *TestRequirementMatrix* shows in an array view if and how requirements are tested by test cases. The left hand side of the array shows all existing test cases. The upper side shows all the requirements. The cells contain an entry if a TestObjective from the test case to the requirement exists in the model, for instance from test case *Code_tc_0* to requirement *REQ1*.



A *TestResultTable* shows in a table form the existing test cases and their current result values. The left column of the table shows all existing test cases. The right column shows the current test case results, for instance verdict *Passed* for test case *Code_tc_0*.



Automatic Test Generation (ATG) Package

The *ATG package* consists of several stereotypes which are enhancements to the UML Testing Profile. For more information about the ATG package and its stereotypes please refer also the *Rhapsody Automatic Test Generation (ATG) User Guide*.

Using the Testing Profile

The Rhapsody Testing Profile is automatically utilized by Rhapsody TestConductor. The functionality of the tool set are explained in the subsequent chapters of this user guide.

Refining Testing Profile Stereotypes

Most model elements in a test architecture created by TestConductor are marked with stereotypes defined in the Testing Profile. Stereotypes are used for three functions: 1.) To arrange special elements in the same group in the model browser ('new term' stereotypes); 2.) As a hook for TestConductor actions (TestConductor actions are only available on certain elements); 3.) Stereotypes add or modify certain properties/tags of elements of the test architecture.

For example test cases in a test architecture are basically operations provided with the new term stereotype <<TestCase>>, which sets some property values and leads to grouping all test cases 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 common operations. As another example the stereotype <<TestingConfiguration>> is used to distinguish standard configurations from testing configurations which are adjusted to the special needs of the TestConductor test architecture. A <<TestingConfiguration>> has additional tags for configuring additional features (like coverage measurement) or fine-tuning the test execution (e.g. rtc_log_kind to define the manner of logging).

Users may wish to create their own stereotypes to have a simple and transparent way to induce specific changes to elements in reoccurring scenarios. But if settings or tags are to be modified which are also affected by a coexisting Testing Profile stereotype on the same element -meaning that two stereotypes are trying to modify the same property in the same way- it is not sure which stereotype's modification is actually applied on the element, therefore it is not recommended to have conflicting stereotypes. The option to replace the Testing Profile stereotype with the user stereotype is not advised either, since the Testing Profile stereotypes act as hooks for TestConductor actions, thus disabling TestConductor functionality on that element. The solution is to have the user stereotype inheriting from the Testing Profile stereotype, thus preventing conflicts and preserving TestConductor functionality on that element[1].

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

[1]Note that changing default values of TestConductor stereotypes may affect the functionality of the test architecture.

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 analyses 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 test architecture (or test harness or test frame) must be constructed
2. Test cases must be defined and implemented
3. Test cases must be executed on the host machine
4. Test cases must be executed on the target machine

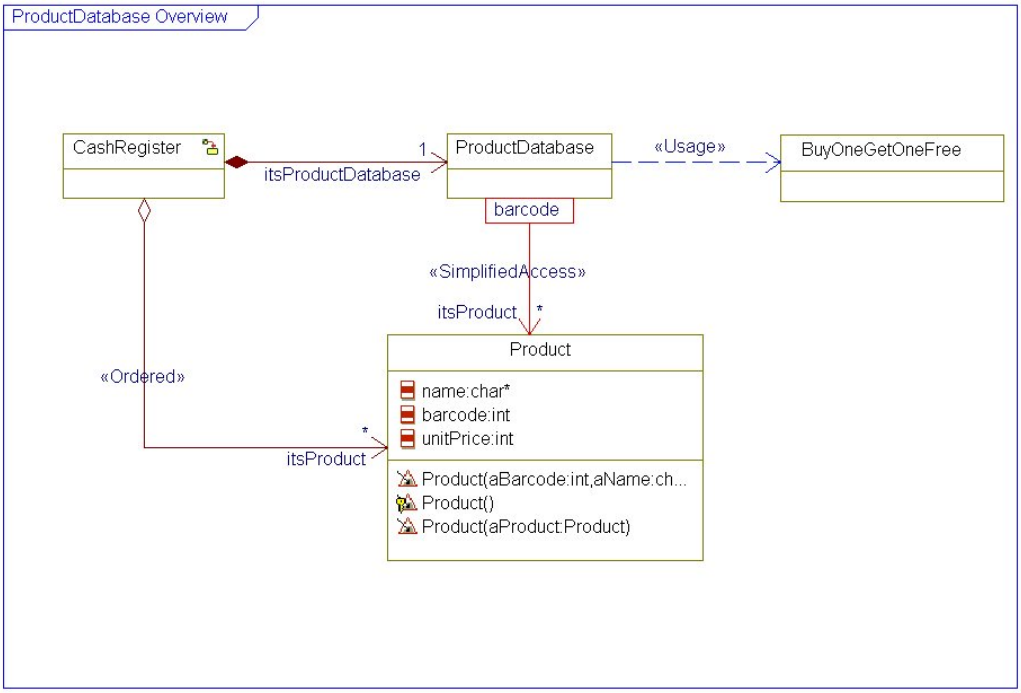
Each of the four mentioned steps is usually time consuming and difficult to perform. TestConductor makes the preparation, execution, and the assessment of tests much easier by lifting the test process up to the level of UML models, and by offering a high degree of automation for the steps listed above.

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 test architecture generation can be used for:

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

The other complex task of unit testing is the definition of test case or test scenarios, 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 test case definition via sequence diagrams or flow charts with the familiar pure code based test cases. Using the optional add-on Rhapsody Automatic Test Generation (ATG), you have also the possibility to perform automatic test case generation.

The next chapters use the CashRegister model known from the *Rhapsody "Essential" Tool Training*. The unit test will be done on the CashRegister class.



Automatic Test Architecture Generation

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

- Creation of a new test package
- Creation of a new test context
- Instantiation of the selected SUT class as part of the test context
- Creation of test components
- Instantiation and 'wiring' of test component instances as parts of the test context
- Creation of an adequate code generation configuration
- Adding a test configuration (dependency-relation) to the test context referring to the created code generation configuration
- Creation and drawing of a test context diagram

Fundamentally, TestConductor supports two different testing modes: Animation based and assertion based testing mode. Test architecture creation will create different resulting test architectures depending on the chosen testing mode:

- animation based testing mode (applicable to C, C++, Java, Ada models): In animation based testing mode, the scheduling and arbitration, i.e., the way TestConductor decides whether a test case is passed or failed, is based on animation messages coming from Rhapsody's animation feature.
  In particular comparison of message observations to the expectations according to the test specification relies on serialization underlying the animation feature.
  Test execution is based upon running an appropriate test specific observer in the

Rhapsody process communicating with the tested application via the Rhapsody animation socket. Hence, animation based testing mode always requires:

  - animation instrumentation (including requirement of appropriate serialization for types, objects, classes, functions, events, e.t.c)
  - socket connection between tested application and Rhapsody application.

- assertion based testing mode (applicable to C and C++ models only, not available for Java and Ada): In contrast to animation based testing mode, in assertion based testing mode both scheduling and arbitration of test cases is directly controlled by assertions that are compiled into the test executable, i.e., scheduling and arbitration of test cases is independent from Rhapsody's animation feature. Since in assertion based testing mode the test cases 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 focusses on animation based testing mode.

For animation based test architectures, test architecture creation will always introduce test components that inherit from original design classes – if possible. Since stubbing of operation calls is only possible in test components, stubbing is restricted by the restrictions for inheritance imposed by the particular modeling language. E.g. Rhapsody in C models only allow inheritance from interfaces – thus, regular classes can't be stubbed by inheriting test components. For Rhapsody in C++ models only virtual operations can be stubbed, non-virtual operations can't be stubbed using inheriting test components.

A table with the main differences between assertion based and animation based testing mode can be found here: Differences between animation and assertion based testing mode on page 102.

TestArchitecture generation can be customized interactively using property `TestConductor::Settings::CreateTestArchitectureMode` (cf TestConductor settings "General Properties", page 132).

If CreateTestArchitectureMode is set to 'Standard', then project properties are used in the generated code generation configuration while 'Advanced' opens a dialog that allows selection of an existing configuration from which all overridden properties. settings, and scope settings will be inherited.

It may sometimes be necessary to manually adjust the scope of the CG Component after automatic test architecture creation. In rare cases, all classes of one package may have been replaced by replacements, but types or events of that package still need to be regarded in the scope. In this case, it might be helpful to select a package with right-click instead of left-click. While left-clicking a package in the scope dialog selects the package and its contents, right-click selects only the package and its non-selectable content.
Note that TestConductor can't determine meaningful parameters for non-standard constructors automatically for instances of test components or classes having no default constructor. It might be necessary to manually adjust the constructor calls for test component instances or for the SUT after test architecture creation w.r.t. constructor arguments.

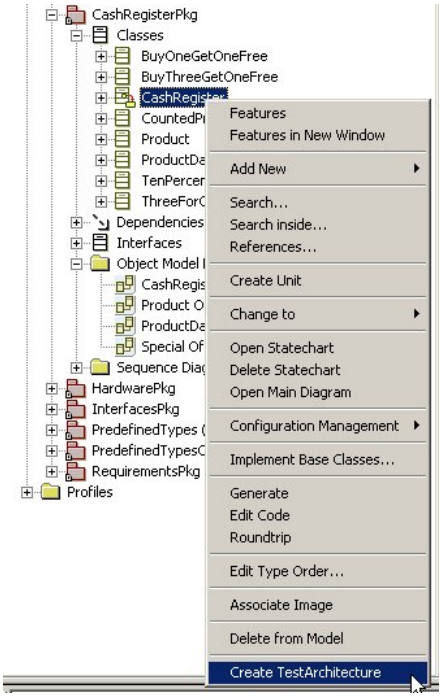
Using Classes

For the next steps do the following:

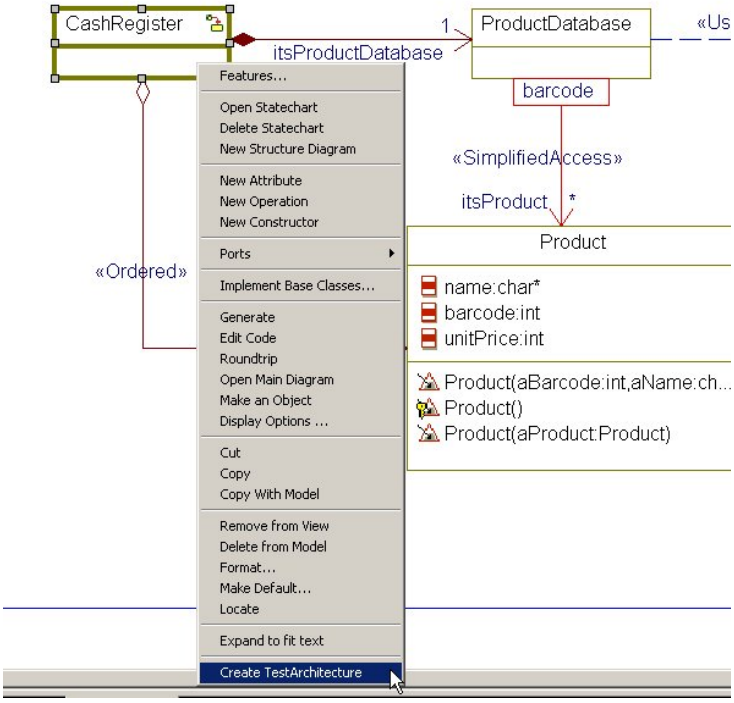
- Open the CashRegister Model from the '`\Samples\CppSamples\TestConductor`' folder.
- Browse to the object model diagram folder in the package `CashRegisterPkg`
- Open the object model diagram `ProductDatabase` Overview

There are two alternative ways to invoke *creation of a test architecture* for the class `CashRegister`:

- Right-click on the `CashRegister` class in the Rhapsody browser and select **Create TestArchitecture**

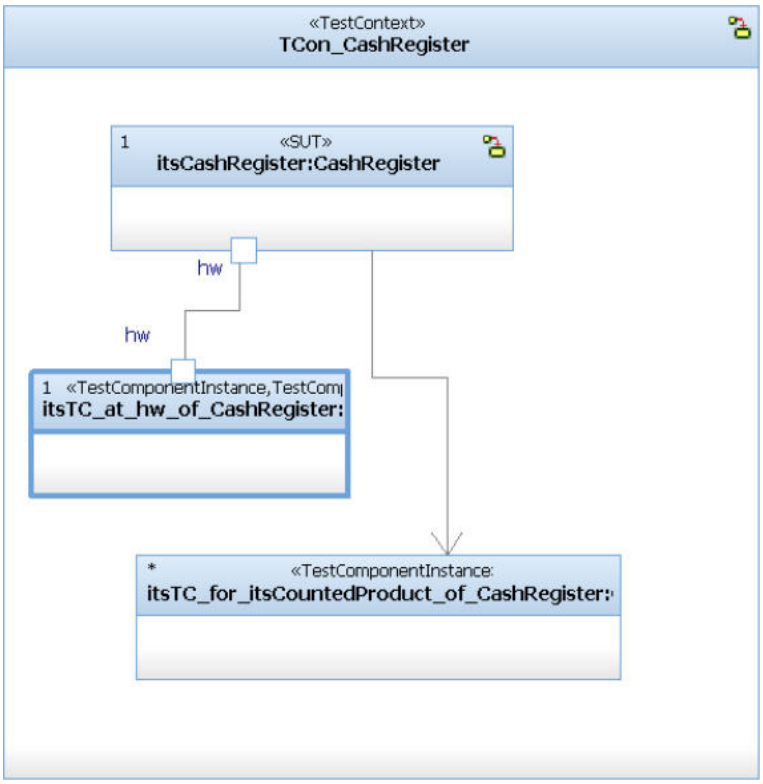


- Right-click on the `CashRegister` class in the object model diagram and select **Create TestArchitecture**

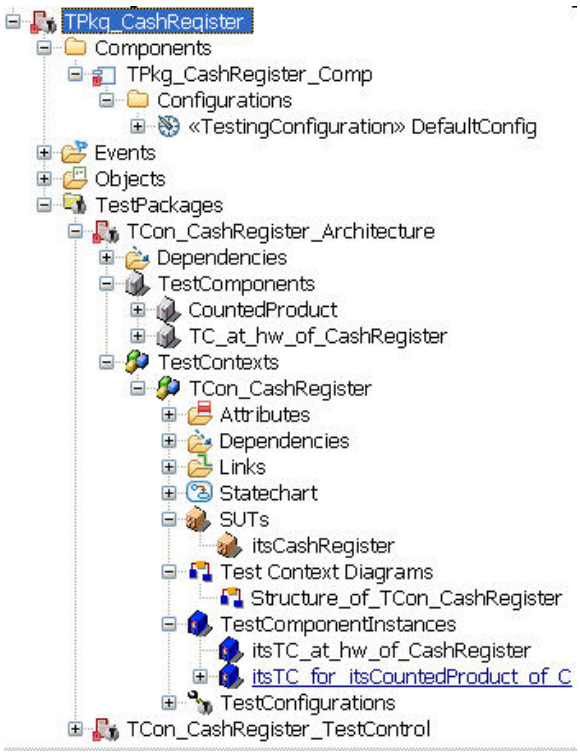


TestConductor automatically creates the complete necessary test architecture which consists of:

- A new *test context diagram* with the test context `TCon_CashRegister` containing the CashRegister object `itsCashRegister` itself as SUT and all necessary test component instances which are derived from the SUT associations and ports.



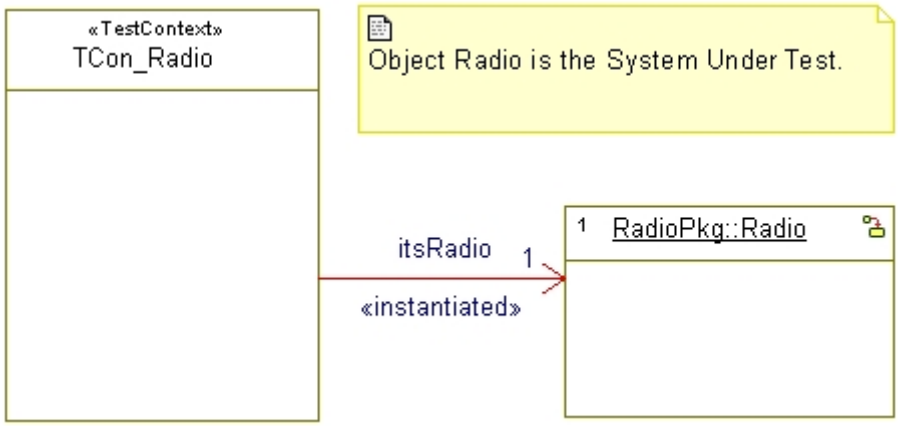
- A new test package `TPkg_CashRegister` which contains all generated test components, the test context `TCon_CashRegister` with the SUT `itsCashRegister`, the test context diagram and the test component instances



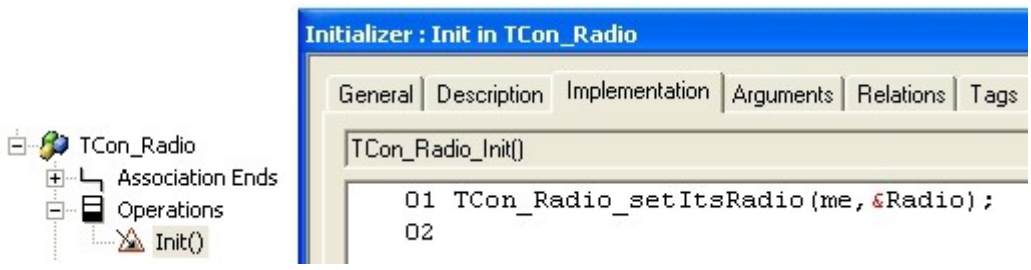
Using Objects

Creating a test architecture on objects is a similar workflow as shown for classes, but in order to create a test architecture for testing an object, the object can not be directly instantiated as part of a test context. If an object was instantiated as part of a test context, 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 test contexts 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 test architectures for objects in animation based testing mode.

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



Except for test architectures created with global object support (to enable global object support for C or C++ models, check property `TestConductor::Settings::CreateTestArchitectureUsingGlobalObjects`) `<<instantiated>>` associations are not initialized by links but the test context 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 The implementation of the constructor/initializer is currently limited to Rhapsody in C/C++.



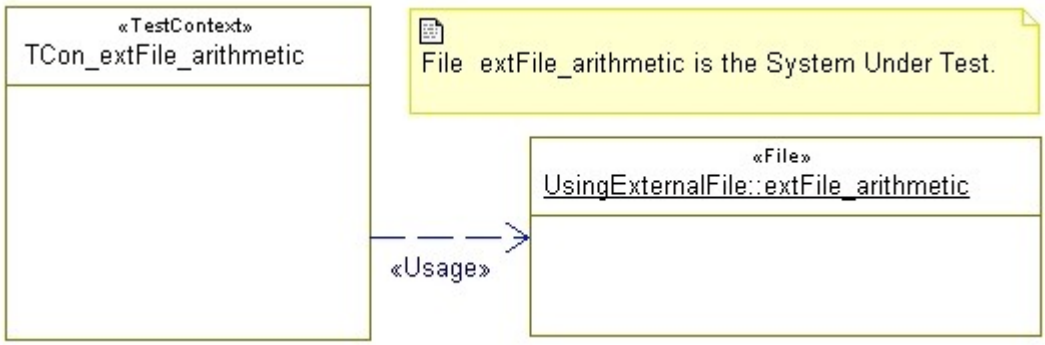
Except for test architectures created with global object support, the test architecture for objects will not care about ports of the object, since the mapping of these ports to ports of other objects may already be defined in the design. The only way to stimulate an object in a system test architecture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

Rhapsody offers an alternative to create a test architecture 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**.

Note: For Rhapsody in Ada, the user has to set the <<instantiated>> association manually. This is due to the fact, that global objects are instantiated after instantiation of the initial instances specified in the Initialization tab of the code generation configuration's feature dialog. In order to set the associaton manually, the initialization code entry of the Initialization tab of the code generation configuration's feature dialog is used, e.g.:
Tpkg_object_0.TCon_object_0.set_itsObject_0(
p_TCon_object_0.all, Default.RiA_Instances.object_0);
if object_0 is an object of object_0_Class.

Using Files (Modules)

Creating a test architecture on files(to be more precise: modules) is a similar workflow 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 test context referring the module using a `<<Usage>>` dependency.



The declaration of external (source and library) files and testing with TestConductor is discussed in the chapter Black-Box Testing of External Files and Libraries at page *216*.

Using Parts

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

TestArchitectures with multiple SUT classes or objects

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.

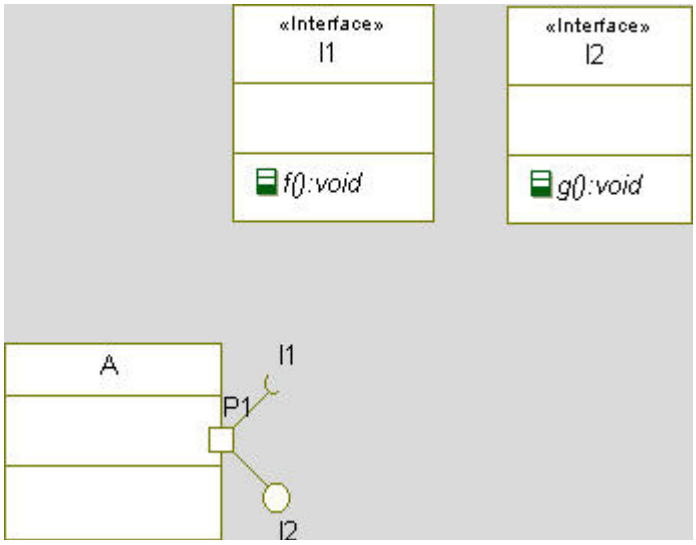
For black box TestArchitectures an iterative approach of TestArchitecture creation, removal of TestComponents, addition of further SUT elements, appropriate connection of SUT elements and TestArchitecture updates can easily performed using Rhapsody modeling capabilities and the context menu helpers in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

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

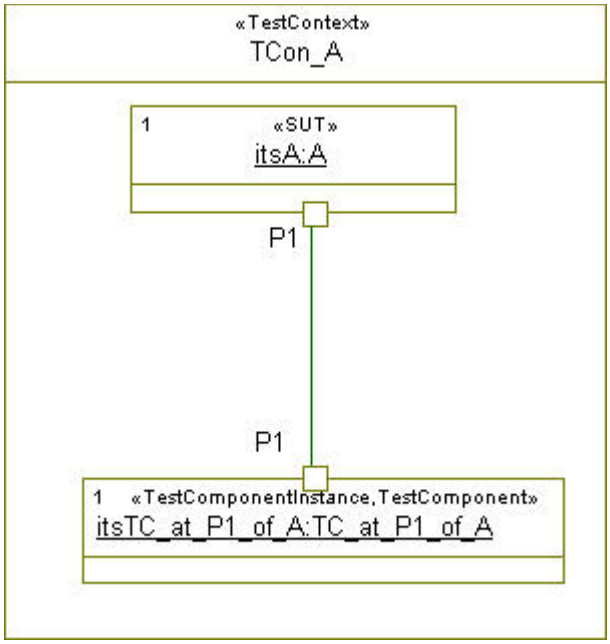
Updating TestArchitectures

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' 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.

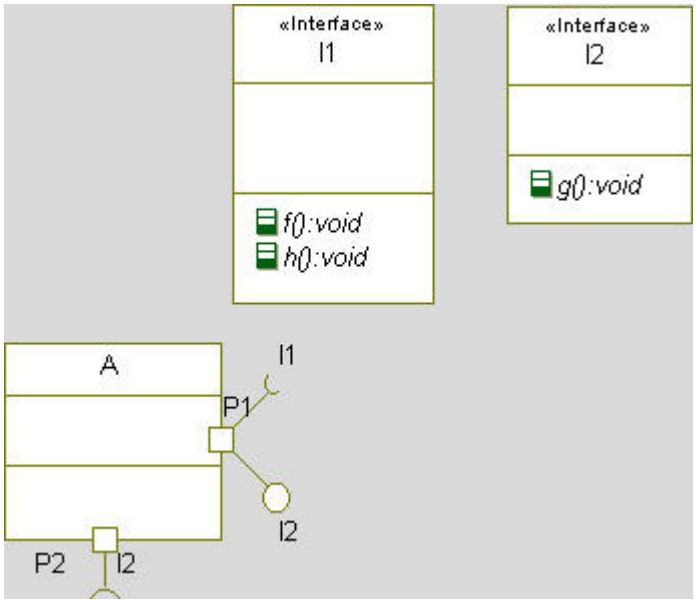
The following example illustrates TestArchitecture update:



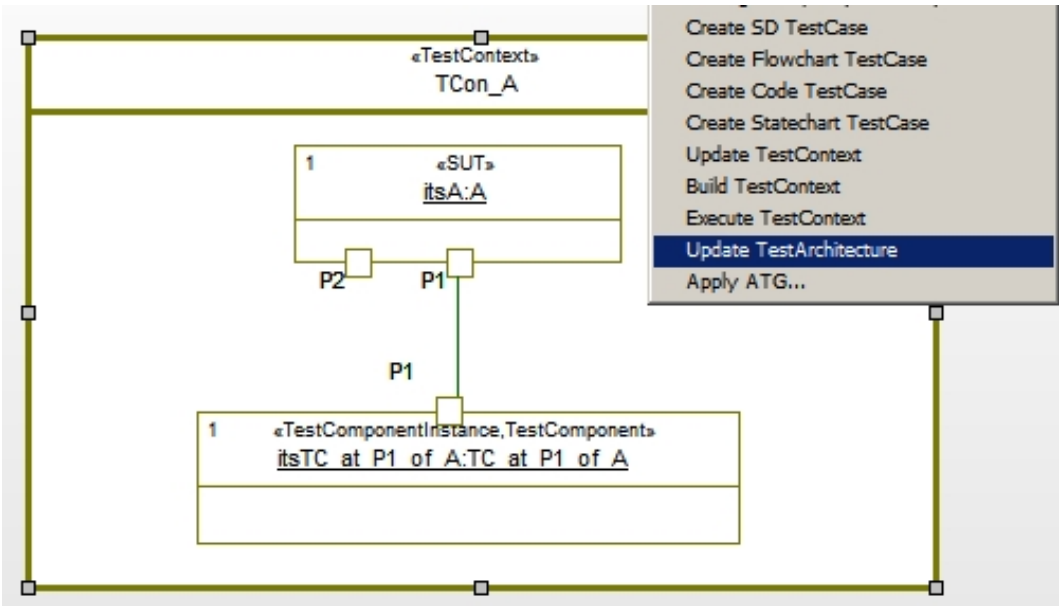
There is a class A that contains a P1 with a required Interface I1 and a provided Interface I2. The interface I1 specifies one operation f() that takes no arguments and has no return type, and interface I2 specifies an operation g() also without arguments and return type. When selecting class A as the SUT, TestConductor creates the following TestArchitecture for it:



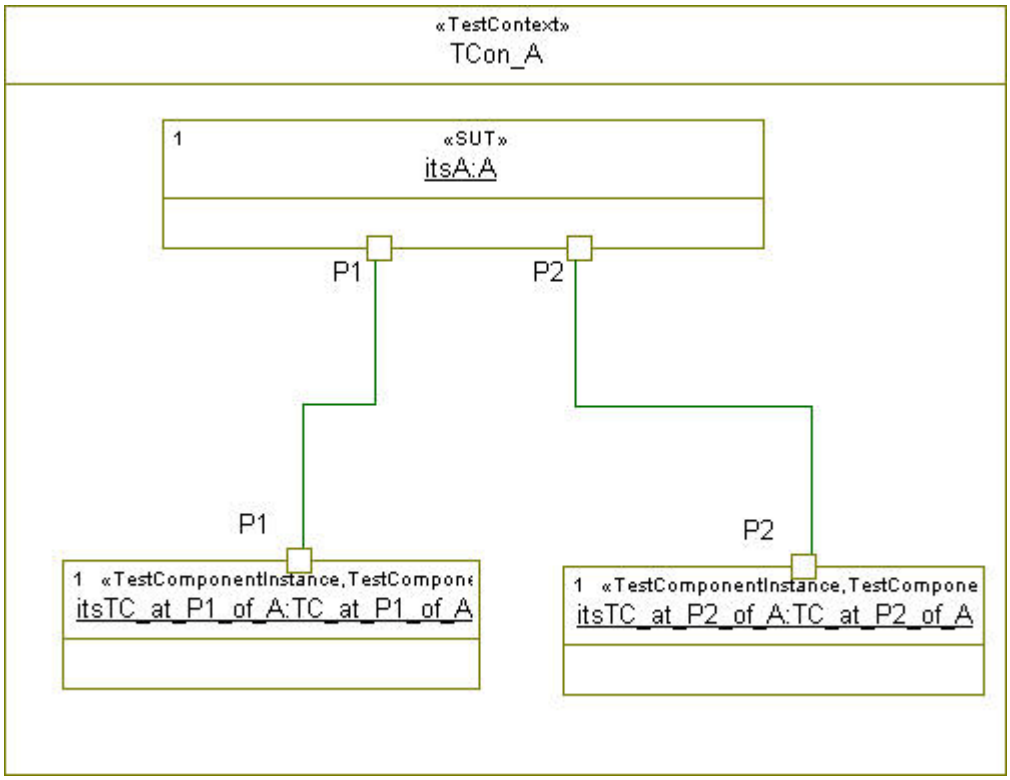
In the generated TestArchitecture, one TestComponent is created containg an appropriate port P1 such that the instance of the TestComponent can be linked to the Port P1 of the SUT instance itsA. Now suppose you do some changes on the SUT class A. For instance, we can add an additional Port P2 with a required Interface I2 to A, and we add a new operation h to the Interface I1:



Because of these design changes, the previously generated TestArchitecture is not complete any more, In order to get again a complete TestArchitecture TestConductor provides the capability to update an existing TestArchitecture. To do this, select the TestContext that should be updated and select “Update TestArchitecture”:

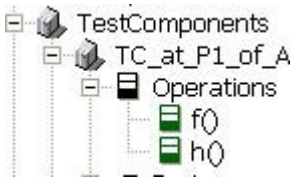


After applying “Update TestArchitecture”, you get the following updated TestArchitecture:



To update the TestArchitecture accordingly, TestConductor did the following modifications to the existing TestArchitecture:

1. A second TestComponent is created that is connected to the new Port P2 of the SUT instance.
2. Since an additional operation was specified for Interface I2, an additional operation h is added to the TestComponent connected to port P1.



After these modifications have been made by TestConductor, the TestArchitecture is complete again.

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.

TestArchitecures for MicroC Models

TestConductor supports testing of MicroC models with a specifically taylored 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 test context.

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. accessability of associations. In particular, this affects the generated initializers of TestContexts for objects (cf. TestArchitecture creation “Using Objects”, page 38). 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 54 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.

TestArchitecures for Code centric Models

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

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

The TestConductor sample “CppCarRadio” demonstrates testing of a code centric model. For the next steps, please open the sample located in folder “Samples\CppSamples\TestConductor\CppCarRadio”, right click class “Radio” and select “Create TestArchitecture”. A dialog appears with the options to select an existing library CG Configuration or to create a new library CG Component and Configuration for the SUT. If the existing CG Configuration “RadioLib::RadioDebug” is selected, a TestArchitecture is created with another CG Component and Configuration for the generation and compilation of the test harness. This CG Configuration has some properties enabled which are usually disabled in the code centric profile, for example properties “CG::Relation::AddGenerate” and “CG::Relation::SetGenerate” are enabled and “CG::Configuration::MainGenerationScheme” is set to “Full”. The scope of the newly created CG Component contains only the test harness and it has a “Usage” dependency to the CG Component of the SUT, making sure the needed header files and the library of the SUT can be found.
If the user selects to create a new CG Component for the SUT library, then TestConductor creates two CG Components in the TestArchitecture: First a library CG Component “libSUT” with the scope set to the SUT class and its associations and the default property settings of the project and second an executable CG Component for the test harness.

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

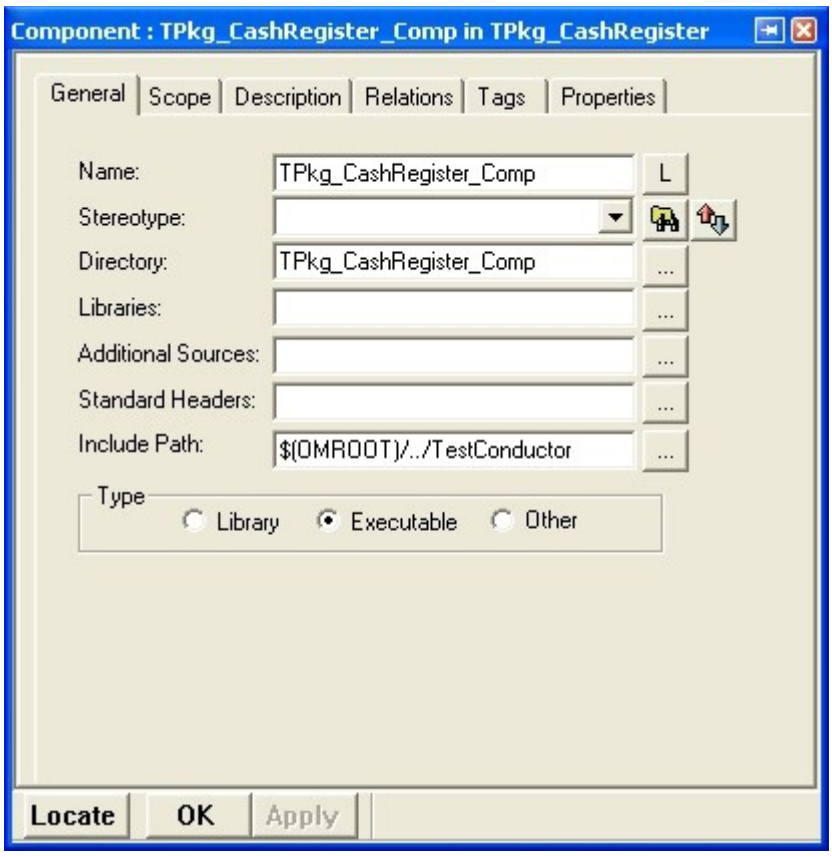
The TestArchitecture for code centric models can be used the same way as TestArchitectures for non code centric models, with some restrictions because of the not animated SUT (internal communication of the SUT cannot be observed).

Unit testing of AUTOSAR Software Components

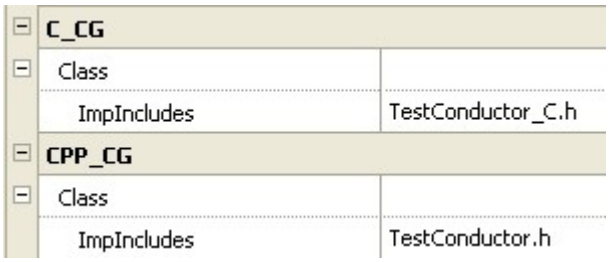
Testing of AUTOSAR Software Components is supported only for AssertionBased testing mode.

TestConductor.h, TestConductor_C.h and TestConductor_C.c, TestConductor.jar, TestConductor.ads and TestConductor.adb

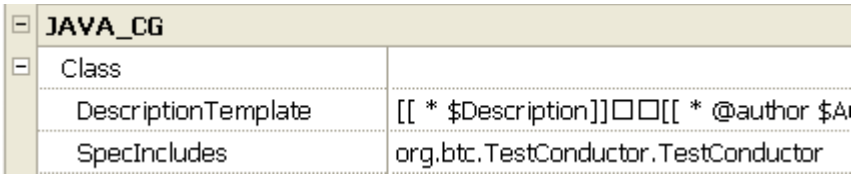
Since Rhapsody 7.1 the testing profile require the test context, test components, and test component instances to include the TestConductor header file by setting property `CPP_CG.Class.ImpInclude` to `TestingConductor.h`. Additionally, TestConductor adds the path '`$(OMROOT)/../TestConductor`' to the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



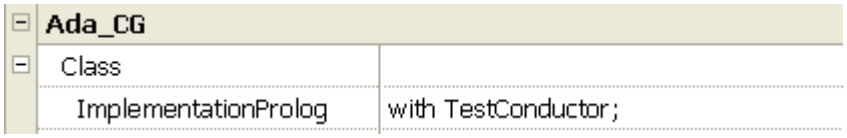
To provide an adequate assertion support for Rhapsody in C, a similar header file is provided and the testing profile was extended, such that test context, test components, and test component instances automatically include an appropriate `TestConductor_C.h` header by setting property `C_CG.Class.ImpInclude` to `TestConductor_C.h`. In contrast to the Rhapsody in C++ solution, for Rhapsody in C also an C-Implementation file was provided, which is linked only once.



For Java, the class “org.btc.TestConductor.TestConductor” is added as specification include for TestContext and TestComponents.



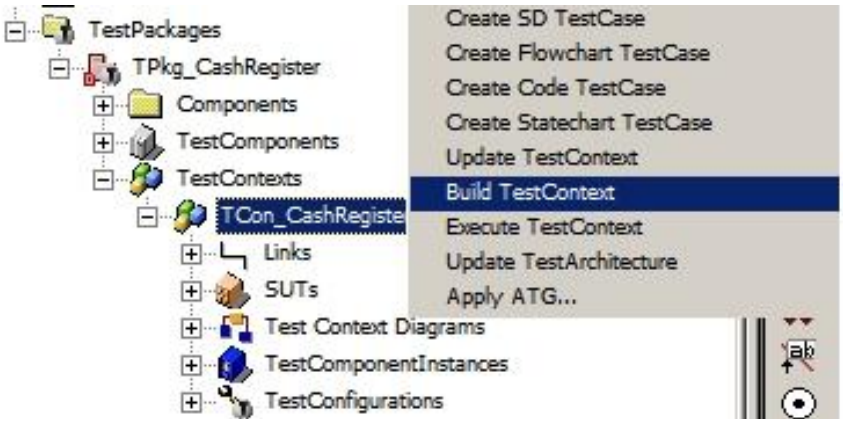
For Ada, the package “TestConductor” is made visible by adding an appropriate “with” clause to the implementation of test contexts and test components.



Generate and Build the Test Context

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

- Right-click on the test context `TCon_CashRegister` 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.

Test Case Definition

Now *test cases* for the generated test context can be defined. TestConductor provides four possible means to define test cases:

- Test case definition with pure code
- Test case definition via flow charts (only in C/C++)
- Test case definition via statecharts
- Test case definition via sequence diagrams

Test Case Definition with Code

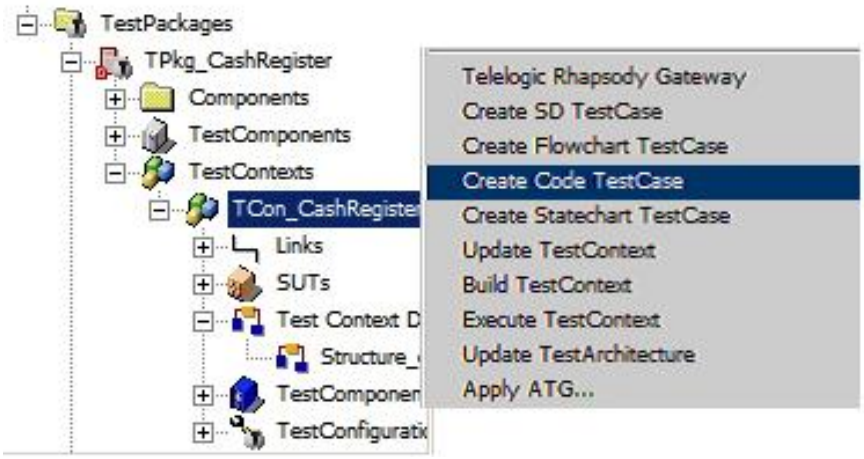
One of the most used means to test units today is writing test cases in the same language than the application is written. In the C/C++/Java/Ada domain, often the complete test environment and also the test cases are written in C/C++/Java/Ada with the goal of functional or coverage testing.

With Rhapsody and TestConductor it is also possible to write test cases manually, because test cases are stereotyped operations of a test context.

Define a Code Test Case

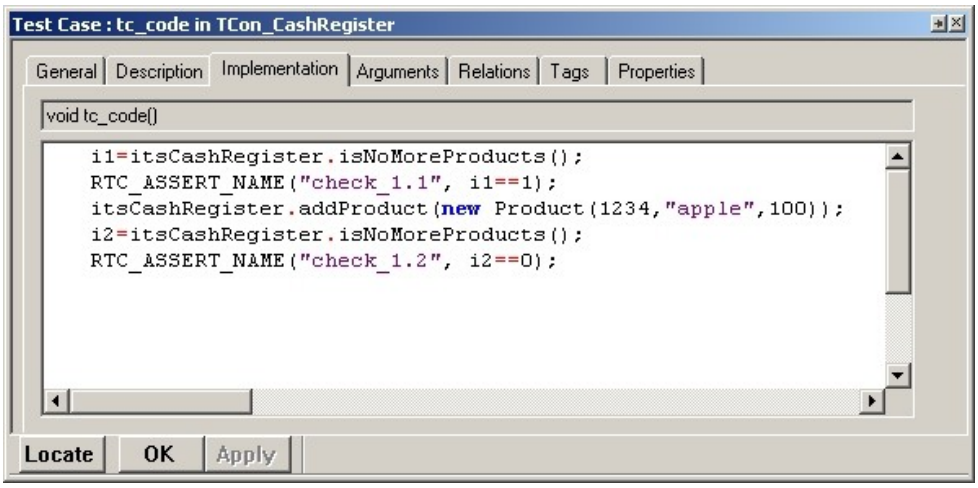
The creation of a new test case is nearly the same than creation of a new operation:

- Right-click on the test context `TCon_CashRegister` and select **Create Code TestCase**



- Name the new test case “tc_code”

- Open the **Features** dialog of the new test case and enter the code into the implementation tab.



The objective of the test case is to verify whether the function `addProduct` correctly adds a product to the bill list (realized by the ordered association `itsProduct`).

First, the test case checks whether the bill list is empty. If not, the operation `isNoMoreProduct` returns `FALSE`. In this case the macro `RTC_ASSERT_NAME ("check_1.1", i1=1)` returns a *FAILED* to TestConductor. Otherwise the result of the RTC_ASSERT_NAME macro is *PASSED*. In the next step a product "apple" is added. At the end the bill list is checked again.

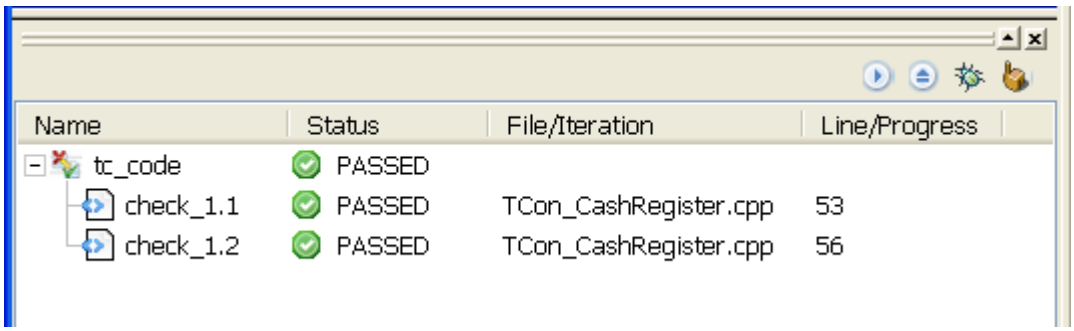
Note: This test case is using two attributes `i1` and `i2` of type `int`. Both attributes have to be defined within the test context `TCon_CashRegister`.

Note: TestConductor provides several RTC_ASSERT macro types, which can be used to define assertions within test cases. A detailed description of these macros can be found in the chapter TestConductor Assert Macro on page *251*.

Execute a Code Test Case

Now you are able to execute the test case by doing following steps:

- Right-click on the test case "tc_code" and select **Build TestCase** from the context menu
- Right-click on the test case "tc_code" and select **Execute TestCase** from the context menu

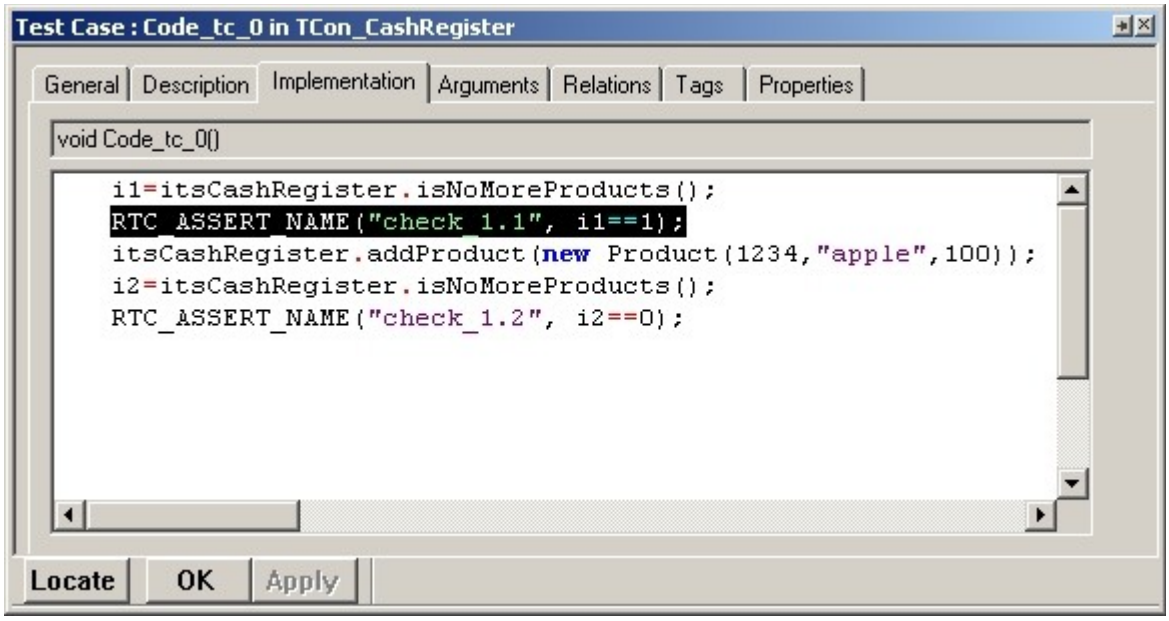


The test execution window shows the result of the checked assertions. Both are *PASSED* meaning that the tested behavior is ok.

Further information about test execution and the related results is described under chapter Test Execution on page *75*.

Failure Analysis in CodeTest Cases

TestConductor lists in the execution dialog all executed assertions. To display the corresponding assertion, select in the execution dialog the item name in the column **Name** and press the button **Show Assertion**.



Further information about failure analysis can be found in chapter Failure Analysis on page *224*.

Testing reactive behavior with Code Test Cases

Since code test cases are basically operations of a test context, testing reactive behavior, i.e. reaction to events, can not be done without modifications to the test context. Operations can't wait on events so please make the TextContext an active object and hence a separate thread. In this case, the thread executing the test context can be delayed unless the SUT has reacted to an event.

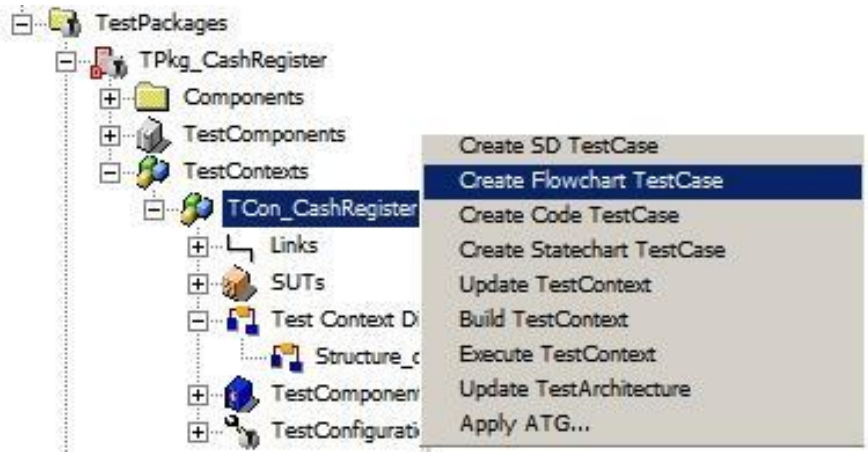
- Example code in C++:
```
itsClass_0.GEN(evX());
OXFTDelay(1000);
RTC_ASSERT_NAME("reaction",itsClass_0.IS_IN(reaction_state));
```
- Example code in C:
```
RiCGEN(&(me->itsClass_0),evX());
RiCOXFDelay(1000);
RiCIS_IN(&(me->itsClass_0),reaction_state);
```
- Example code in Java:
```
itsStopWatch.gen(new evPressKey(1));

try {

  wait(4000);

} catch(Exception e)

{ }

TestConductor.ASSERT_NAME("Check state of
stopwatch",itsStopWatch.isIn(ROOT.Running));
```

Test Case Definition with Flow Charts

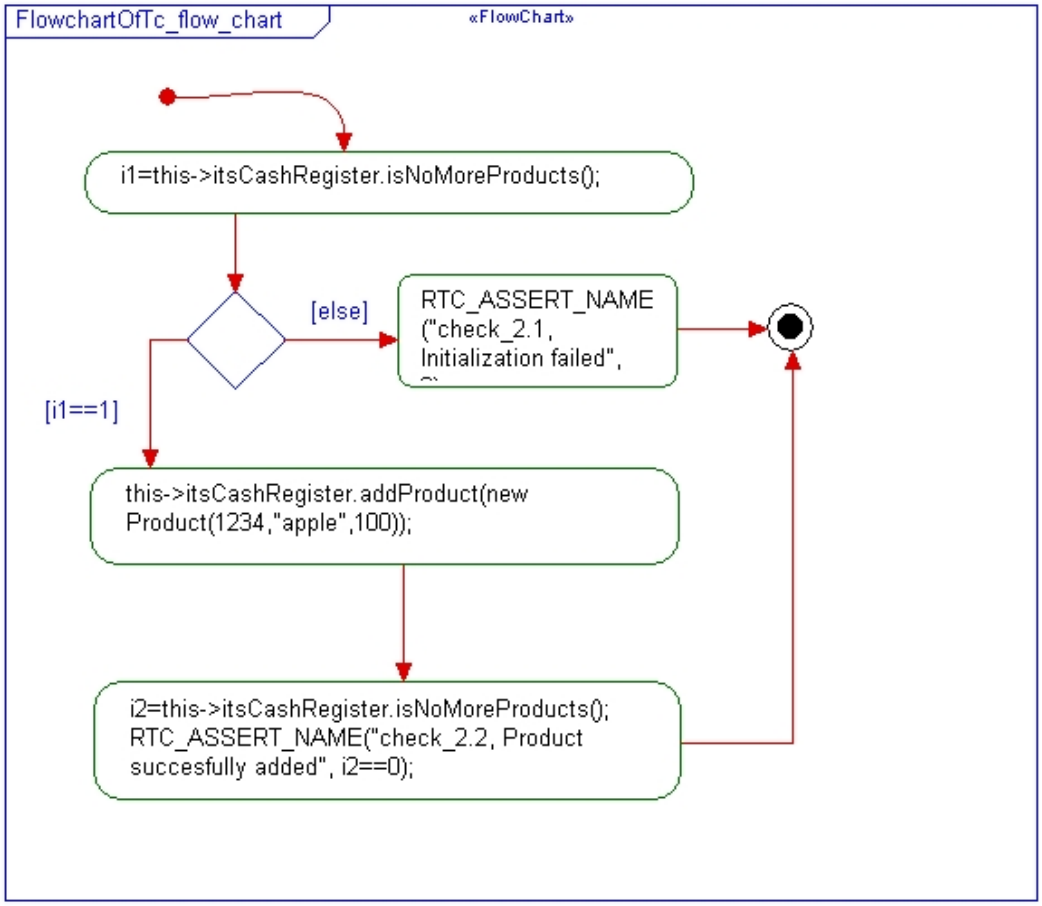
A graphical way to describe test cases is by using flow charts. Since test cases are special operations of a test context you can use flow charts. Flow charts can be used to define the behavior of operations with Rhapsody.
Defining test cases by flow charts is available for C++ and C only.

Define a Flow Chart Test Case

- Right-click on the test context `TCon_CashRegister` and select **Create FlowChart TestCase**



- Name the new test case “tc_flow_chart”
- Draw the following flow chart



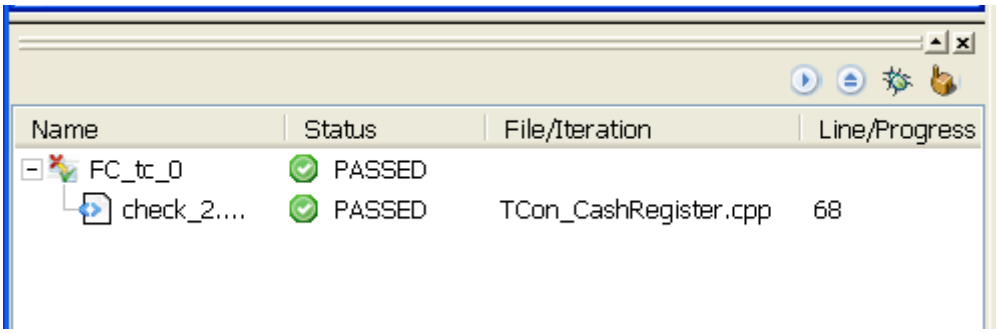
The objective of the test case is the same as used in the code test case above.

First, the test case checks whether the bill list is empty. If not, the operation `isNoMoreProduct` returns `FALSE`. In this case the macro `RTC_ASSERT_NAME ("check_2.1, Initialization failed", 0)` returns a *FAILED* to TestConductor. In the next step a product "apple" is added. At the end the bill list is checked again

Execute a Flow Chart Test Case

Now you are be able to execute the test case by doing following steps:

- Right-click on the test case "tc_flow_chart" and select **Build TestCase** from the context menu
- Right-click on the test case "tc_flow_chart" and select **Execute TestCase** from the context menu

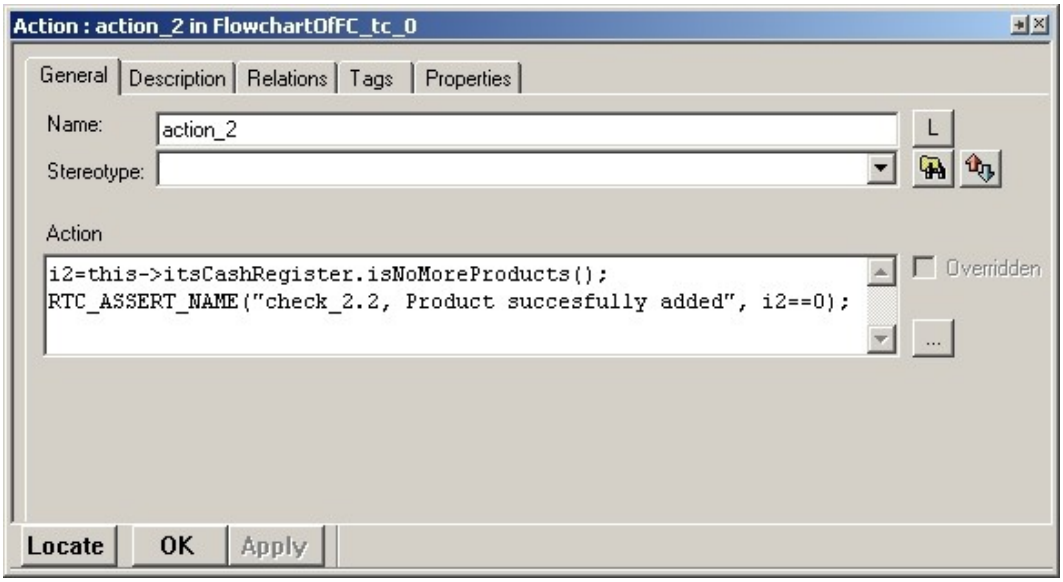


The test execution dialog shows the result of the defined assertions. The assertion “check_2.2, Product successfully added” passed the test, which means that the tested behavior is ok. Other than in the code test case here you can only see one assertion in the execution dialog. This is due to the condition connector used in the flow chart. Only when the condition `[i1==1]` is `false`, the assertion “check_2.1, Initialization failed” is executed.

Further information about test execution and the related results is described under chapter Test Execution on page *75*.

Failure Analysis in Flow Chart Test Cases

TestConductor lists in the execution dialog all executed assertions. To display the corresponding assertion, select in the execution dialog the item name in the column **Name** and press the button **Show Assertion**.



Further information about failure analysis can be found in chapter Failure Analysis on page *224*.

Testing reactive behavior with Flow Chart Test Cases

Since flow chart test cases are basically operations of a test context, testing reactive behavior, i.e. reaction to events, can not be done without modifications to the test context. Operations can themselves not wait on events. Thus, the test context has to be active, i.e. must run in a thread different form the thread executing the SUT. In this case, the thread executing the test context can be delayed unless the SUT has reacted to an event.

- Example code in C++:
```
itsClass_0.GEN(evX());
OXFTDelay(1000);
RTC_ASSERT_NAME("reaction",itsClass_0.IS_IN(reaction_state));
```

- Example code in C:
```
RiCGEN(&(me->itsClass_0),evX());
RiCOXFDelay(1000);
RiCIS_IN(&(me->itsClass_0),reaction_state);
```

- Example code in Java:
```
itsStopWatch.gen(new evPressKey(1));

try {

  wait(4000);

} catch(Exception e)

{ }

TestConductor.ASSERT_NAME("Check state of
stopwatch",itsStopWatch.isIn(ROOT.Running));
```

TestCase Definition with Statecharts

Test cases can alternatively be defined using statecharts. Due to their ability to wait on timeouts, statechart test cases are particularly suited for testing reactive behavior. In order to separate test case behavior from possible reactive behavior of the test context, statechart test cases are defined using specialized test components, which are then dynamically instantiated for test execution.

Statechart test cases are comprised of the following model elements:

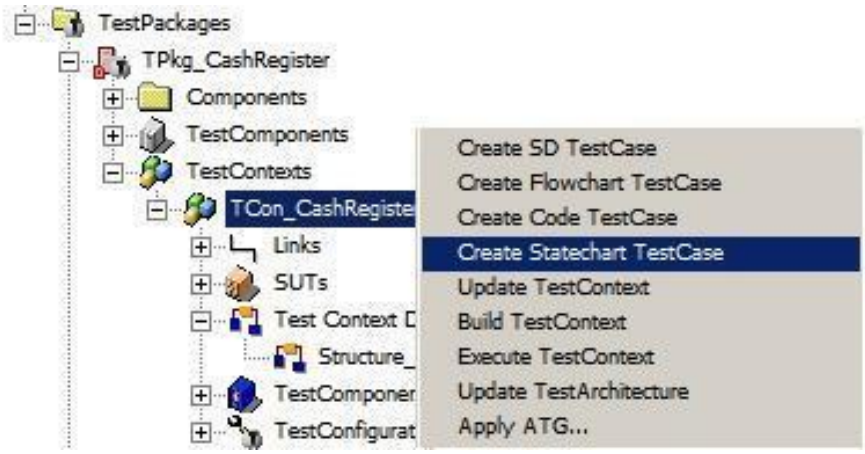
- a TestCase, i.e. basically an operation of the test context.
- a TestComponent owning the statechart defining the test case behavior.
- a dependency of the test case on the test component. This dependency is stereotyped <<StatechartTestCase>>.

This chapter gives a short overview about the usage of statechart test cases. It describes:

- How to define a simple statechart test case.
- How the model is populated for executing a statechart test case.
- How statechart test cases can be executed.

Define a Statechart Test Case

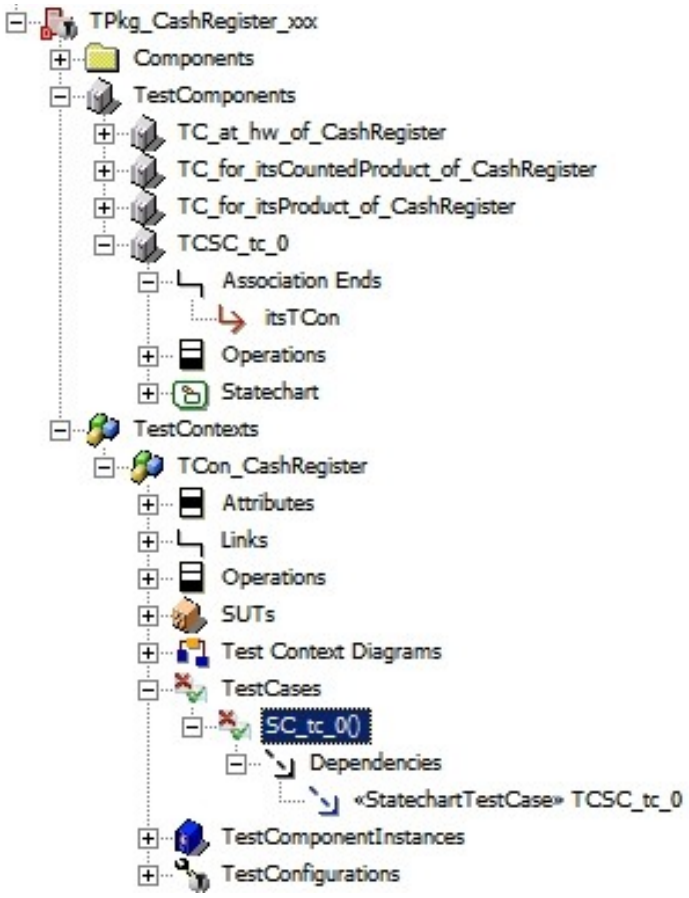
- Right-click on the test context `TCon_CashRegister` and select **Create Statechart TestCase**



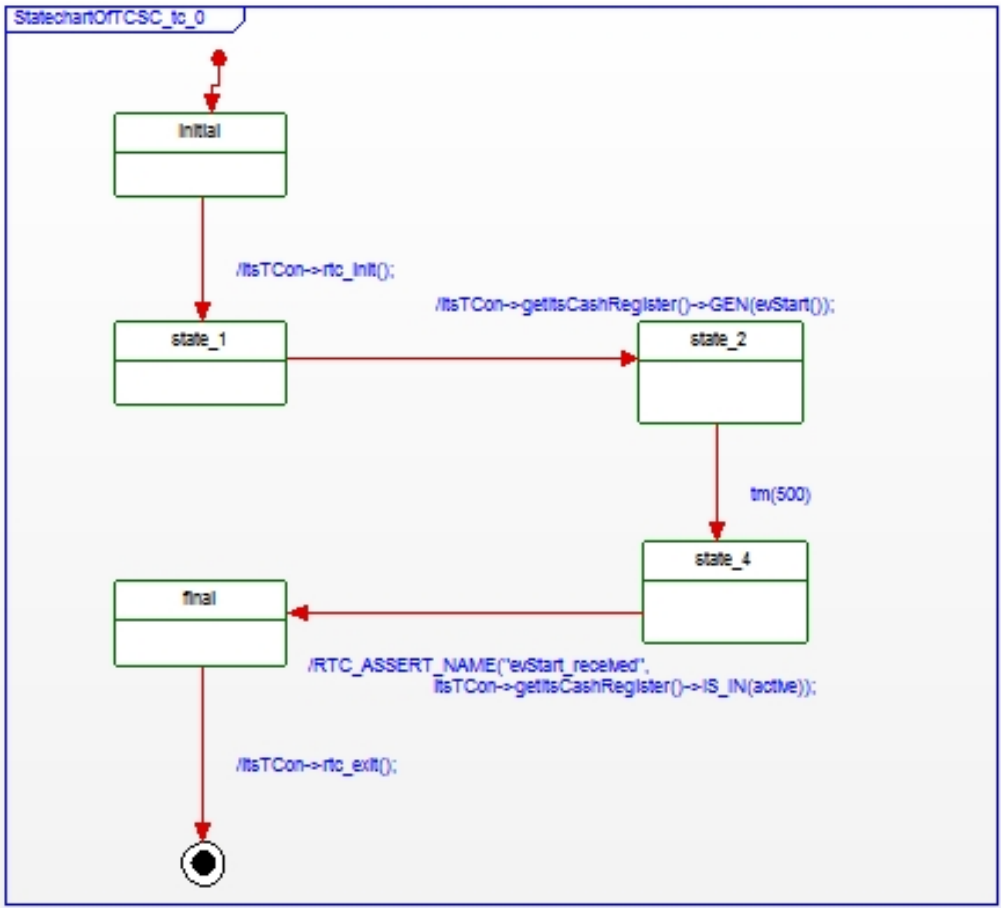
Creation of a statechart test case adds a test case to the test context. This test case has a dependency on a newly created test component owning the statechart. The test component has a directed association to the test context, which can be used to refer to parts, variables and operations of the test context. Upon execution, the statechart test case dynamically instantiates the test component, initailizes the association and starts statechart execution.

Furthermore, the test context needs to be populated with a *rtc_init()* and a *rtc_exit()* operation which are invoked by the statechart. This population is initiated by “Update TestCase”, “Update TestContext”, and “Update TestPackage”, respectively.

The following figure shows the browser after statechart test case creation:



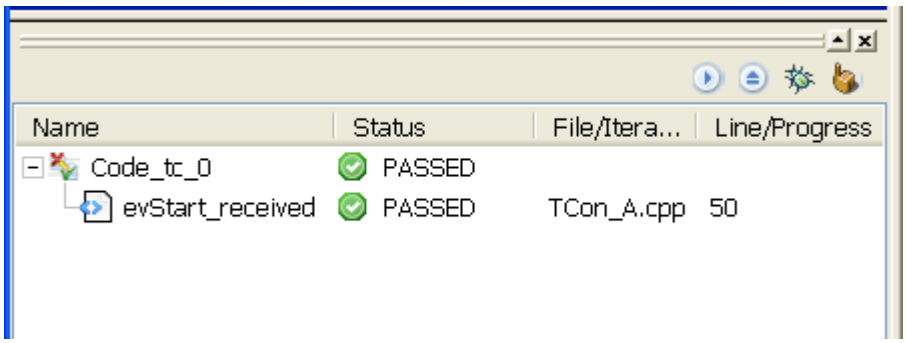
- Name the new test case “tc_statechart”
- Draw the following statechart



Execute a Statechart Test Case

Now you are be able to execute the test case by doing following steps:

- Right-click on the test case “tc_statechart” and select **Update TestCase** from the context menu
- Right-click on the test case “tc_statechart” and select **Build TestCase** from the context menu
- Right-click on the test case “tc_statechart” and select **Execute TestCase** from the context men



The test execution dialog shows the result of the defined assertions. The assertion “evStart_received” passed the test, which means that the tested behavior is ok.

Further information about test execution and the related results is described under chapter Test Execution on page *75*.

Failure Analysis in Statechart Test Cases

TestConductor lists in the execution dialog all executed assertions. To display the corresponding assertion, select in the execution dialog the item name in the column **Name** and press the button **Show Assertion**.

Further information about failure analysis can be found in chapter Failure Analysis on page *224*.

Test Case Definition with Sequence Diagrams

Another option to define test cases is by using sequence diagrams. In the context of the Rhapsody Testing Profile such sequence diagrams are stereotyped as *test scenarios (new term: TestScenarios)*. Test scenarios 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.

This chapter gives a short overview about the usage of sequence diagram based test cases. It describes:

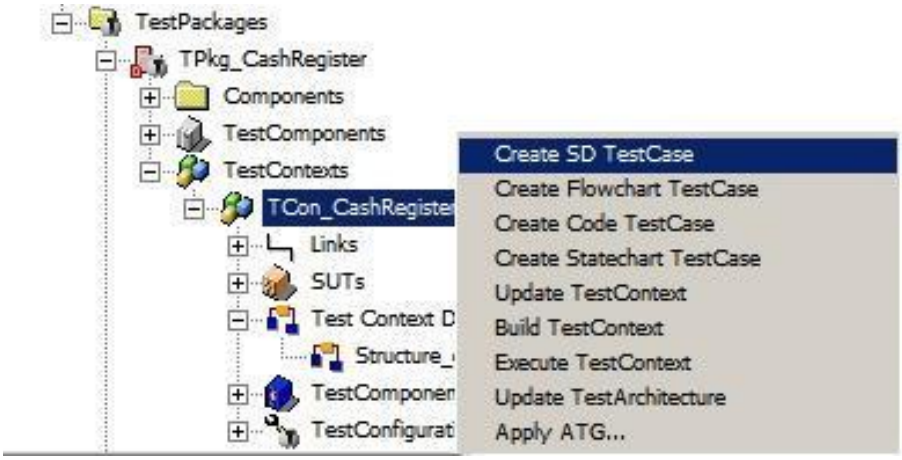
- How to define a simple sequence diagram test case
- How the generation of driver and sub operation works (see also chapter *Model Population* on page *62*)
- How sequence diagram test cases can be executed

Detailed information regarding the usage of the powerful features of sequence diagram test cases are described in chapter Advanced Test Definition on page *157 ff*.

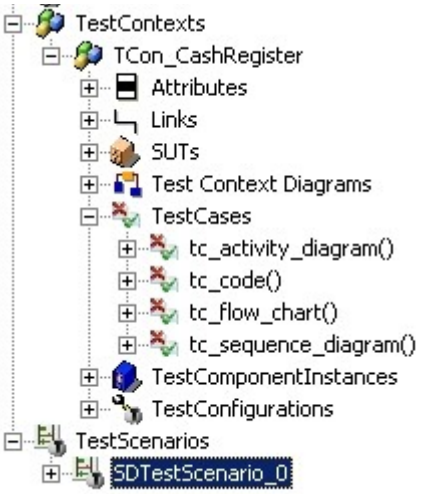
Define a Sequence Diagram Test Case

Driving the SUT using Test Components

- Right-click on the test context `TCon_CashRegister` and select **Create SD TestCase**

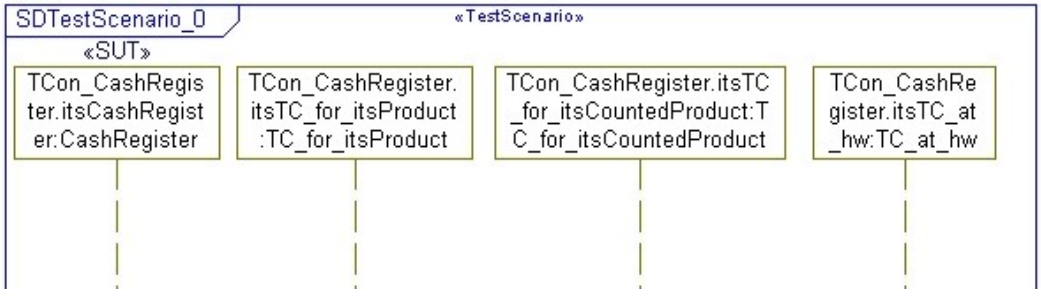


Note: TestConductor generates a new test case "SD_tc_0()" with a dependency "SD_tc_0" to a newly generated test scenario "SDTestScenario_0".

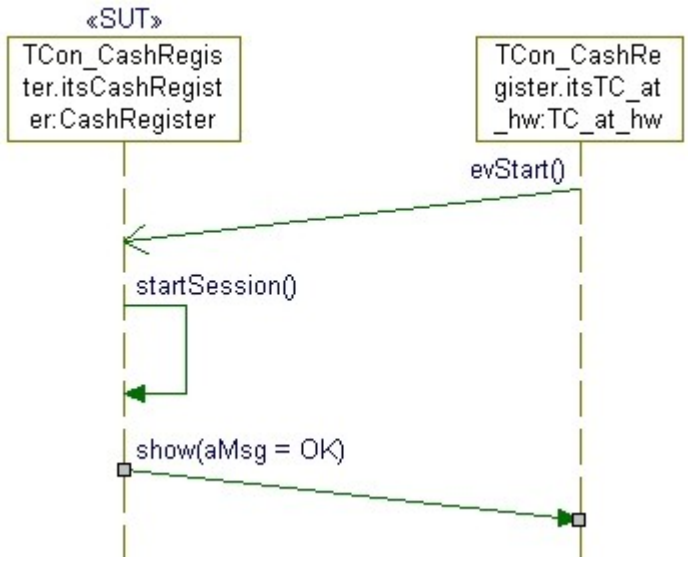


- Rename the new test case to "tc_SimpleStart"
- Rename the new test scenario to "SDSimpleStart"

The generated test scenario looks like the following diagram. It contains lifelines for each SUT and test component object defined in the test architecture.



- Remove the lifelines `TCon_CashRegister.itsTC_For_itsProduct` and `TCon_CashRegister.itsTC_for_itsCountedProduct` from the view, because these lifelines are not used in the following test scenario
- Draw the following messages into the test scenario



In this test scenario the test component `TCon_CashRegister.itsTC_at_hw` is driving the SUT with the message `evStart()`. The expected result is the message shown below `show()`.

Note: During execution parameter values containing quotes will consistently be stripped, e.g. the expression “OK” will be converted to OK and ““OK”” will be converted to “OK”.

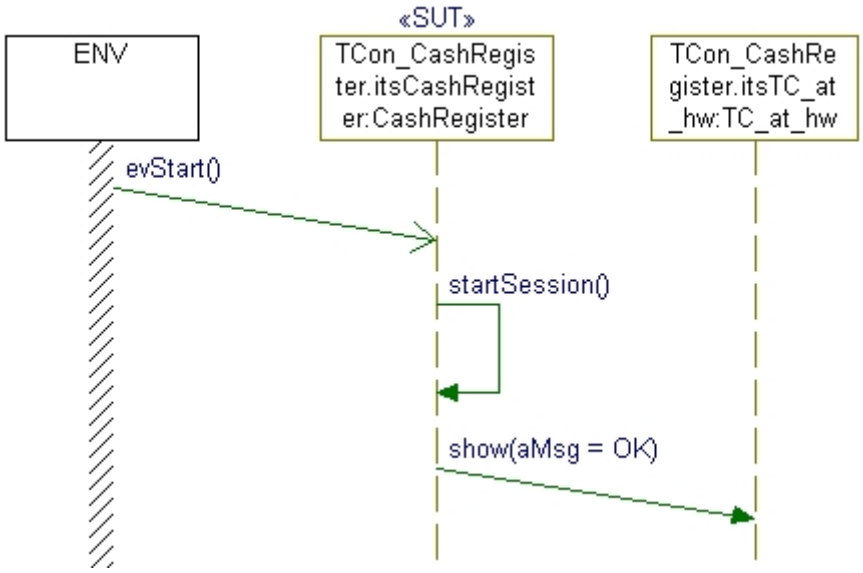
The scenario describes the normal way in which objects communicates among each other. Messages from an environment line are only necessary when messages have to be sent from the system boundary (e.g. an actor is sending an event to an object of the system).

Driving the SUT using ENV

If you are testing an animated application, inputs can also originate from the ENV life line in a sequence diagram. To define a sequence diagram test case in such a manner you have to draw a slightly different test scenario.

- Create a new test case as described above
- Rename the new test case to “tc_SimpleStartENV”
- Rename the new test scenario to “SDSimpleStartENV”

- Remove the lifelines `TCon_CashRegister.itsTC_For_itsProduct` and `TCon_CashRegister.itsTC_for_itsCountedProduct` from view, because these lifelines are not used in the following test scenario
- Add an ENV line to the test scenario
- Draw the following messages into the test scenario

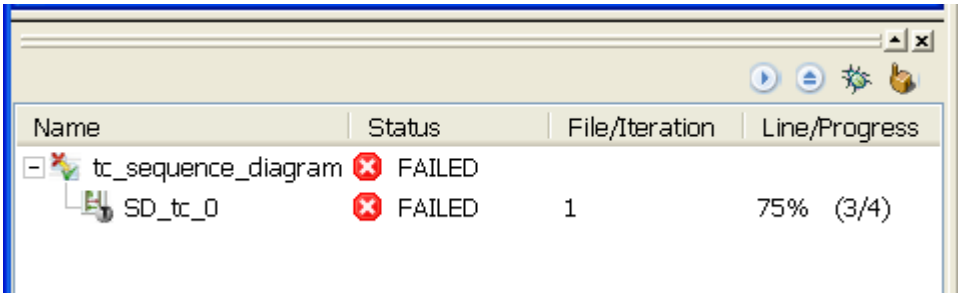


Execute a Sequence Diagram Test Case

Now you are be able to execute the test case by doing following steps:

- Right-click on the test case “tc_SimpleStart” and select **Update TestCase** from the context menu
- Right-click on the test case “tc_SimpleStart” and select **Build TestCase** from the context menu
- Right-click on the test case “tc_SimpleStart” and select **Execute TestCase** from the context menu – Alternatively you can right-click on test scenario to “SDSimpleStart” and select “Exceute TestCase of TestScenario” from the context menu.

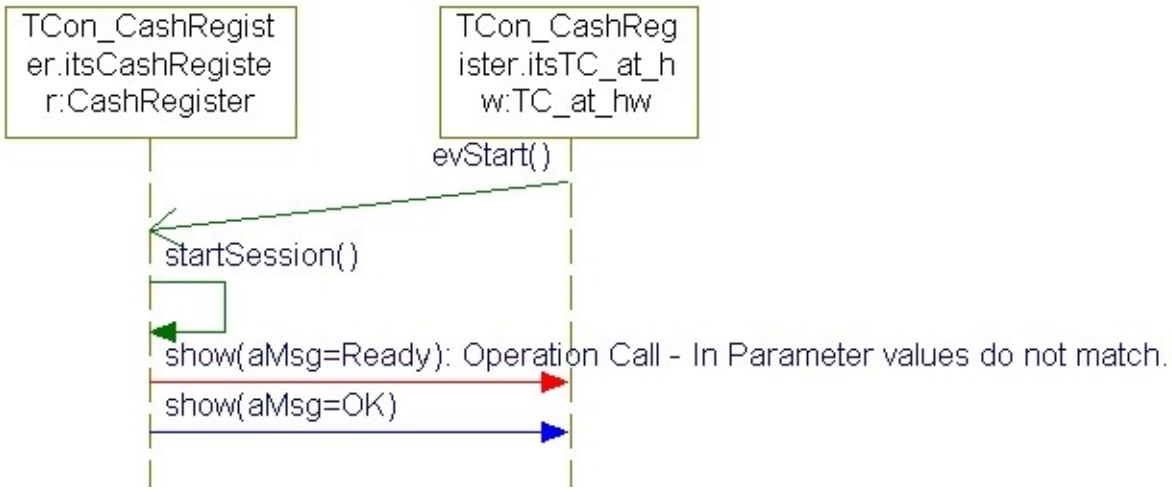
- The test is executed, and you can see the results in the execution window.



Failure Analysis in Sequence Diagram Test Cases

The execution of the test case failed. To find out why you can do the following:
Select the item “SD_tc_0” in the execution dialog and double-click the item. Alternatively, select the item “SD_tc_0” and select “Show as SD” from the context menu.

With **Show as SD** TestConductor has generated a new color coded sequence diagram which shows the found failure.



In this case the argument of the show() message sent by the SUT has a different value than expected. The expected argument value is “aMsg=OK” while the real observed value is “aMsg=Ready”. The reason for the problem is that we specified an incorrect test scenario which must be corrected now.

You can change the argument from “OK” to “Ready” in the test scenario “SDSimpleStart”. Then again perform the steps described above.

Note: During execution parameter values containing quotes will consistently be stripped, e.g. the expression “OK” will be converted to OK and ““OK”” will be converted to “OK”.

Further information about test execution and the related results is described in chapter Test Execution on page *75*.

Further information about failure analysis can be found in chapter Failure Analysis on page *224*.

Model Population – Create Driver Operations and Stub Operations

Whenever test components are used to drive input messages of the SUT or to be forced to return a pre-defined value of an operation call to the test component users have to provide driver or stub operations for test components.

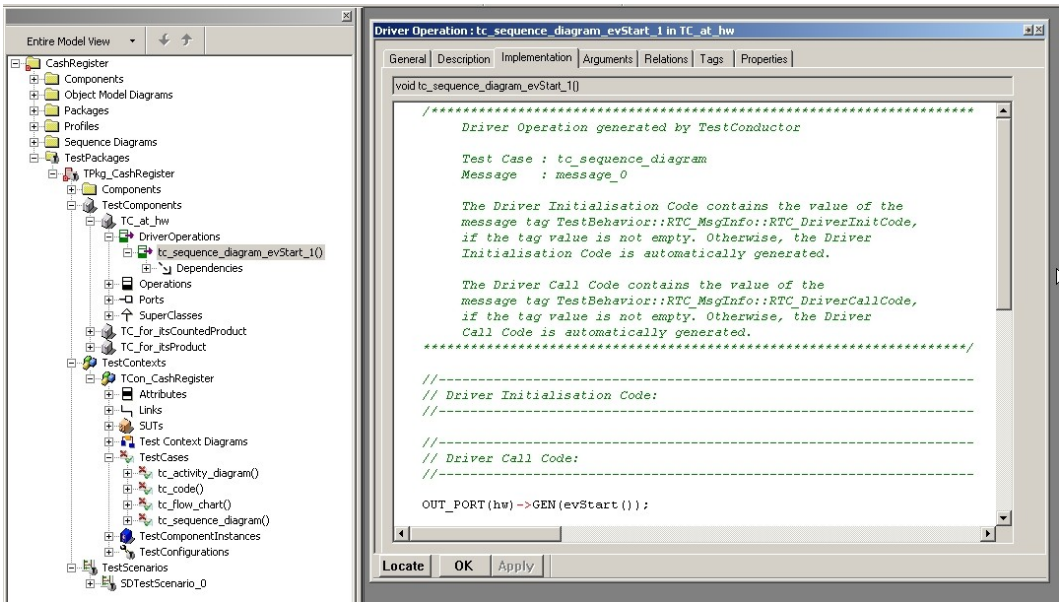
By using sequence diagram test cases TestConductor automates the generation of driver operations and stub operations. Simply by choosing the context menu **Update TestCase** on test case level, by choosing the context menu **Update TestContext** on test context level, or by choosing the context menu **Update TestPackage** on test package level the

work is done. Choosing one of these menu entries starts the so-called “model population” process of TestConductor. It analyses each defined sequence diagram instance and the linked test scenarios to generate necessary driver and stub operations for the test components.

Driver Operations

Driver operations (DriverOperations) are created for any message going from a test component to the SUT, except for messages carrying the tag *RTC_Monitor,* or messages starting at an instance line with the tag RTC_Monitor. In this case TestConductor assumes the message should not be driven. Driver operations will be generated only for messages from sequence diagrams referred by a sequence diagram instance with the mode “driver and monitor”.

For example look into the generated driver operation of the test case “tc_SimpleStart”:



TestConductor analyzed the given test architecture, the ports, and the interfaces, and then TestConductor generated a new driver operation for the test component `TC_at_hw` called `tc_SimpleStart_evStart_1()`. The implementation tab of this operation shows the generated code. Beside some comments there is the code line

```
OUT_PORT(hw)->GEN(evStart());
```

This implementation realizes the sending of the message `evStart()` from the `TestComponentInstance TCon_CashRegister.itsTC_at_hw` through the port `hw` to the SUT. During test execution TestConductor will call the driver operation `tc_SimpleStart_evStart_1()` which in turn generates the specified input event `evStart()` using the port connection (`hw`).

The name of the driver operation is the concatenation of the name of the test case, “_”, 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 identifier of the message

and the name of the test case for which the driver operation was generated. This allows the user to identify the correct driver operation if he wants to edit it.

In the context of the model-population, the identifier of a message is the value of the tag `TestBehavior::RTC_MsgInfo::RTC_MsgId`. TestConductor generates such an identifier for a message when needed, using the naming scheme '`message_<unique_number>`'.

The visibility of the driver operation will be public, the property `CG.Operation.AnimAllowInvocation` of this operation will be set to "`All`" 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 on the destination instance itself or via a port, this is derived from the test context).

The values of any input argument for the driven operation call is derived from the specification in the sequence diagram, the specified return-value(if existent) and the specified output argument values are stored in local variables. TestConductor makes sure that the call is done on the correct instance of the SUT if multiple instances of the same SUT class exist.

If the sequence diagram specifies that the returned value should be checked, the macro RTC_ASSERT_SD_NAME is used to check if the returned value and the expected returned value are equal. The same macro is used to check if out or in/out argument values returned by the operation call are as specified in the sequence diagram. If any of these checks fails the test case fails.

The values of parameters defined for the sequence diagram instance are propagated to the driver operation this way: If any parameter is used in the argument value- or return value specification of the operation that should be driven, then in the body of the driver operation the argument-value or return-value is substituted with the value of the parameter. A corresponding substitution is taken into account, if sequence diagram parameter values are used as sequence diagram instance names.

For further information how to customize the driver operation please read the chapter User Defined Driving Operation Calls at page *208*.

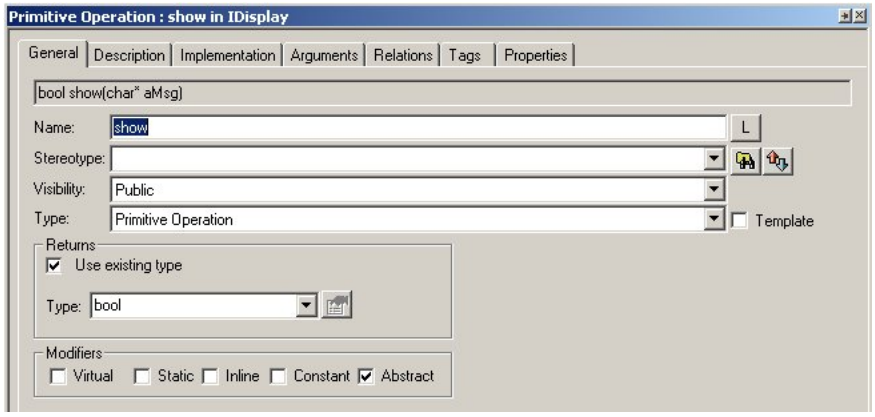
Stub Operations

Typically *stub operations (StubOperations)* are used to return a special return value for an operation call that is needed to test a special behavior of the SUT that depends on this return value.

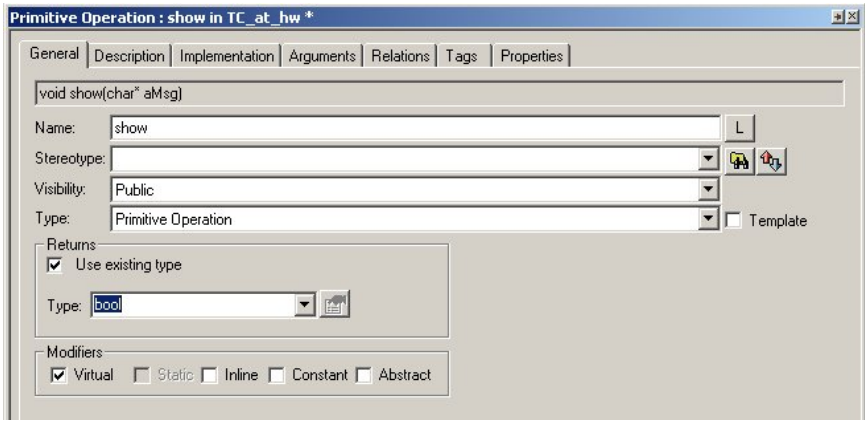
Stub operations are created for any operation call in the sequence diagram going from the SUT to a test component if a return value (or an out value for an out or in/out argument) is specified for this operation. TestConductor needs the ability to determine and control the value returned by the operation. On the other hand there might be some calls to the same operation without a specified return value or the operation is called by a test component on a test component. Because of this TestConductor has to generate a different body for the operation, but it must still be possible to call the original operation.

To show this in an example you have to do some model changes:

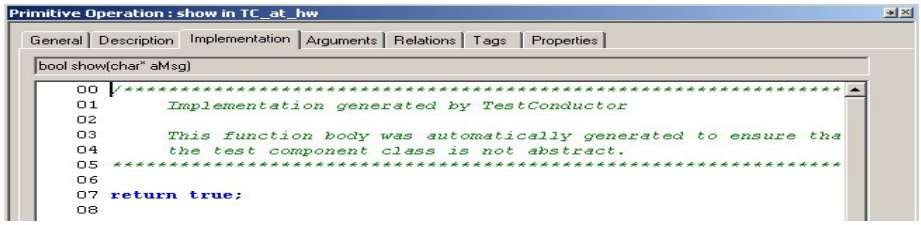
- Open the feature dialog of operation `show()` of class `IDisplay` in package `InterfacePkg`
- Change the return type from `void` to `bool`



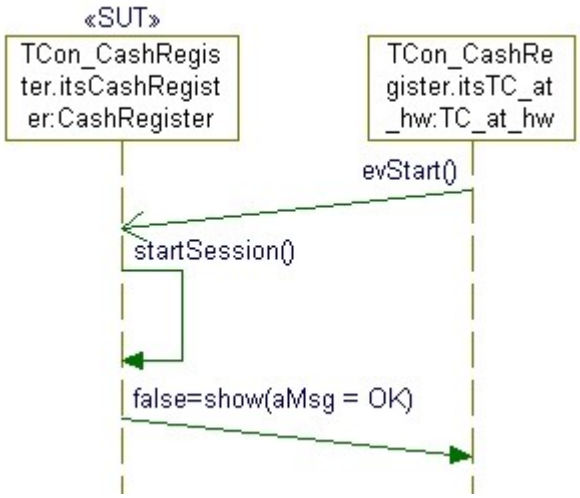
- Open the feature dialog of operation `show()` of the test component `TC_at_hw` in package `TPkg_CashRegister_0`
- Change the return type from `void` to `bool`



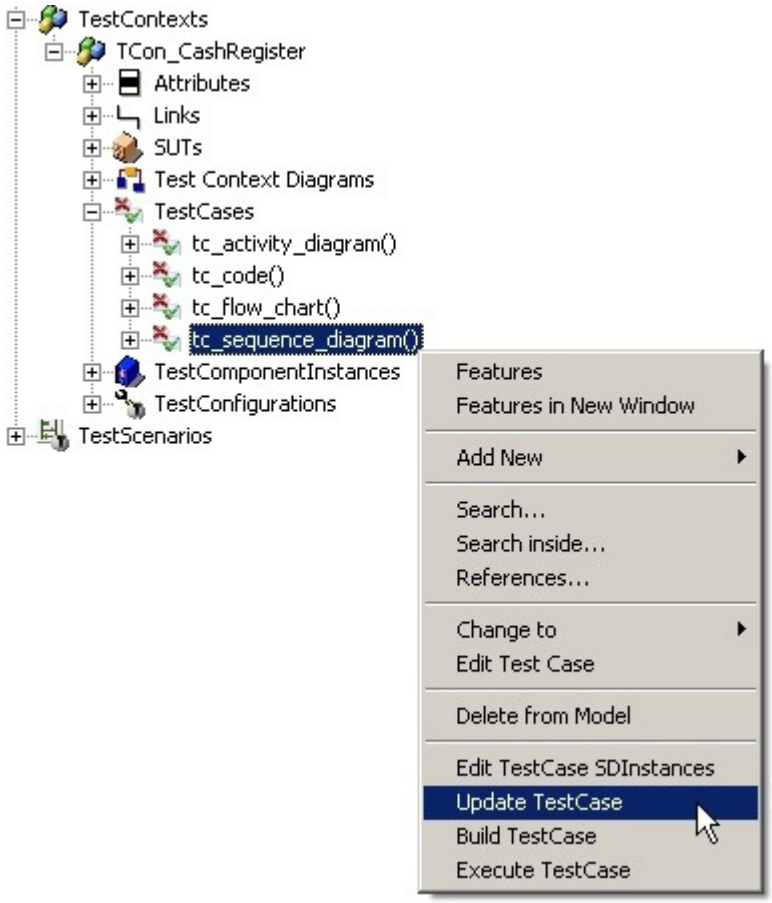
- Change the implementation of the operation `show()` from "`return`" to "`return true`".



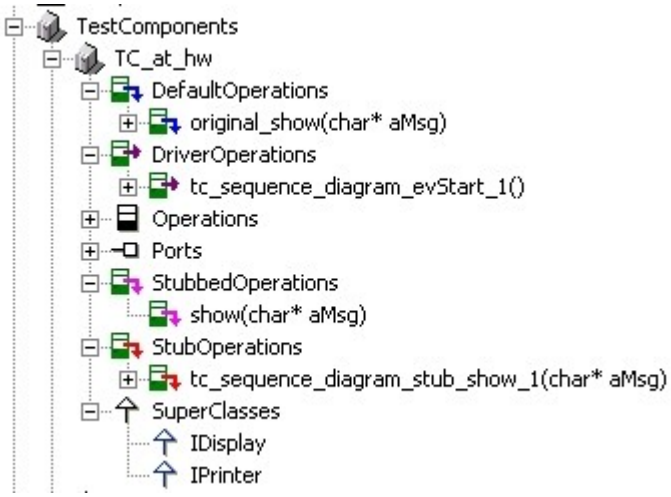
- Define a return value `false` for the message `show()` in the test scenario "SimpleStart".



- Choose **Update TestCase** from the context menu of test case “tc_SimpleStart”

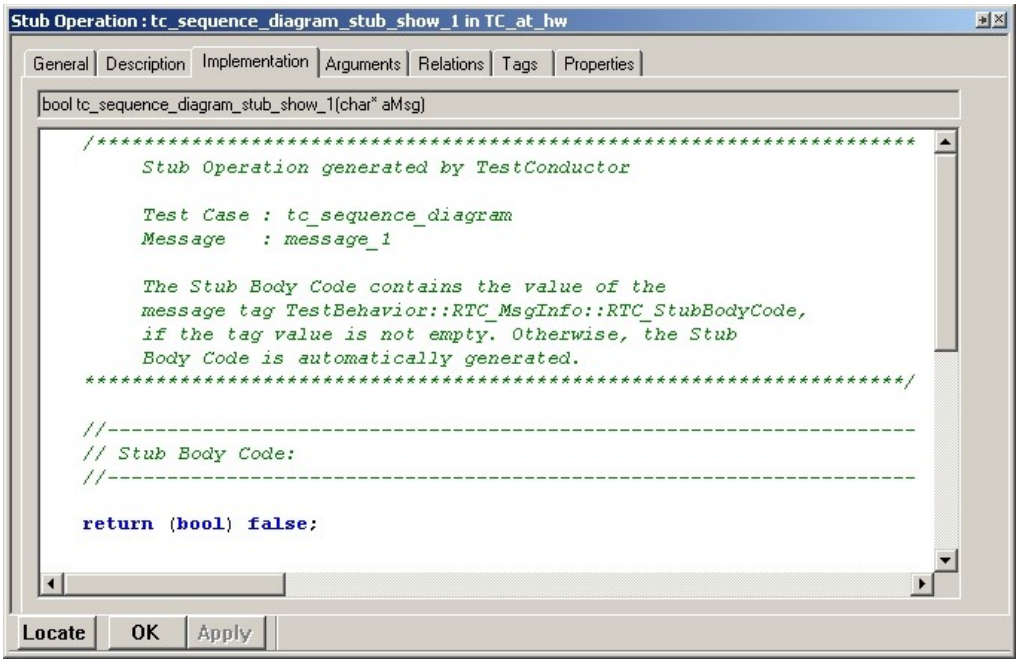


The result of the update and model population process can be seen in the Rhapsody browser (see following figure)

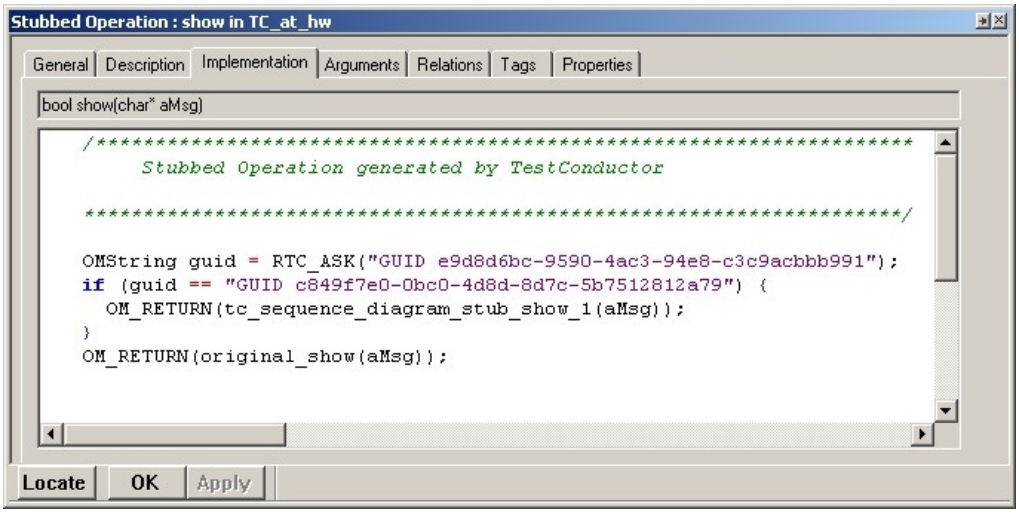


TestConductor has done some modifications within the test component `TC_at_hw`.

- The operation `show()` has been renamed to `original_show(..)` and is stereotyped with *DefaultOperation.*
- A new stub operation `tc_SimpleStart_stub_show_1()` has been generated. The generated stub operation returns a value `false` needed for the test case “tc_SimpleStart” .



- A new stubbed operation `show()` has been generated.



The stubbed operation `show()` replaces the original operation `show()` and is called always when the SUT calls the operation `show()` on the specified test component. This operation immediately decides whether the original show message has to be called or if a stubbed value shall be generated. This behaviour is realized on a per test case and on a per message basis.

Note: Each message in a sequence diagram has a unique Rhapsody GUID. So TestConductor is able to uniquely identify each message with in a sequence diagram.

For further information how to customize the stub operation please read the chapter User Defined Stub Operation Calls at page *213*.

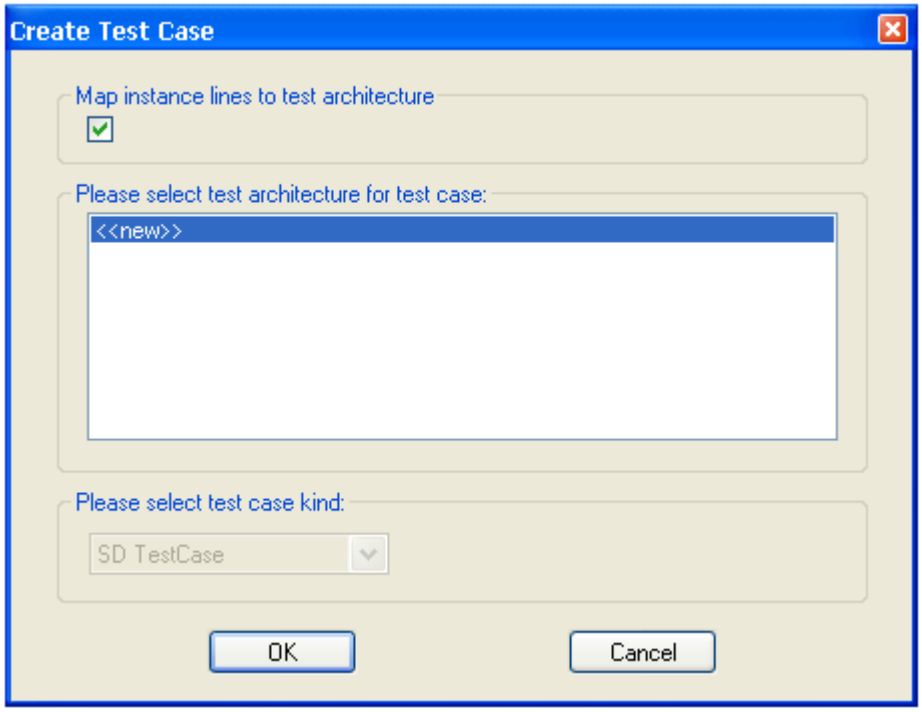
Creating test cases with the test case wizard

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

- Sequence Diagrams
- Operations and Event Receptions
- Requirements

In order to create a test cases based on an existing Sequence Diagram, do the following:

1. In the browser or in the sequence diagram editor, right click the sequence diagram and select “Create TestCase…”. This opens the test case wizard dialog:

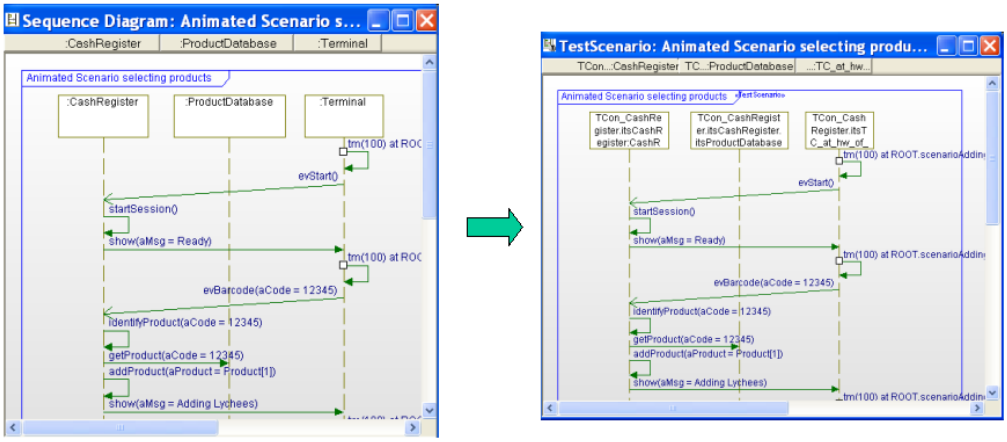


2. In the test case wizard dialog, all test architectures (i.e., all test contexts) that are suitable to map the life lines of the existing sequence diagram to the life lines that are available in the test architecture (i.e., the life lines of the SUT instances and the life lines of the test component instances) are listed. A test architecture is suitable, if

- All life lines of the existing sequence diagram can be mapped to life lines of SUT instances or test component instances s.t. all specified messages can occur also between the remapped life lines of the test architecture.
- 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 test architecture. 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 test architecture. This mode allows to remap an existing sequence diagram also to test architectures 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 test architecture is found, the list contains only the element. <<new>>. When selecting <<new>>, a new dialog will open that lists all classes of all life lines of the selected sequence diagram. In this dialog, one has to choose one of the listed classes as the SUT class for the new test architecture. After pressing ok, a new test architecture will be created for the selected SUT class.

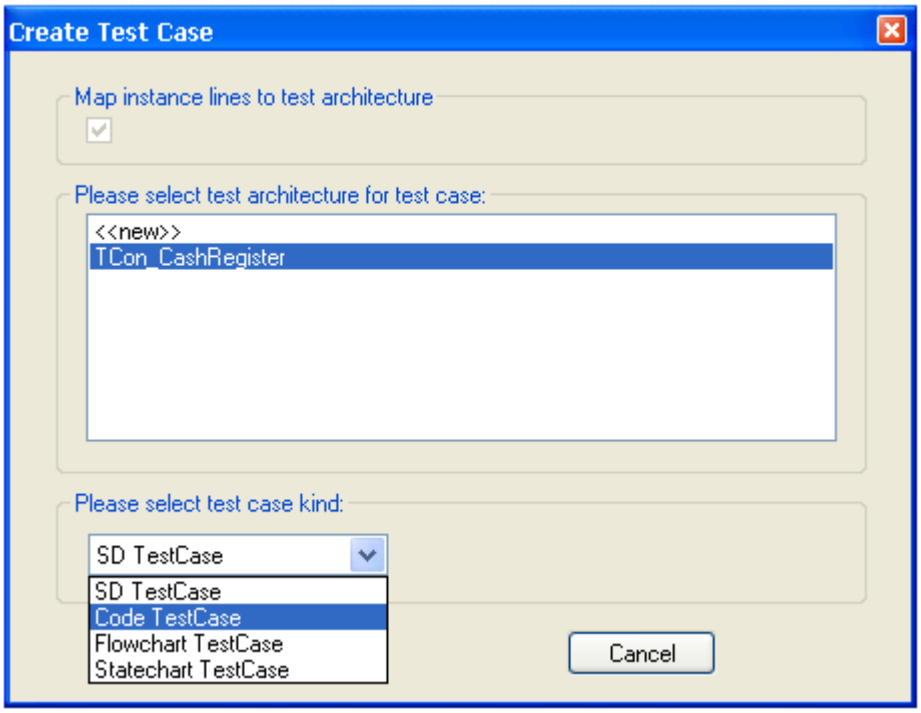
4. As a result, a new sequence diagram test case will be created that contains the same messages as the original sequence diagram, but the life lines of the test architecture.



In order to create a test cases 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 test case wizard dialog, all test architectures (i.e., all test contexts) 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. Furthermore, a dropdown box can be used to select the kind of test case one wants to create. Depending of the selection of the test architecture and the test case kind, a new test case is created and added to the selected test architecture. When <<new>> is selected, a new test architecture for the class (or file/object) of the selected operation is created.



3. The created test case 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 test cases 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 test case wizard dialog, all test architectures (i.e., all test contexts) of the model are listed. Additionally, the element <<new>> is listed. Furthermore, a dropdown box can be used to select the kind of test case one wants to create. Depending of the selection of the test architecture and the test case kind, a new test case is created and added to the selected test architecture. When <<new>> is selected, a new test architecture (a subsequent dialogs asks for the class for which a new test architecture should be created) is created. Furthermore, the original requirement for which the new test case has been created is linked as a test objective to the test case.

Creating Sequence Diagram test cases from existing Scenarios using an explicit instance mapping

Creating Sequence Diagram test cases from existing Scenarios can be done either fully automated using the Test Case Wizard (page 68) or by explicitly providing a mapping of the classifiers of the source scenario to classifiers in the test architecture for which the test case will be created.

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

The heuristics of the mapping algorithm maps classifiers of the source scenario to 'compatible' – according to the chosen mapping strategy (weak or strict, cf. pages 68 ff. and 132 ff.) – classifiers in the selected test architecture. 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 test case wizard is only capable of an instance to instance mapping. Merging or splitting instance lines – e.g. according to composite and part relations – is not supported by the test case wizard.

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

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

For a user defined mapping and a determined test architecture, the test case is created in any case and a detailed report provides feedback about the individual actions the algorithm performed for test case creation and scenario mapping. If no mapping is active on invocation of '`Create TestCase from Scenario`' the resulting test case resembles the result of invoking the test case wizard with the major difference that the test case is created in the target test architecture even though the test case wizard considers the test architecture not suitable. The 'MappingReport' comment in the created test case 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 instance lines of classifiers into a set of instance lines of particular classifiers – as needed e.g. for mapping a composite to its parts,
- merging instance lines of a set of classifiers to one instance line of a particular classifier – as e.g. used in mapping parts to its parent composite.

Once created mappings are part of the model (TestingPofile model element `SDMapping`) and can be shared for further test case creations. Definition of mappings is described on page 73.

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

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

- 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.
- The target test architecture is determined by setting one of its code generation configurations active. The active code generation configuration must be stereotyped `<<TestingConfiguration>>` or `<<AnimationBasedTestingConfiguration>>` or by a stereotype inheriting from one of them.
- Invocation of '`Create TestCase from Scenario`' on a sequence diagram or a `TestScenario`.

The testing cookbook provides examples for reusing scenarios for test case creation.

Definition of mappings for sequence diagram test case creation from existing scenarios

Testing profile model elements

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

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

These model elements have – depending on their meaning to the mapping – tags '`Origin`' and '`Target`' of type ModelElement[2].

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. Instance lines referring to '`Origin`' shall be mapped to '`Target`'.

[2]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.

- SDInstanceRealizationSplit – splitting instance lines of into a set of instance 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 instance lines referring to 'Origin' classifier shall be split.
- SDInstanceRealizationMerge - merging instance lines of a set of classifiers to one instance 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 instance lines shall be merged to an instance 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 test architecture *to which* the SDMapping maps classifiers of scenarios.

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

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

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

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.

Test Execution

During test execution, TestConductor drives events, operation calls, and dataflows sent from the test components, test context or environment to SUT objects, and monitors all messages between objects, actors and environment as specified in the test cases. This means that TestConductor automatically checks and reports whether the order of messages sent and received corresponds to the real order in the running application. In addition, TestConductor monitors the arguments of messages. Since TestConductor checks the application behavior (against requirements) using animation mechanisms, you must generate code for the test configuration with animation instrumentation switched on (at least for test components). See the *Rhapsody User Guide* for detailed information on animation settings.

Overview

TestConductor supports several kinds of execution modes

- Execution of code test cases
- Execution of flow chart test cases
- Execution of statechart test cases
- Execution of sequence diagram test cases
- Execution of a test context
- Execution of a test package
- Batch mode execution

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

Test Configuration

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

TestConductor supports test execution against different code generation configurations. In a (valid) test architecture, located underneath the TestContext there is a <<TestConfiguration>> dependency targeting a Rhapsody Configuration[3]. The algorithm TestConductor uses to choose the appropriate configuration is as following:

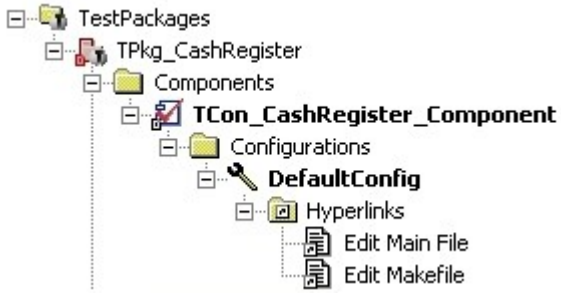
[3]In assertion based test architectures this Rhapsody configuration is required to have the stereotype <<TestingConfiguration>>.

- If the currently active configuration is located in the same component as the configuration targeted by the <<TestConfiguration>> dependency of the TestContext use the currently active configuration.
- Otherwise use the configuration targeted by the <<TestConfiguration>> dependency (Default Testing Configuration) of the TestContext.

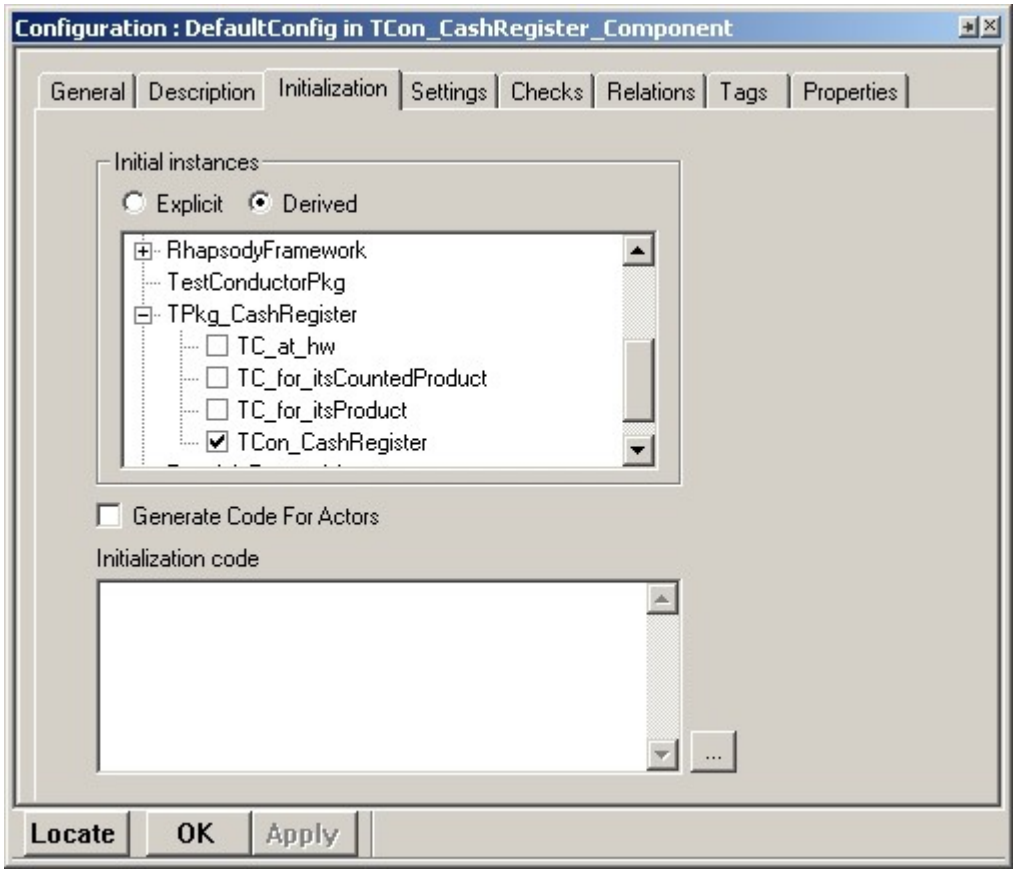
One can switch between the code generation configurations by switching the active Rhapsody configuration from those configurations in the same component as the default Testing Configuration.

Test Configuration for animation based testing

By using the automatic test architecture generation feature of TestConductor a new component and a related configuration is automatically added to the model for each test context. For example a component TCon_CashRegister_Component and a configuration “DefaultConfig” was generated automatically for the test context TCon_CashRegister.



Also the settings for the code generation are done.



Note: For test execution the instrumentation mode must be set to animation, because TestConductor needs the animation information to observe the behavior of the test context.

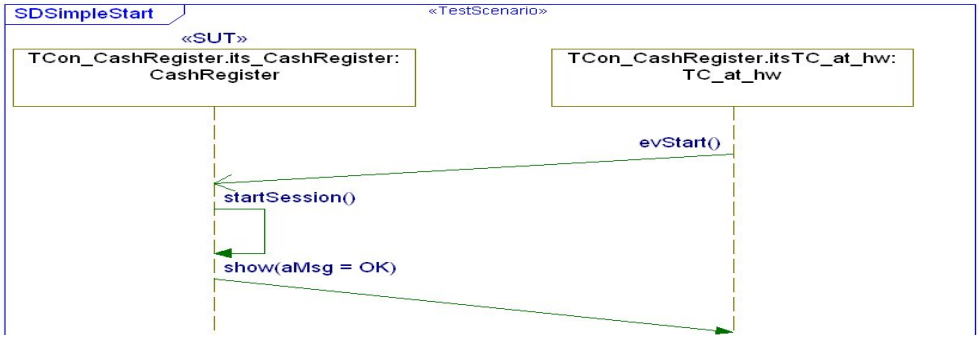
The animation mode is necessary for all elements around the SUT in the test context. In order to perform (black box) production code testing the animation of the SUT can be switched off. Thus, the test execution can be done in

- White box mode
- Black box mode

White box mode means that the test context and also the SUT classes are generated with animation code, while in black box mode the SUT classes are generated without any animation code information (production code).

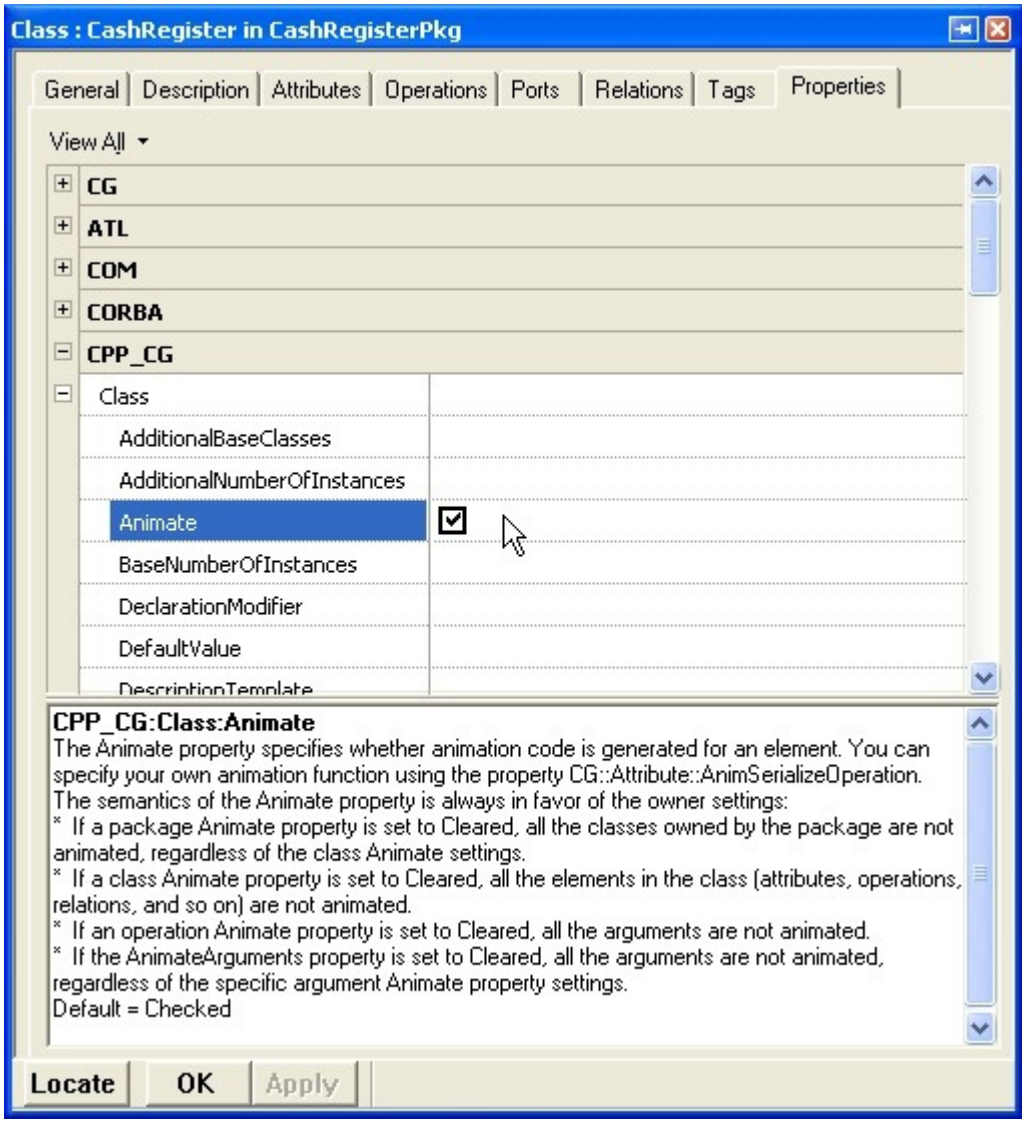
White Box Testing

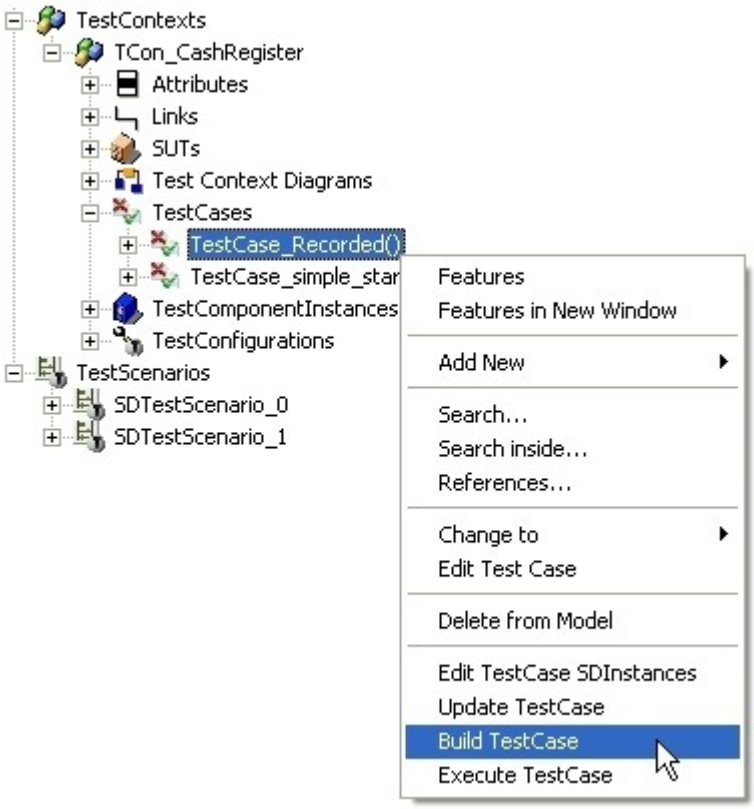
White box testing means that the internal behavior of the SUT can be observed. For example the message `startSession()` can be observed in white box mode, because the SUT was generated with animation information.



Build Test Context (White Box)

TestConductor supports the code generation for white box testing via enabling the animation of the SUT class. To enable white box testing select the property `CPPCG::Class::Animate` of the SUT class `CashRegister`.





After switching the property you have to build the test case in order to get animated code. The result of this process is an executable with animation code for the SUT object. TestConductor will automatically recognize that the SUT shall be tested in white box mode.

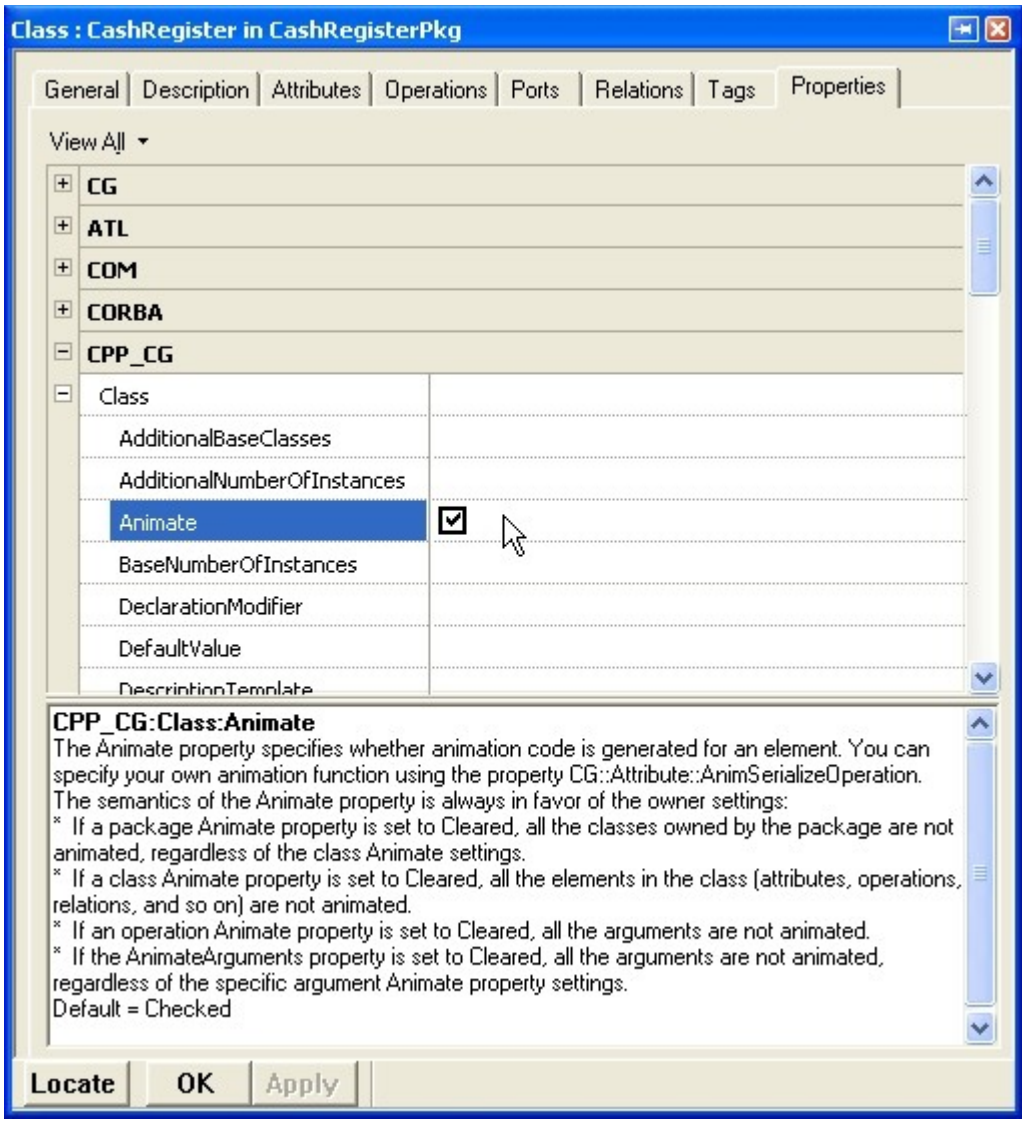
Production Code (Black Box) Testing

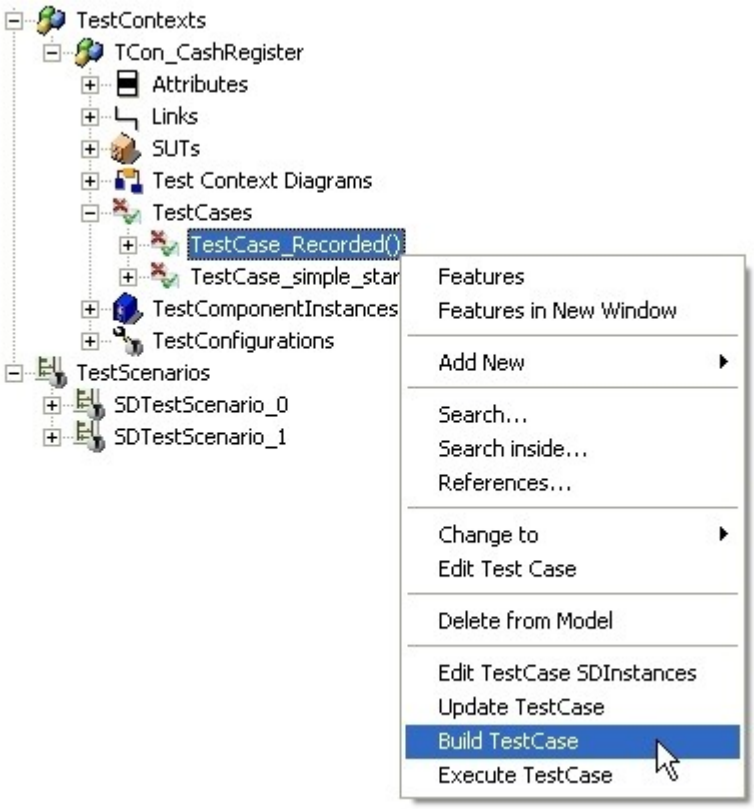
Production code or *black box testing* means that the internal behavior of the SUT can not be observed by TestConductor. The objective is to test the interface behavior of a SUT.

Note: You can use the same test cases 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.

Build Test Context (Black Box for animation based testing mode)

Rhapsody supports the code generation for black box testing via disabling the animation of the SUT class. To enable black box testing deselect the property `CPPCG::Class::Animate` of the SUT class `CashRegister`.


Class : CashRegister in CashRegisterPkg
General
Description
Attributes
Operations
Ports
Relations
Tags
Properties
View All
CG
ATL
COM
CORBA
CPP_CG
Class
AdditionalBaseClasses
AdditionalNumberOfInstances
Animate
BaseNumberOfInstances
DeclarationModifier
DefaultValue
CPP_CG:Class:Animate
The Animate property specifies whether animation code is generated for an element. You can specify your own animation function using the property CG::Attribute::AnimSerializeOperation.
The semantics of the Animate property is always in favor of the owner settings:
* If a package Animate property is set to Cleared, all the classes owned by the package are not animated, regardless of the class Animate settings.
* If a class Animate property is set to Cleared, all the elements in the class (attributes, operations, relations, and so on) are not animated.
* If an operation Animate property is set to Cleared, all the arguments are not animated.
* If the AnimateArguments property is set to Cleared, all the arguments are not animated, regardless of the specific argument Animate property settings.
Default = Checked
Locate
OK
Apply



After switching the property you have to build the test case in order to get non animated code for the SUT. The result of this process is an executable without animated SUT objects. TestConductor will automatically recognize that the SUT shall be tested in black box mode.

Test Case Execution

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

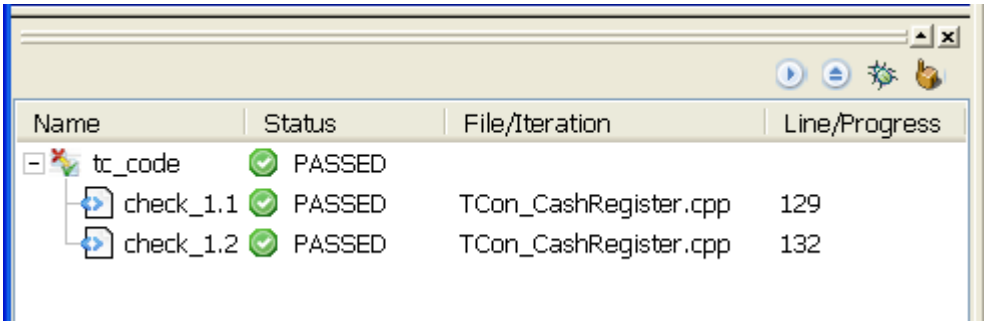
Flow chart, code, and statechart test cases are merely code based test cases, because TestConductor uses the code generation capabilities of Rhapsody's code generator. The execution dialog enables you to activate the actual test execution and displays the test results.

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

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

Test Execution Dialog

TestConductor displays the assertions defined in a code, flow chart, or statechart test case at run-time of the test case. 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.



After the test case 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 test case 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 test case is executed, it shows the progress of the SD instance.
- **Result**—Shows the result of the assertion. The possible values are *PASSED* and *FAILED*.

Controlling test case execution

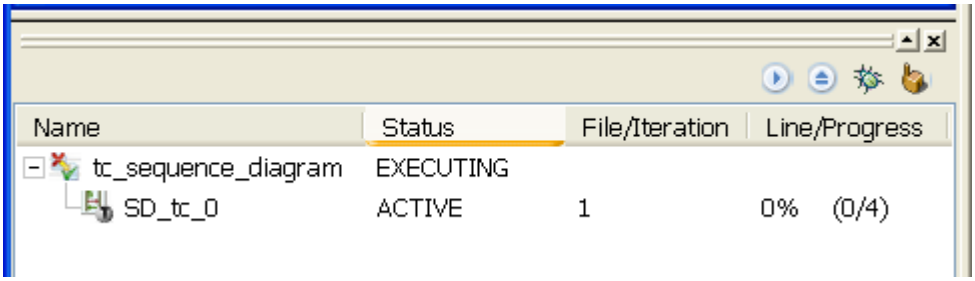
The test case execution dialog provides several functions that can be used to control the test case execution. The functions are available by pressing one of the icons in the top right corner of the execution dialog.

Test Execution Dialog for sequence diagram based tests

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

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

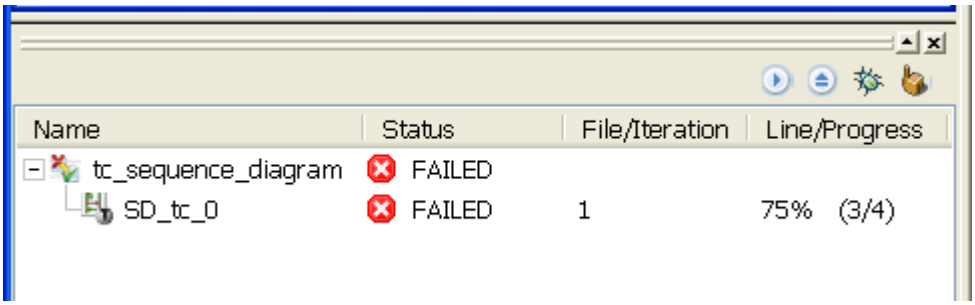
Context menu entry **Execute TestCase** of a selected test case opens the execution dialog. For a sequence diagram that is exclusively referenced by only one test case, the execution dialog can alternatively be opened using the context menu entry **Execute TestCase of TestScenario** of the selected sequence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



Test Execution Dialog

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

TestConductor displays the first iterations of sequence diagram instances without specified ordered predecessors as the initial run-time instances in the execution dialog. During test execution new run-time instances are listed as soon as their ordered predecessors or previous iterations have been fully traversed. Each line in the dialog displays information about one sequence diagram run-time instance, including intermediate and final results, as shown in the following figure.



Since the test is still running you cannot modify it. However, you can verify the test configuration, the activation conditions of the sequence diagram instances, and so on.

Test Information

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.

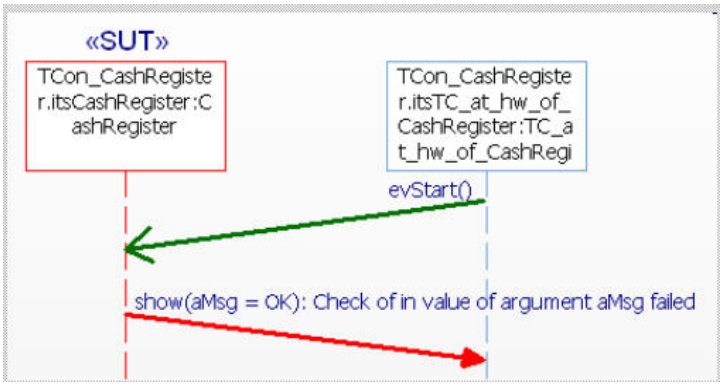
- **File/Iteration**—Shows the absolute number of the currently executed run-time instance of the sequence diagram instance under consideration. At each point in time, you can have at most one active run-time instance of an sequence diagram instance. However, over time you can have infinitely many invocations. In the example of the “tc_SimpleStart” test, only one run-time instance appears in this field, because you selected single iteration mode. An arbitrary number of run-time instances can be created during model execution if the execution mode of a sequence diagram instance is set to multiple iteration with a concrete number.
- **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 specifies two ordered message actions. TestConductor displays the progress as “percentage X/Y”. The X stands for the number of actions that passed; Y stands for all the actions specified in the sequence diagram. For example, this test failed at 75%, and 3 out of 4 actions passed.

Displaying Test Results by witness scenarios

You can display the test results graphically in order to analyze the states of a run-time instance at different points in time.

For example, to display a failure in the “tc_SimpleStart”, do the following: To see the graphical representation of the results, select a run-time instance in the list and select **Show as SD** from the context menu. A recorded sequence diagram is displayed, showing the actual order of the messages passed through the model simulation.



The resulting sequence diagram can be used for failure analysis or can be saved for further documentation.

In the sequence diagram created for a run-time instance, the following messages are displayed:

- Messages that have already occurred in the executed application. Observed messages are shown in *green*.
- Messages that are missed. Expected but not seen messages are shown in blue.
- A message that has wrongly arrived or parameter values that do not match. Messages that are observed in not expected order (failure) are shown in *red*. If a parameter or return value of the message is wrong, per default the observed value is shown in the witness scenario (assertion based testing mode with option rtc_assert_handling set to by_string).

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:

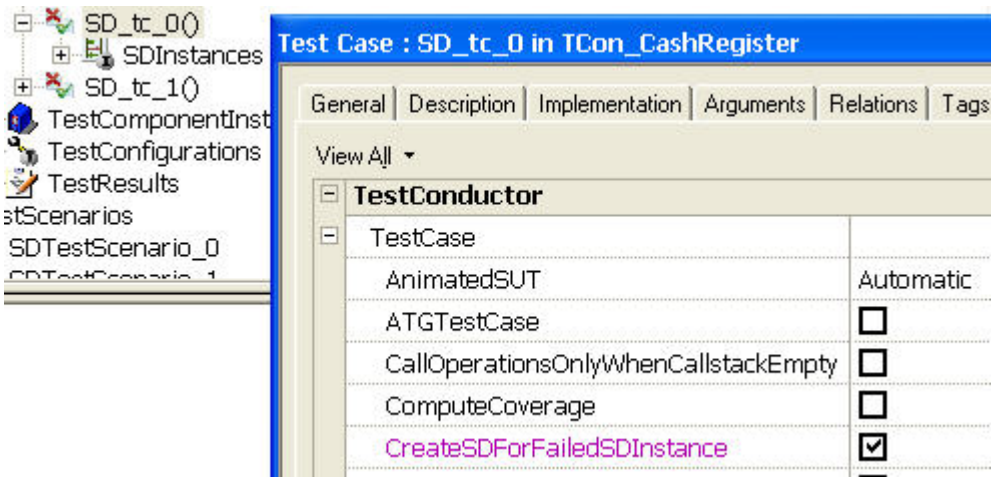
- Sending out of order
- Event Sending - Parameter values do not match
- Event Sending - Parameter values not in range
- Consumption out of order
- Event Consumption - Parameter values do not match
- Event Consumption - Parameter values not in range
- Operation Call out of order
- Operation Call - In Parameter values do not match
- Operation Call - In Parameter values not in range
- Operation Call returned - Return value does not match
- Operation Call returned - Out Parameter values do not match
- Operation Call returned - Out Parameter values not in range
- DataFlow Message - Value does not match
- DataFlow Message - Value not in range
- DataFlow Message out of order

See page *224* for more information about failure analysis.

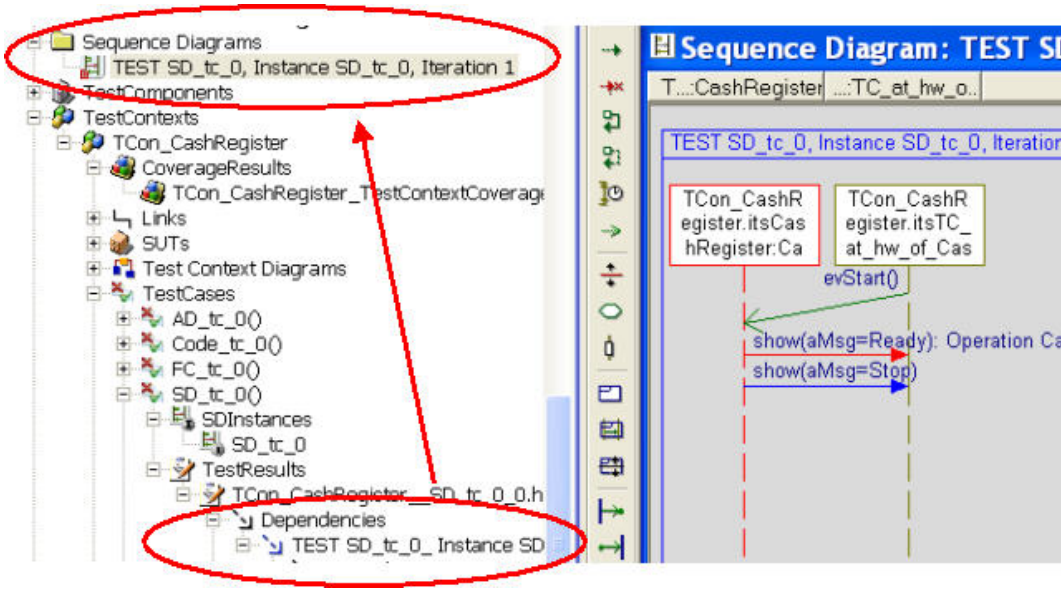
Note: When doing Show as SD in animation based testing mode, the color coded scenario is not permanently added to the model. It is intended for analyzing the current test execution, not for documentation. After closing the diagram or the model the scenario will be lost.
To permanently add a witness scenario to the model in animation based testing mode, select **Add to model** instead of Show as SD in the context menu.
In assertion based testing mode, each time you do a “Show as SD”, TestConductor automatically adds a color coded scenario to the model. The color coded scenario is added to the model to the same owner as the original specification scenario. By default, the test case operation is the owner of the specification scenario.

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 test case execution, e.g. for documentation purposes or if test cases are executed in batch mode and failed test cases are analyzed later. In order to do this in animation based testing mode, switch on the property "TestConductor.TestCase.CreateSDForFailedSDInstance":



Now, after executing a test case that has switched on this property, TestConductor automatically adds a scenario to the model showing the reason of the test case failure. Additionally, a dependency is added to the TestResult of the executed test case linking the TestResult to the added SD. This dependency can be used to navigate directly from the TestResult to the SDs that have been added for the failed SDInstances.



In assertion based testing mode, switch on the tag "CreateWitnessScenarioForFailedSDTestCase" on the code generation configuration used for test execution. If this tag is enabled, TestConductor will automatically create a witness

scenario for each executed sequence diagram test case which did not pass and add it to the model.

Abort Test Execution

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

Execution Timeout

Execution timeout for animation based testing

The testing profile defines a global timeout, which can be overwritten for every test package, test context and test case. This default value is 600 seconds.

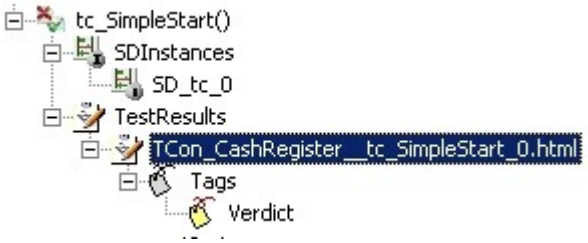
You may define a *timeout* for every test case separately via the property

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

In case a timeout is defined and the application does not show any activity for <value of timeout> seconds the execution of this test case is interrupted. In this case, this test case will be marked as "*timeout*" in the result report.

Test Execution Report

After the execution of a test case has finished and the execution dialog has closed, an execution report is written into a *HTML file*. This file is added to the test case as a controlled file.[4] If a report file already exists it is overwritten, only the report of the last execution is stored in the model. If a test case 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 test case execution.

[4]Note that with the property TestConductor.Settings.ReportLocation (see page 134) a user can specify a dedicated report location)

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

A double click on the test result "`TCon_CashRegister__tc_SimpleStart_0.html`" opens the linked HTML test report.

Test Case Result

Test Case: tc_SimpleStart

16:25:24, Tuesday, March 06, 2007

Environment Info	
Test executed on machine:	NBOSC38
Test executed by user:	rsanders
Used OS version:	Windows 2000 / Windows XP
Used Rhapsody version:	Aries, build 805506
Used TestConductor version:	2.0, build 552

Tested Project	
Project:	CashRegister
Active Component:	TCon_CashRegister_0
Active Configuration:	DefaultConfig

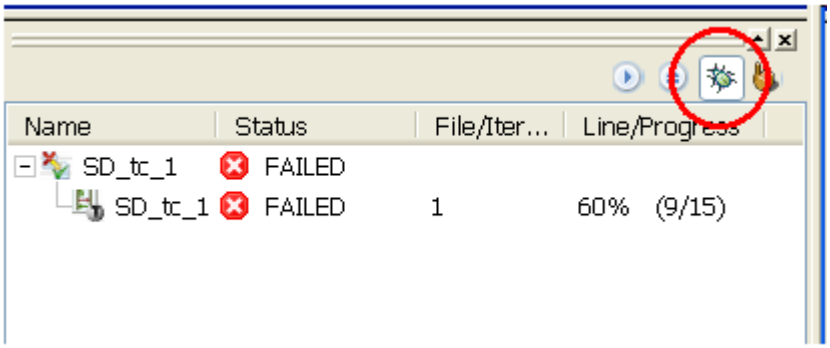
SDs used in test
TPkg_CashRegister_0::SimpleStart

Summary Info	
Total number of SDs used:	1
Total number of SD instances in test:	1
Total number of executed SD instances:	1
Total number of PASSED SD instances:	0 (0%)
Total number of FAILED SD instances:	1 (100%)
Total number of ACTIVE SD instances:	0 (0%)
Total number of NOT ACTIVE SD instances:	0 (0%)

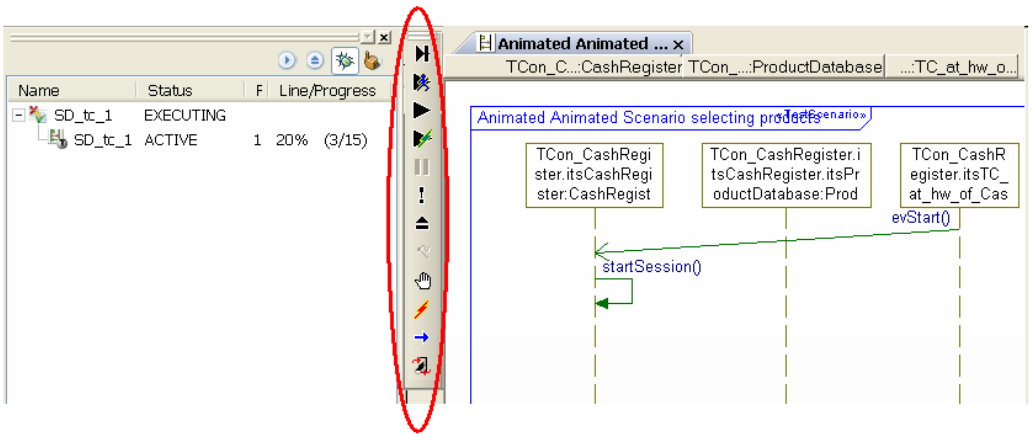
Detailed Results	
SD instance 'SD_tc_0'	
Iterations:	SD_tc_0
Status:	failed
Progress:	75% (3/4)

Debugging test cases

When a test case fails one can use TestConductor's debugging capabilities in order to find out the reason for the fail. In order to turn on test case debugging, one has to turn on "Debugging mode" in the test case execution window:



After turning on debugging mode, one can restart the test case, 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. For instance, one can step through the test case by using multiple “Go Step” commands in the animation toolbar. In the execution window, one can see the current progress of the test case, and in parallel one can use Rhapsody’s animation features (e.g. animated sequence diagrams or animated statecharts) to inspect the model during debugging of the test case. Besides “Go Step”, also all other animation commands like “Go Idle” etc. are available, e.g. one can add tracer commands etc.
This interactive, step-by-step execution of test case is available both for animation based and assertion based testing mode. But is is available only when testing applications with animation instrumentation.
Debugging a test case is possible only when executing a single test case. When executing a test context or test package the Debug button is disabled (and switched off).



Using breaks and tracer commands during debugging

In debugging mode, in addition to stepping through the test case execution using Rhapsody’s animation toolbar, one can also define breaks and tracer commands in the test cases. When a break command is reached, the test case execution is breaked at this location. When a tracer command is reached, it is simply executed. Both breaks and tracer commands can be used in all kinds of test cases.

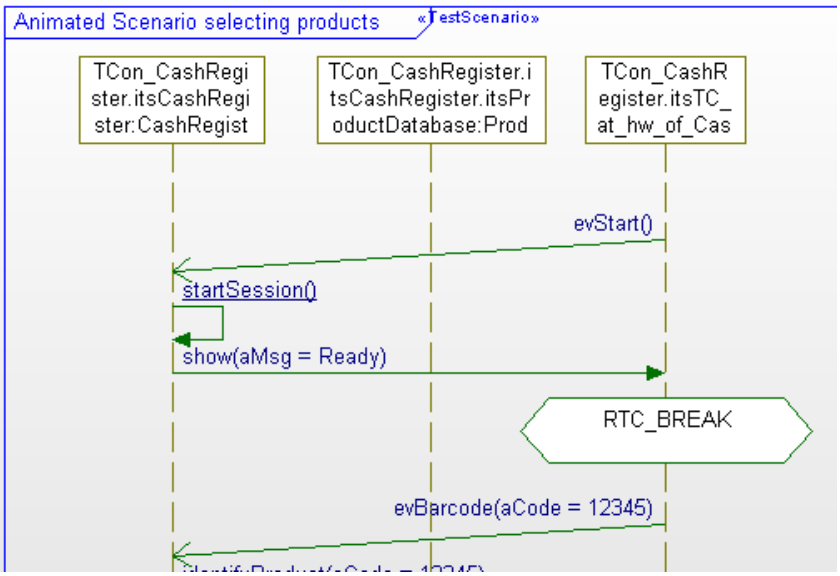
1. Defining breaks and tracer commands in code/flowchart/statechart test cases:

   To define a break in a code, flowchart or statechart test case, one has to write the macro “RTC_BREAK” (C/C++) resp. “TestConductor.BREAK()”. When the test case execution reaches the break, it is executed and the test case execution is stopped. One can proceed the test case execution by using Rhapsody’s animation toolbar (e.g. by pressing “Go Step” or “Go Idle” etc.). To execute a specific tracer command during test case execution, one has to use the macro

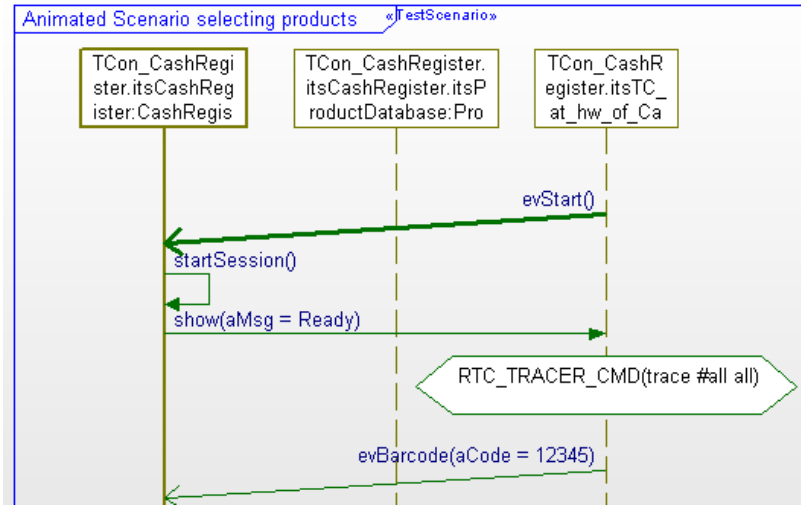
"RTC_TRACER_COMMAND(cmd)" (C/C++) resp. the function "TestConductor.TRACER_COMMAND(cmd)". For details about the supported syntax of the "cmd" argument please look into Rhapsody's User Guide. When the test case execution reaches the specified tracer command, it is simply executed as any other tracer command that was entered directly in Rhapsody's animation toolbar.

2. Defining breaks and tracer commands in sequence diagram test cases:

   To define a break in a sequence diagram test case, one has to add a condition on one of the life lines in the sequence diagram. In the condition, one has to write "RTC_BREAK". When executing the test case in debugging mode, the test case execution stops when the break is reached. In Rhapsody's animation output tab the information "Reached TestCase breakpoint" is printed.



   To define tracer commands in a sequence diagram test case, one has to add a condition on one of the life lines in the sequence diagram. In the condition, one has to write "RTC_TRACER_COMMAND". When executing the test case in debugging mode, the test case execution executes the specified tracer command when the execution reaches the position of the tracer command.

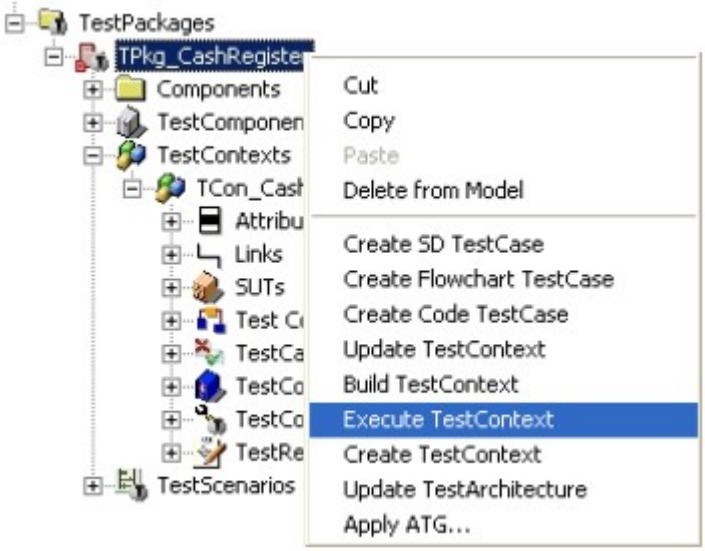


Test Context Execution

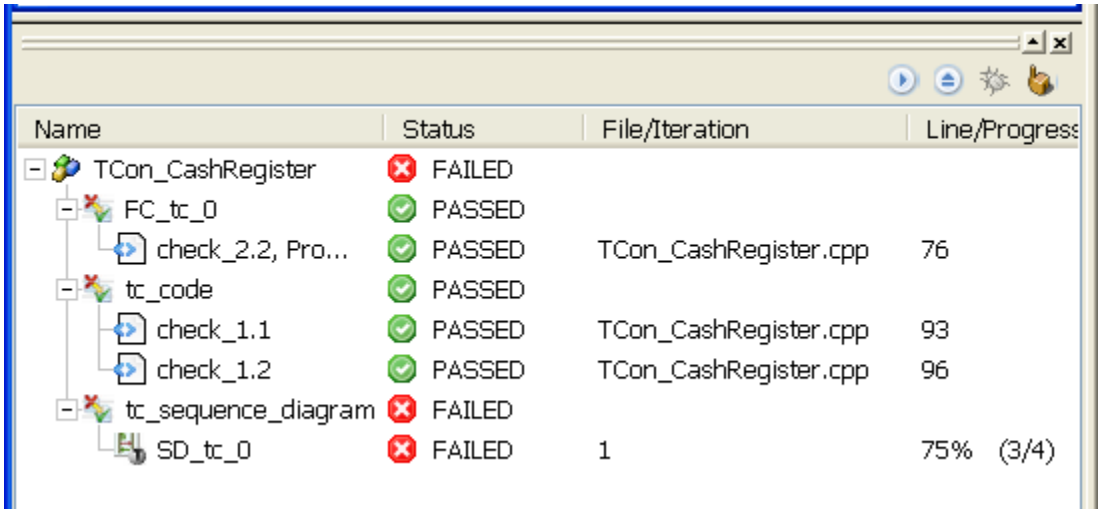
Starting Test Execution

One kind of batch execution is the execution of a complete test context. It will then execute all test cases belonging to a test context.

- Right-click on the test context `TCon_CashRegister` and select **Update TestContext.** This updates all necessary driver and stub operations derived from the defined sequence diagram test cases within the test context.
- Right-click on the test context `TCon_CashRegister` and select **Build TestContext.** This re-generates the necessary code for all elements of the test architecture and starts the compile and link process for the test architecture.
- Right-click on the test context `TCon_CashRegister` and select **Execute TestContext.** This starts the batch execution for all defined test cases within the test context.



If the user selects a test context and invokes its execution, all test cases of this test context are executed in a sequence. To terminate the execution of a test context or a test package, press the **abort icon** in the test execution window.



Stopping Test Execution

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

Execution Timeout

The testing profile defines a global timeout, which can be overwritten for every test package, test context and test case. This default value is 600 seconds.

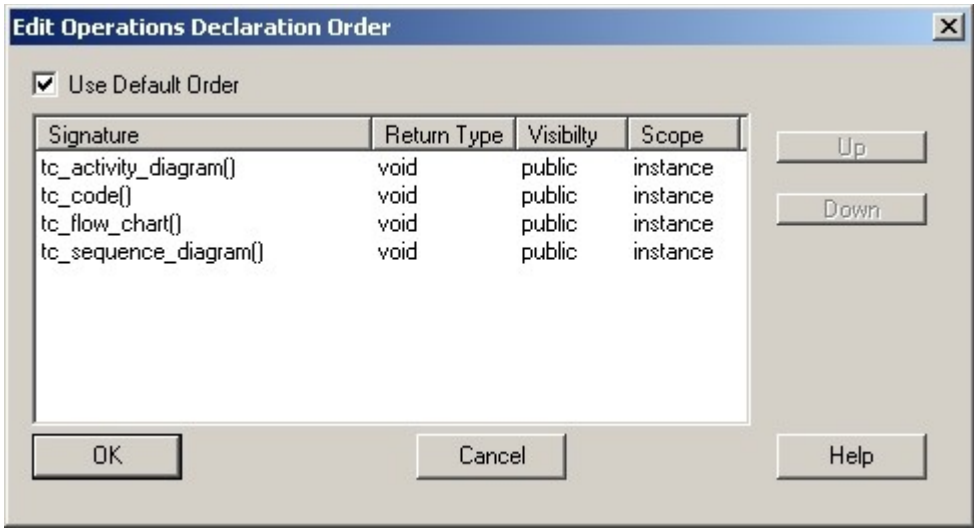
You may define a timeout for this batch mode execution of test cases individually per test case. This can be done via the property

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

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

Ordering of Test Cases

The *order of the test cases* inside the test context (similar to the "*Edit Operations Order*" in the Rhapsody browser) can be changed. In this way you can influence the execution order of the test cases.



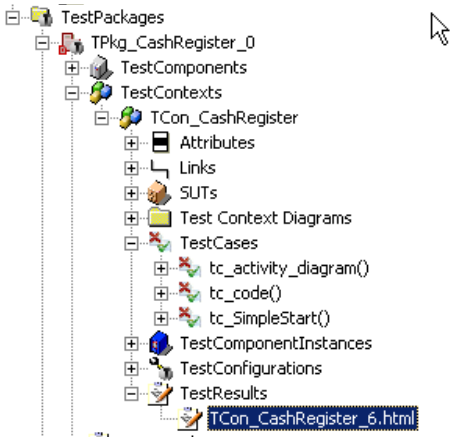
Per default the test cases are sorted and executed in alphabetical order.

Test Execution Report

After execution of each test case its result *HTML report* is written. The file is added to the test case as controlled file.[5]

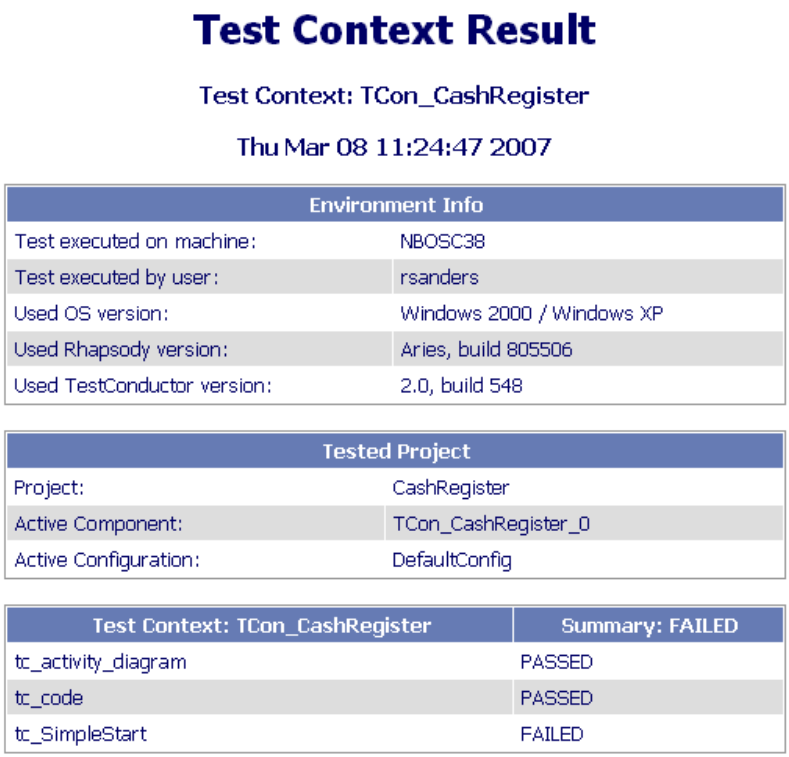
After execution of all test cases an execution report of the whole test context is written into an HTML file. The file is added to the test context as controlled file.

If a report file already exists it is overwritten, only the report of the last execution is stored in the model. If a test case or test context 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.



[5]Note that with the property TestConductor.Settings.ReportLocation (see page 134) a user can specify a dedicated report location)

- A double-click on the test result "`TCon_CashRegister_6.html`" opens the linked test report.



Test Context Result

Test Context: TCon_CashRegister

Thu Mar 08 11:24:47 2007

Environment Info	
Test executed on machine:	NBOSC38
Test executed by user:	rsanders
Used OS version:	Windows 2000 / Windows XP
Used Rhapsody version:	Aries, build 805506
Used TestConductor version:	2.0, build 548

Tested Project	
Project:	CashRegister
Active Component:	TCon_CashRegister_0
Active Configuration:	DefaultConfig

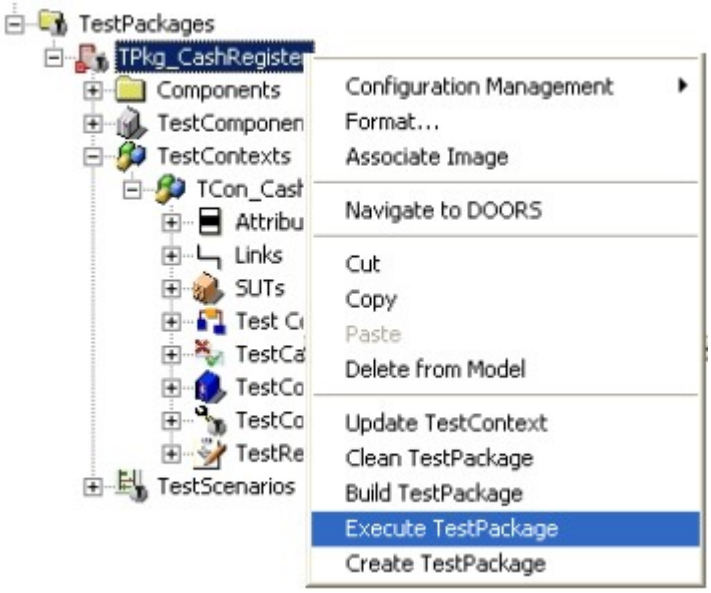
Test Context: TCon_CashRegister	Summary: FAILED
tc_activity_diagram	PASSED
tc_code	PASSED
tc_SimpleStart	FAILED

Test Package Execution

Starting Test Execution

One kind of batch execution is the execution of a complete test package. It will then execute all test cases underneath all test contexts belonging to a test package.

- Right-click on the test package `TPkg_CashRegister` and select **Update TestPackage.** This updates all necessary driver and stub operations derived from the defined sequence diagram test cases within the test package.
- Right-click on the test package `TPkg_CashRegister` and select **Build TestPackage.** This re-generates the necessary code for all elements of the test architectures and starts the compile and link process of all test architectures.
- Right-click on the test package `TPkg_CashRegister` and select **Execute TestPackage.** This starts the batch execution of all defined test cases within the test package.



If you select a test package and invoke its execution, each defined test context of this test package is executed one after the other. The procedure is almost like the execution of a test context, except the following differences:

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

Stopping Execution

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

Execution Timeout

The testing profile defines a global timeout, which can be overwritten for every test package, test context and test case. This default value is 600 seconds.

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

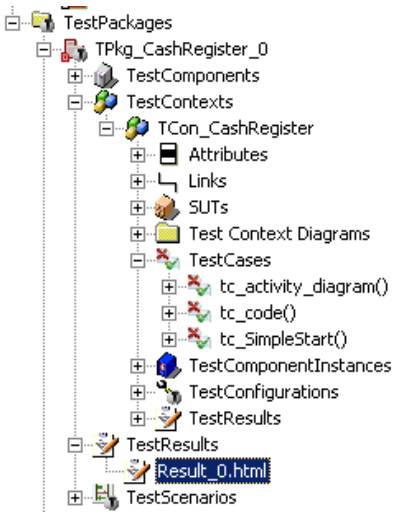
```
TestConductor::TestCase::ExecutionIdleTimeout
```

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

Test Execution Report

After the execution of all test cases, the execution report is written into an HTML file. This file is added to the test package as a controlled file.[6] A report for each test context and test case that has been executed was also created during execution.

If a report file already exists it is overwritten, only the report of the last execution is stored in the model. If a test case or test context or test package 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.



- A double click on the test result "`Result_0.html`" opens the linked test report

[6]Note that with the property TestConductor.Settings.ReportLocation (see page 134) a user can specify a dedicated report location)

Test Package Result

Test Package: TPkg_CashRegister_0

Thu Mar 08 11:31:07 2007

Environment Info	
Test executed on machine:	NBOSC38
Test executed by user:	rsanders
Used OS version:	Windows 2000 / Windows XP
Used Rhapsody version:	Aries, build 805506
Used TestConductor version:	2.0, build 548

Tested Project	
Project:	CashRegister
Active Component:	TCon_CashRegister_0
Active Configuration:	DefaultConfig

Test Package: TPkg_CashRegister_0	Summary: FAILED
Containing Packages:	
Containing Test Contexts:	
TCon_CashRegister	FAILED

Assertion based testing mode

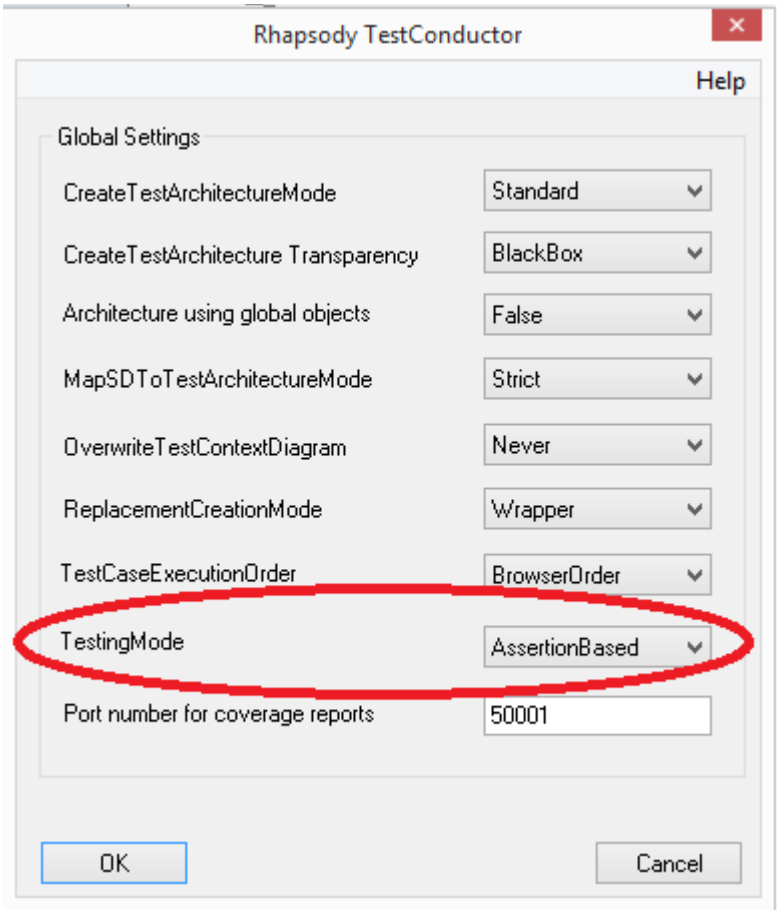
Before Rhapsody 7.6, TestConductor only supports so-called animation based testing mode. In animation based testing mode, the scheduling and arbitration, i.e., the way TestConductor decides whether a test case is passed or failed, is based on animation messages coming from Rhapsody's animation feature. Starting from Rhapsody 7.6, TestConductor also supports so-called assertion based testing mode. In contrast to animation based testing mode, in assertion based testing mode both scheduling and arbitration of test cases is directly controlled by assertions that are compiled into the test executable, i.e., scheduling and arbitration of test cases is independent from Rhapsody's animation feature. Since in assertion based testing mode the test cases are part of the application itself, observation of messages or behavior in the initialization of the application is limited. The test case 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.

In general, both animation based testing and assertion based testing mode provide the same set of features, however, there are still some differences because of the underlying testing approach. In this section, we highlight the characteristics of assertion based testing.

Choosing between testing modes

By default new test architectures 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" on the top level test package and the architecture is created accordingly. For existing test packages that have been created with a Rhapsody version older than 7.6 this property is set to "AnimationBased", i.e. for such test architectures the animation based testing mode is applied and the tests are executed the same way as before, based on Rhapsody animation messages.

The mode for newly created test architectures can be defined in the TestConductor main dialog. Open the TestConductor main dialog by choosing "TestConductor" from the tools menu. In the upcoming dialog, select the testing mode you want TestConductor to apply on newly created test architectures.
This setting does not affect the mode of any existing test architecture in the model. TestConductor supports using animation and assertion based test architectures in the same model but it is not supported to mix these modes in one test architecture. Because of the different structure of the test architectures for each mode, it is not supported to switch an existing test architecture to a different mode by setting the TestingMode property on the test package. For some kinds of animation based test architectures, TestConductor supports an automatic migration of the test architecture to assertion based testing mode. See "Migrating animation based test architecture to assertion based test architecture" on page 100.



Migrating animation based test architecture to assertion based test architecture

There are several differences between an assertion based and an animation based test architecture, so an animation based test architecture cannot be converted into an animation based test architecture just by changing the property "TestConductor.Settings.TestingMode". Instead, it is recommended to create a new test architecture and to create new test cases based on the original ones.

To manually migrate an animation based into an assertion based test architecture, the following approach should be applied:

- Make sure the project property "TestConductor.Settings.TestingMode" is set to "AssertionBased" (see section "Choosing between testing modes" on page 99).
- Create a new test architecture for the class, file or object which was tested by the animation based test architecture.
- Migrate the test cases of the original test architecture one after another. For the different kinds of test cases, the following migration steps should be applied:
  - Code based test cases
    A code based test case can be copied to the new assertion based test architecture. It is recommended to inspect the code of the test case and check for references of test components which might have a different name.

- Flowchart based test cases
  A flowchart based test case can be copied to the new assertion based test architecture. It is recommended to inspect the code of the test case and check for references of test components which might have a different name.

- Statechart based test case
  A statechart based test case should be migrated this way:

  - Create a new test case by applying the helper “Create Statechart TestCase” on the new test context.

  - Select all elements in the new statechart and delete them

  - Open the statechart of the original test case

  - 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 test package 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 test package as the trigger of the transition and remove the line “TCon_<name>_rtc_init(me->itsTCon)” from the Action of the transition.

  - Adjust the last transition in the statechart (from state “final” to the termination state):
    For language C++: In the Action of the transition, change line “itsTCon->rtc_exit()” to “itsTCon->finishTestCase()”.
    For language C: In the Action of the transition, change line “TCon_<name>_rtc_exit(me->itsTCon)” to “Tcon_<name>_finishTestCase(me->itsTCon)”.

- Sequence diagram based test case
  A sequence diagram based test case should be migrated this way:

  - If the old and the new test architecture have similar test components, the test case wizard can be used to create a new test cased based on the test scenario of the old test case. To do this, right click the original test scenario and select “Create TestCase...”. In the dialog, the destination test context can be selected: If the new test context of the assertion based test architecture is listed, select the new test context and confirm the creation of a new test case by clicking the Ok button. The wizard will create a new test case in the animation based test architecture, based on the original test scenario.

  - If the wizard cannot match the test component instances of the animation based test architectures with the test component instances of the assertion based test architecture, the sequence diagram based test cases need to be migrated manually. To do so, create a new new test case by applying the helper “Create SD TestCase” on the new test context. Then add the messages of the original test scenario one after another.

Automatical Migration of animation based TestArchitectures to assertion based Testing mode

When updating a TestContect of an animation based TestArchitecture, TestConductor checks for applicability of automatical migration to assertion based testing mode. Automatical 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, automatical 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. Automatical migration isn't offered to the user for this TestContext again unless property TestConductor.TestContext.MigrateToAssertionBasedMode is checked, i.e.set to 'True'.

In particular SD TestCases may be affected by several limitations of the assertion based TestingMode:

- assertion based execution only supports linearly ordered SDInstances.
- assertion based execution only supports 'driving and monitoring' SDInstances.
- assertion based execution only supports SDTestCases with single SDInstances.
- multiple iteration of SDInstances isn't supported in assertion based execution.
- ordered predecessors aren't supported by assertion based execution.

Potential problems are reported on the console and these migration messages are also recorded in a comment that is stored below the TestContext in the new TestArchitecture obtained by automatical migration.Note, that most TestConductor.TestCase properties aren't regarded in assertion based execution.

Differences between animation and assertion based testing mode

The table below is listing the main differences between the two testing modes.

Animation based testing mode	Assertion based testing mode
Not certified.	Certified for IEC 61508 and derived standards.
No validation suite available.	TestConductor validation suite for on site qualification available.
Based on Rhapsody animation feature: The test architecture needs to be instrumented with animation instrumentation.	No animation instrumentation needed.
Allows white box testing if the SUT is instrumented with animation instrumentation.	Black box testing (optional grey box testing using special TestArchitecture).

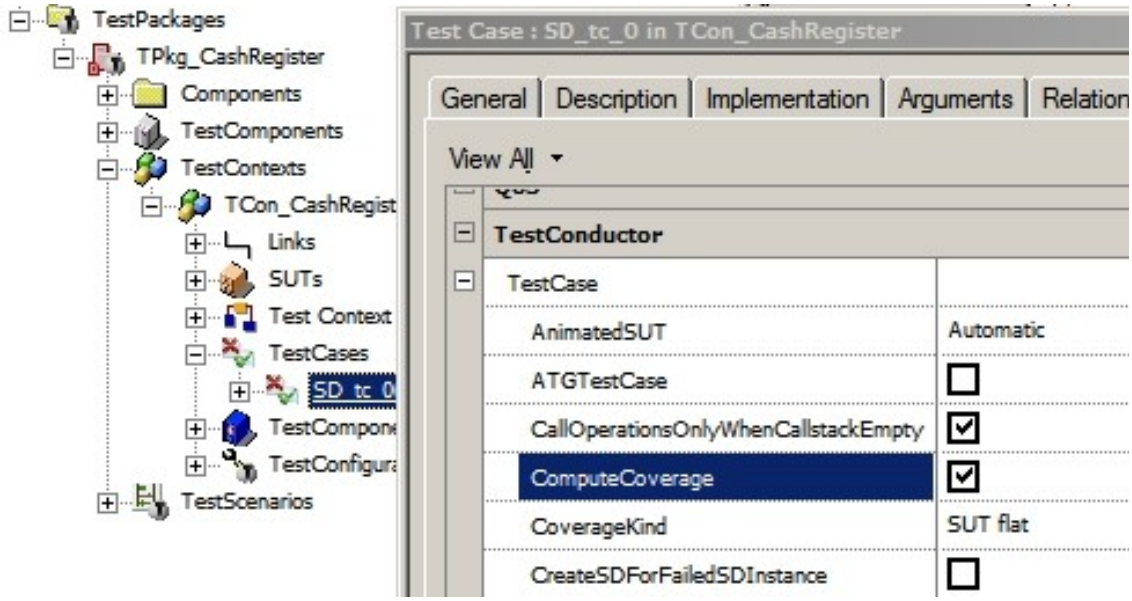
Animation based testing mode	Assertion based testing mode
Serialization and unserialization functions needed to validate and inject message arguments.	Serialization or unserialization functions not needed.
Supports computation of model coverage if the SUT is instrumented with animation instrumentation.	Supports computation of model coverage if the SUT is instrumented with animation instrumentation.
No computation of code coverage (only with third party tools).	Supports computation of code coverage.
Test scheduler and arbiter are in TestConductor tool.	Test scheduler and arbiter are part of the tested application.
SD based TestCases can have multiple SDInstances.	SD based TestCases can have only one SDInstance.
No support of SD operators in SD based TestCases.	Support of SD operators in SD based TestCases (not all).
Available for C++, C, Java, Ada.	Available for C++, C.

Computing Model Coverage during Test Execution

When executing TestCases, i.e., either individual TestCases, a TestContext or a TestPackage, TestConductor provides the possibility to compute which model parts of the SUT are executed during the execution of the TestCases. This information is provided by an HTML report that is created and added to the model after the execution of the test cases. 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 Test Cases

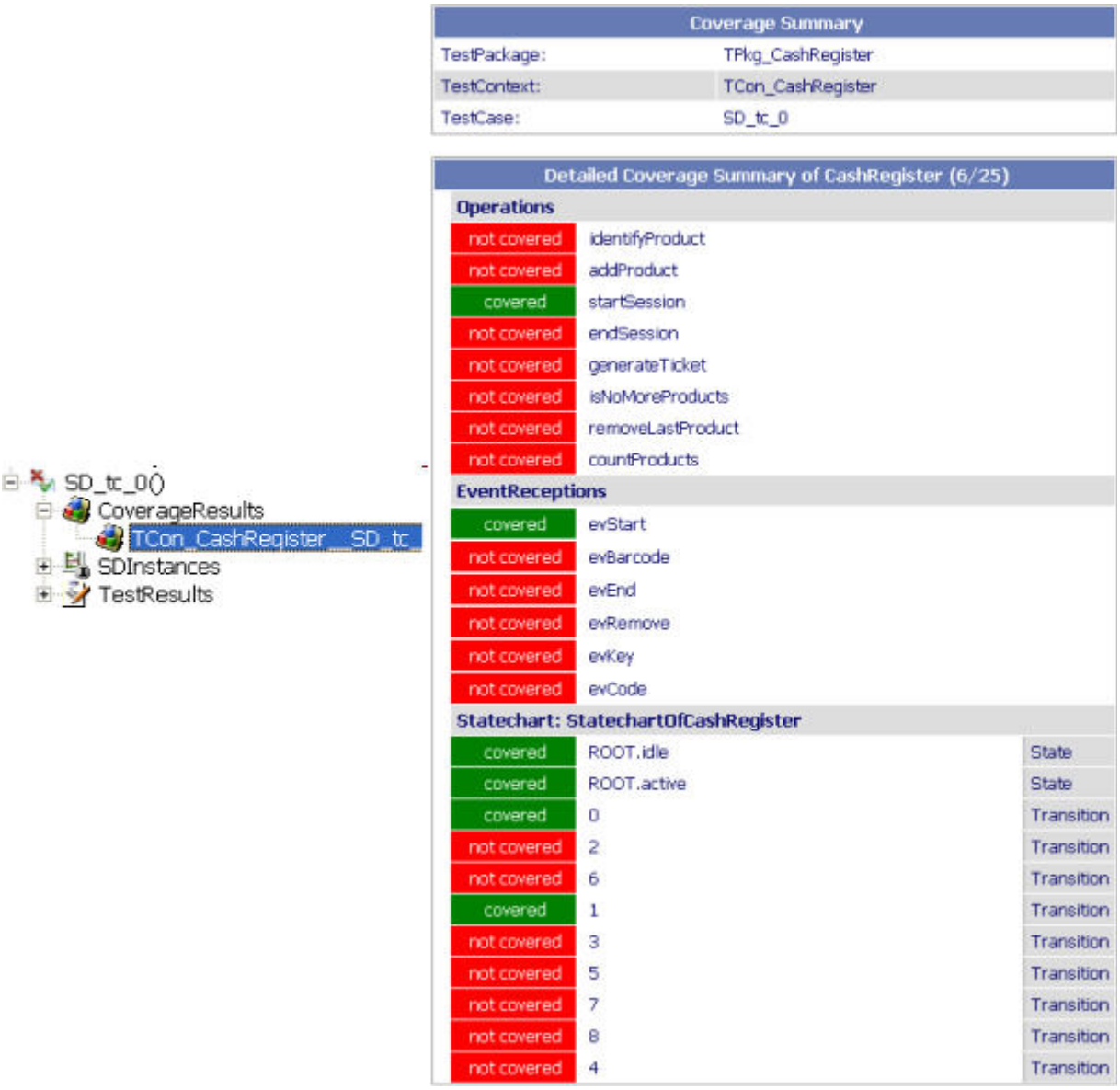
For animation based testing (TestConductor.Settings.TestingMode == AnimationBased), to compute the model coverage of single test cases, switch on the property “TestConductor.TestCase.ComputeCoverage”:



Alternatively, computation of model coverage can be enabled also for animation based testing mode by switching on tag “ComputeModelCoverage” on the code generation configuration used for testing.

Now, each time you execute the test case, in addition to the test case execution report, TestConductor creates a model coverage report and adds it to the model[7]. If a test case is executed for multiple code generation configurations, for each configuration a separate model coverage report is stored in the model.

[7]Note that with the property TestConductor.Settings.ReportLocation (see page 134) a user can specify a dedicated report location)



Coverage Items

Model elements which are subject to the coverage are the *operations*, *event receptions* and elements in *behavior specifications* (statecharts or activities) of the classes for which coverage is measured (for the selection of classes for the coverage measurement see section 'Choosing the Coverage Kind for Model Coverage'). If an operation is specified by a behavior diagram[8], this behavior is considered as well. Of a behavior all vertexes and transitions contained in the behavior are considered. If a coverage item is marked as 'covered' this means that the corresponding code generated for the model element has been traversed during the execution of the test, e.g. an operation has been called or a state in a statechart has been reached[9]. The coverage information is from the model view, there is no information about how much of the user code has been traversed, but only that the model element was used. For a code view with detailed information about the coverage of the generated and the user code you need to use code coverage.

[8]For operations only token oriented activities are allowed.
[9]For some statechart and activity elements which are directly dependent of other elements Rhapsody does not generate animation messages which are used by TestConductor to measure the coverage. For these elements TestConductor applies a set of dependency rules to derive the coverage.

Limitations:

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

Coverage Measurement

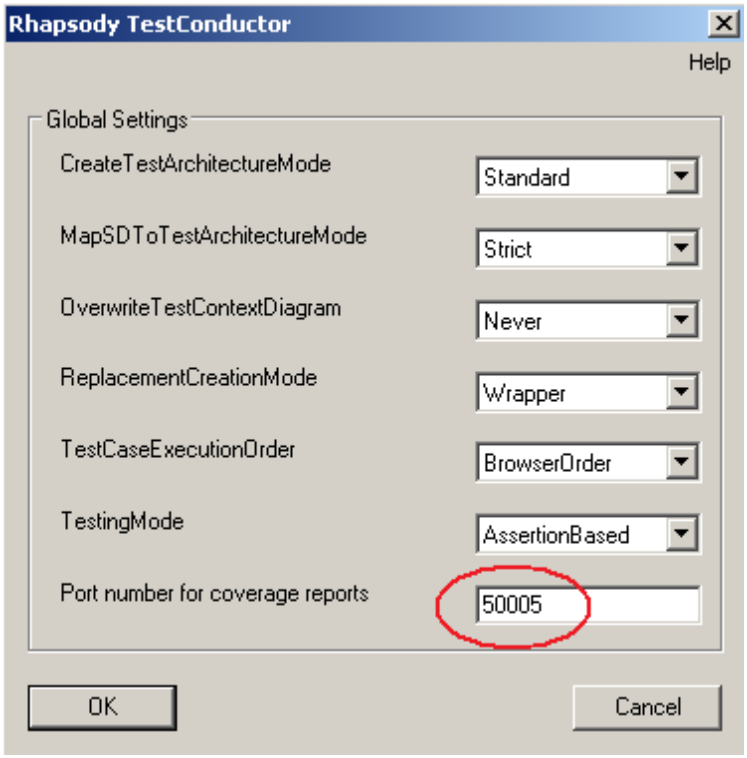
TestConductor uses the Rhapsody animation to determine the coverage of model elements, therefore usage of model coverage requires the 'Instrumentation Mode' of the configuration set to 'Animation'. With this setting the Rhapsody code generation instruments the code with additional animation code, TestConductor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

Traceability of Coverage Items

The html report contains links for the navigation from the report to the Rhapsody model: When clicking on the link of an operation, event, state or transition, the corresponding model element is highlighted in the Rhapsody browser.
Note: This is not supported for Internet Explorer 6, to be able to use this feature, Internet Explorer 7 or higher is needed. Also supported browsers are Firefox 3 and higher, Opera and Chrome. Highlighting model elements will work only if Javascript is enabled in the browser and no popup 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.



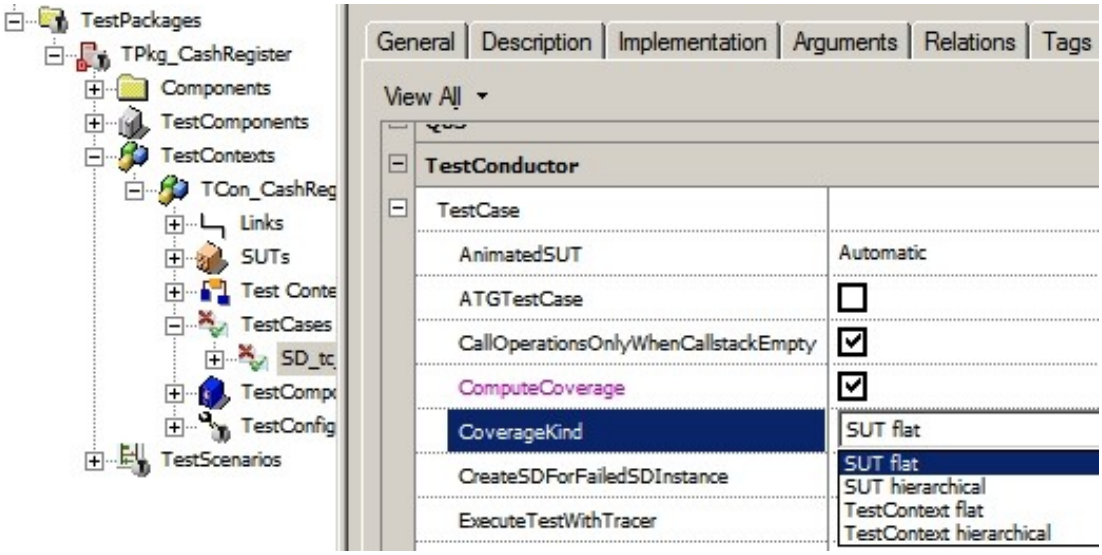
Choosing the Coverage Kind for Model Coverage

TestConductor supports four different kinds of coverage measures, which can be chosen using property TestConductor.TestCase.CoverageKind (if TestConductor.Settings.TestingMode == AnimationBased) or tag “CoverageKind” of the testing configuration (if TestConductor.Settings.TestingMode == AssertionBased)

- SUT flat (Default): Only coverage of the toplevel class of the SUT is measured, i.e. states, transitions, and operations of parts of the SUT are not considered. Coverage of model elements of test components is also not measured.
- SUT hierachical : Coverage of the SUT is measured in a hierarchical manner, i.e. also states, transitions, and operations of parts of the SUT are hierarchically

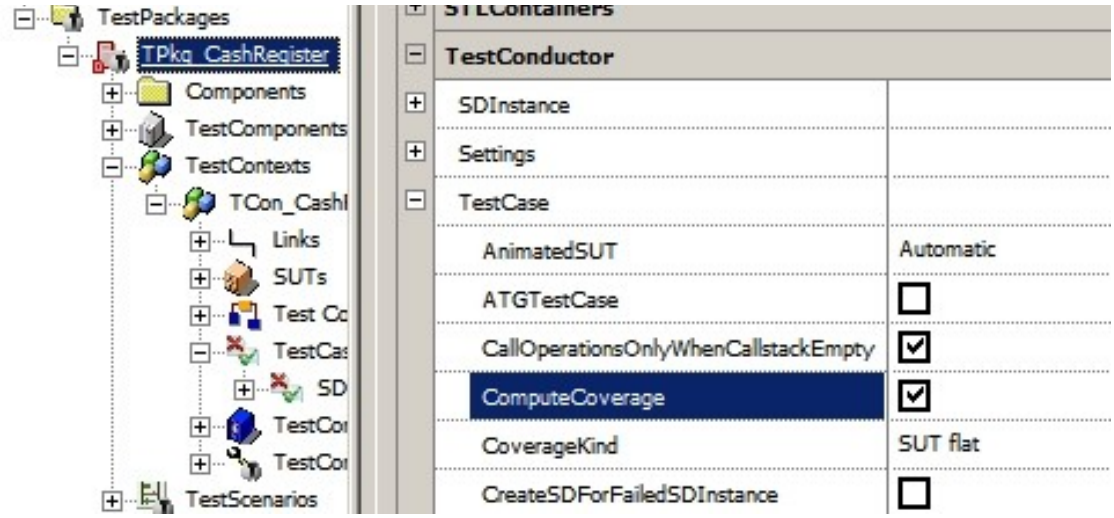
regarded for coverage measure. Coverage of model elements of test components is again not measured.

- TestContext flat: Coverage is measured in terms of all states, transitions, and operations defined at the first decomposition level of the test context, i.e. all states, transitions, and operations of the direct parts of the test context are considered.
- TestContext hierarchical: all states, transitions, and operations in the hierarchal structure of the test context are considered in coverage measure.



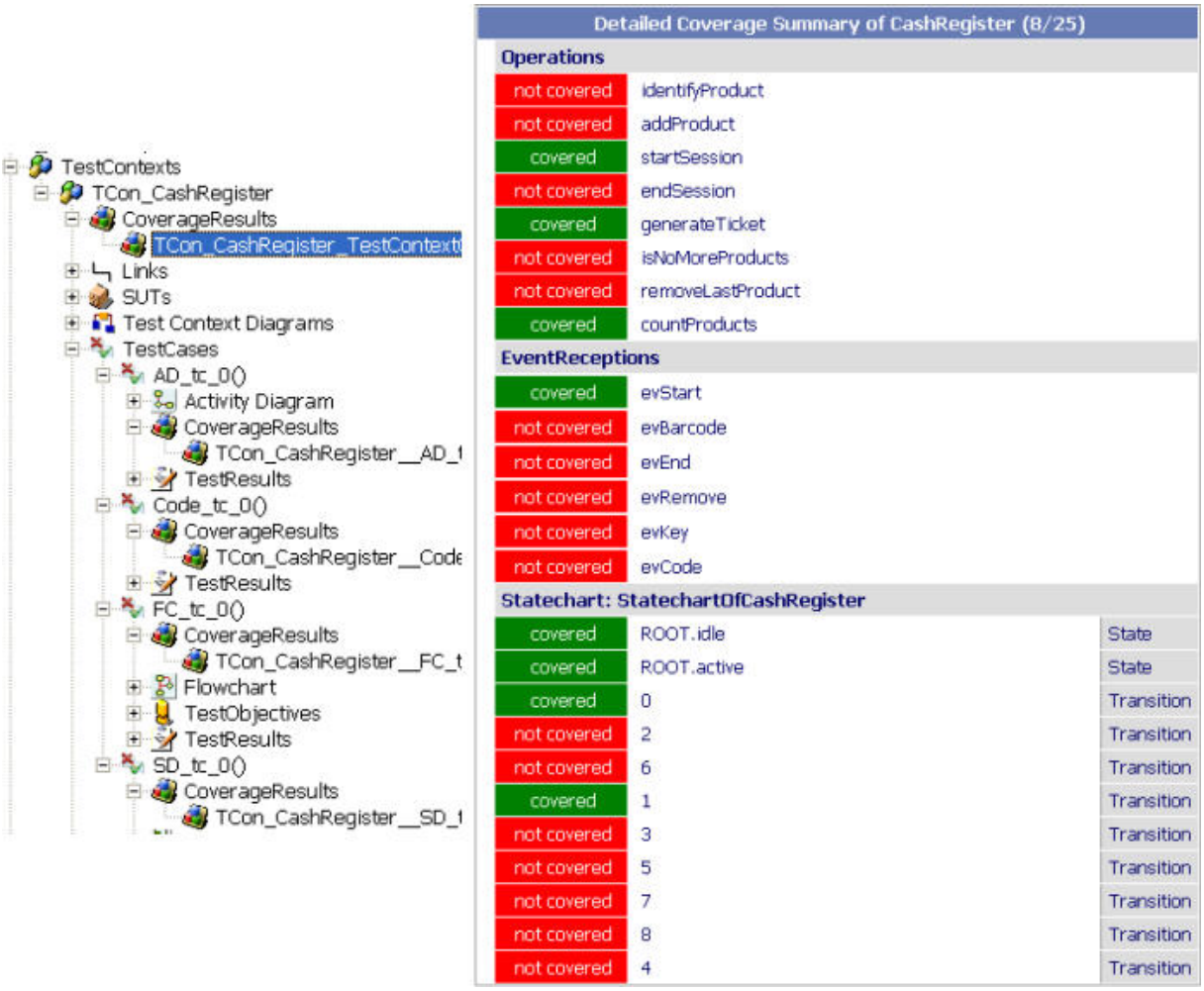
Computing cumulative Model Coverage for TestContexts

To compute the model coverage for TestContexts, for at least one of the TestCases of the TestContext the property “TestConductor.TestCase.ComputeCoverage” must be switched on (if TestConductor.Settings.TestingMode == AnimationBased) or the tag “ComputeModelCoverage” (if TestConductor.Settings.TestingMode == AssertionBased) must be turned on. However, if the property is switched on for more than one test case of the TestContext, TestConductor computes the cumulative coverage of all executed test cases that have switched on this property and stores the result as a coverage report underneath the TestContext. In order to compute the cumulative coverage of all test cases of a TestContext this property has to be switched on for all test cases belonging to the TestContext. A simple way to do it is to set the property directly for the TestPackage that contains the TestContext:



Alternatively, computation of model coverage can be enabled also for animation based testing mode by switching on tag “ComputeModelCoverage” on the code generation configuration used for testing.

Now, when executing the complete TestContext, a coverage report is generated for each of the contained test cases, and a cumulative coverage report is generated for the TestContext. If a test context or test case is executed for multiple code generation configurations, for each configuration a separate model coverage report is stored in the model.



Computing cumulative Model Coverage for TestPackages

Analogously to computing the cumulative coverage of TestContexts, TestConductor also provides the possibility to compute the cumulative coverage of TestPackages. To compute the model coverage for TestPackages, for at least one of the TestCases of the TestPackage the property “TestConductor.TestCase.ComputeCoverage” must be switched on (if TestConductor.Settings.TestingMode == AnimationBased) or the tag “ComputeModelCoverage” (if TestConductor.Settings.TestingMode == AssertionBased) must be turned on for the code generation configurations being used for testing. However, if the property is switched on for more than one test case of the TestPackage, TestConductor computes the cumulative coverage of all executed test cases that have switched on this property and stores the result as a coverage report underneath the TestPackage. In order to compute the cumulative coverage of all test cases of a TestPackage this property has to be switched on for all test cases belonging to the TestPackage. A simple way to do it is to set the property directly for the TestPackage for which the cumulative coverage shall be computed.

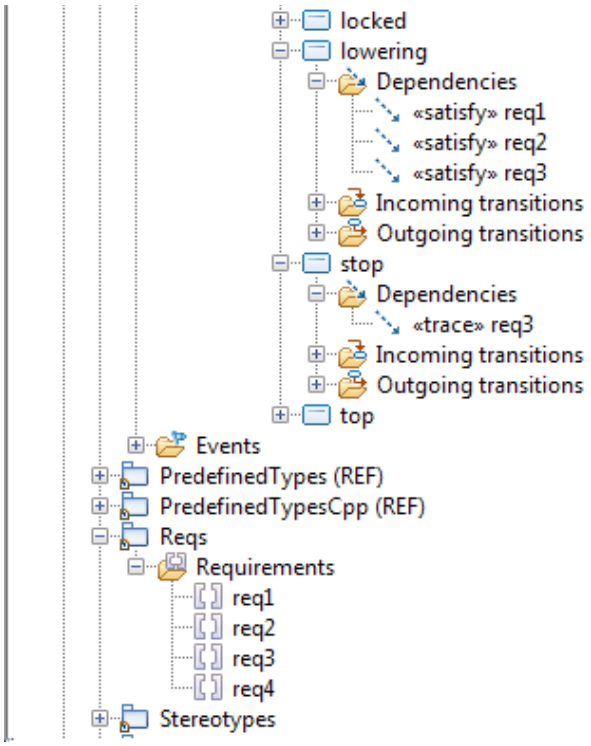
Alternatively, computation of model coverage can be enabled also for animation based testing mode by switching on tag “ComputeModelCoverage” on the code generation configurations used for testing.

Computing Requirement Coverage

Computing Requirement Coverage for Test Cases and TestContexts

Beyond measuring and reporting model element coverage for executed test cases and test contexts, TestConductor offers also the measurement of the dynamic requirement coverage for the executed test cases and test contexts.

Precondition for measuring requirements coverage by individual test cases and test contexts 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.



Note: For the animation based testing mode the stereotype <<AnimationBasedTestingConfiguration>> must be set on the code generation configuration in order to have an access to the tags and properties necessary to enable and customize computation of requirement coverage.

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

The figure above shows <<statisfy>> dependencies from state lowering to requirements req1, req2, and req3 and <<trace>> dependencies from state stop to requirement req3.

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

TestConductor provides also two properties for the user in order to configure the requirement coverage scope for TestConductor. So the user can specify the packages (and their sub-packages), whose requirements shall be regarded at the requirement coverage calculation within the property '`ModelBasedTesting.Settings.Requirement CoverageRequirementsScope`' of the code generation configuration. The setting of multiple packages (and their sub-packages) can be archived via a comma separated list of the fully qualified package paths, e.g. "`RequirementsAnalysisPkg::Requi rementsPkg::SecSysReqs,TestPkg::RequirementsPkg::SecSysTestR eqs`". The second property '`ModelBasedTesting.Settings.Requirement CoverageRegardedTags`' of the same code generation configuration, specifies via a "requirement tag with name and value" those requirements within the pre-selected packages, who shall be considered at the requirements coverage calculation. Again the setting of multiple "requirement tags with name and value" can be archived via a comma separated list, e.g. "`RequirementType=functional,RequirementType= additional`".

TestConductor provides additionally two properties for the user in order to configure the model elements scope for the TestConductor requirement coverage calculation. So the user can specify the packages (and their sub-packages), the classes (blocks) or actors, whose model elements shall be regarded at the requirement coverage calculation within the property '`ModelBasedTesting.Settings.RequirementCoverageModel ElementsScope`' of the code generation configuration. The setting of multiple packages (and their sub-packages), classes (blocks) or actors can be archived via a comma separated list of the fully qualified package, classes (blocks) or actor paths, e.g. "`DesignSynthe sisPkg::SecSysControllerPkg::SecSysController,ActorPkg::Card ReaderEntry`". The second property '`ModelBasedTesting.Settings.Requi rementCoverageExcludedMetaClasses`' of the same code generation configuration, specifies via an "excluded meta classes tag with name and value" those meta classes within the pre-selected packages, classes (blocks) or actors, who shall be excluded from (not considered at) the requirements coverage calculation. Again the setting of multiple "excluded meta classes tags with name and value" can be archived via a comma separated list, e.g. "`Attribute,Class,Event`".

TestConductor distinguishes two kinds of requirement coverage by test cases:

full coverage

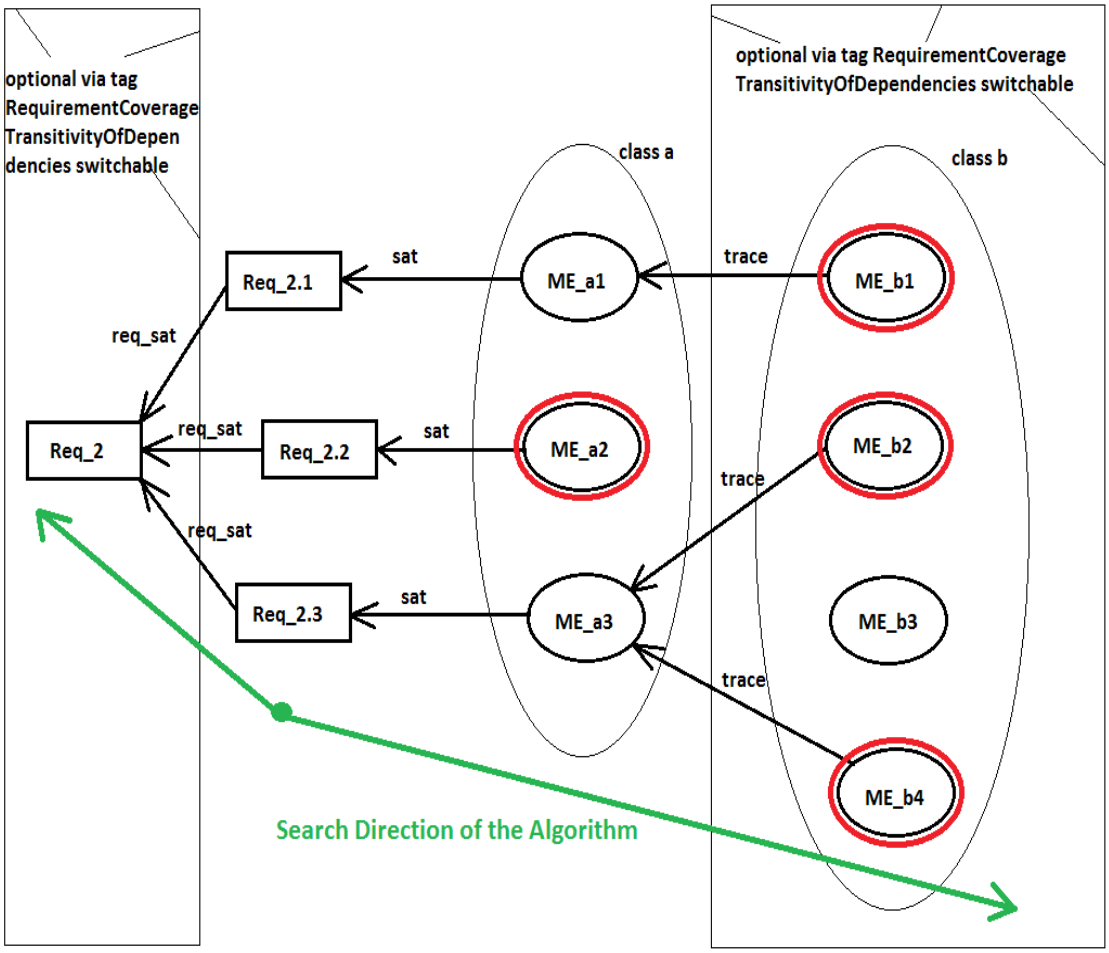
All model elements depending on a particular requirement (w.r.t. specified dependency stereotypes) are covered by a test case or test context. The test case or test context then `fully` covers the requirement - a dependency stereotyped `fully` on the requirement is added to the Requirement Coverage Result Report of the test case or test context.

partial coverage

Not all model elements depending on a particular requirement (w.r.t. specified dependency stereotypes) are covered by a test case or a test context. The test case or test context then only `partially` covers the requirement - a dependency stereotyped `partially` on the requirement is added to the Requirement Coverage Result Report of the test case or test context.

Transitivity of Dependencies (Refinement of model elements and requirements)

Via the TestConductor property "`ModelBasedTesting.Settings.RequirementCoverageTransitivityOfDependencies`" the support for the refinement of model elements and the refinement of requirements (for the TestConductor requirement coverage calculation) can be switched on or off.



The figure above shows an application for the refinement of requirements and model elements. If transitivity of dependencies is switched off, ME_a1 is connected to Req_1.2, ME_a2 is connected to Req_2.2 and ME_a3 is connected to Req_2.3. But if transitivity of dependencies is switched on, the refinements of ME_a1 by ME_b1 and of ME_a3 by ME_b2 and ME_b4 are considered during the requirement coverage calculation. This means, the requirement Req_2.3 for example is only fully covered by a test case or a test context if both model elements ME_b2 and also ME_b4 are covered by this test case or test context (if class B is within the model element scope).
If transitivity of dependencies is switched off the connections between the requirements Req_2.1, Req_2.2 and Req_2.3 to the requirement Req_2 are not considered. But if transitivity of dependencies is switched on the requirement Req_2 is refined by the requirements Req_2.1, Req_2.2 and Req_2.3 and these refinements are considered at the requirement coverage

calculation. This means, the requirement Req_2 is only fully covered by a test case or a test context, if the requirements Req_2.1, Req_2.2 and Req_2.3 are as well fully covered by this test case or test context (if Req_2 is within the requirements scope).

An example explaining the transitivity of dependencies related to the handling of refined model elements: A model element "A1" (class A) has a satisfy dependency to a requirement "req_17". And there is a refinement of the model element "A1", as the two model elements "B1" and "B2" (class B) have both a trace dependency to the model element "A1". And in the same way model element "B1" is refined by the model elements "C1" and "C2" (class C) and model element "B2" is refined by the model elements "C3" and "C4" (class C). If the property "RequirementCoverageTransitivityOfDependencies" is set and "RequirementCoverageModelElementScope" is only set to "class B", then the requirement "req_17" is fully covered by a test case, if the test case covers all of the model elements "B1" and "B2". But if the test case covers only one of the model elements "B1" and "B2", then the requirement "req_17" is only partially covered.

An example explaining the transitivity of dependencies related to the handling of refined requirements: A model element "A1" (class A) has a satisfy dependency to a low level requirement "req_LL_11". And this low level requirement "req_LL_11" has on his part again a satisfy dependency to a high level requirement "req_HL_01". A model element "A2" (class A) has a satisfy dependency to a low level requirement "req_LL_22". And this low level requirement "req_LL_22" has on his part again a satisfy dependency to a high level requirement "req_HL_01". If the property "RequirementCoverageTransitivityOfDepen- dencies" is set and the high level requirement "req_HL_01" is within the requirement scope, then this high level requirement "req_HL_01" is fully covered by a test case, if this test case covers both the low level requirements "req_LL_11" and also "req_LL_22" fully. But if a test case covers either the low level requirement "req_LL_11" or "req_LL_22" only partially, then this high level requirement "req_HL_01" is also only partially covered by this test case.

Computing Code Coverage

Computation of code coverage is supported only for Rhapsody in C++ and Rhapsody in C when using assertion based testing mode.

Integration with CUnit/CppUnit Framework

In the area of testing, CUnit and CppUnit frameworks have become de-facto standards in recent years. Many developers and companies have already organized their testing process using these frameworks. In order to ease migration to a model driven development approach, TestConductor offers a test integration for Rhapsody with the CUnit and CppUnit frameworks.

- CUnit integration has been developed and tested using CUnit-2.1-0.
- CppUnit integration has been developed and tested using cppunit-1.12.1.

This integration is realized using stereotypes defined in the TestingProfile. The stereotypes for CUnit integration are defined in subpackage RTC::TestArchitecture::CUnit, whereas the sterotypes for CppUnit integration are defined in subpackage RTC::TestArchitecture::CppUnit.

Stereotypes for CUnit integration

Stereotype *CUnitContext* can be applied to a class and sets some properties for CUnit testing integration. You can change a test context to CUnitContext – and vice versa - by right-clicking a test context and secting “Change to > CUnitContext”.

Stereotype *CUnitConfig* can be applied to a configuration and provides a set of tags for customization of CUnit testing integration with Rhapsody. CUnitConfig overrides property CG.Configuration.StartFrameworkInMainThread, s.t. the Rhapsody framework ist started in a new thread and control returns to the main thread. Right after starting the framework either a single test case is invoked or all test cases of the test context (only for CUnitContextExecutionKind == NoRestart).

“Update TestCase”, “Update TestContext”, and “Update TestPackage” with respect to a CUnitContext (refering to a confioguration stereotyped <<CUnitConfig>>) will instrument the CUnitContext with a set of operations:

- **int cunit_init()—**CUnit requires an initialization and a cleanup function for each test suite. These functions are provided by TestConductor as prototypes, which can be used to add application or test specific code.
- **int cunit_clean()--**the test suite cleanup function.
- **void cunitmain(char* tc_name)—**the main function for CUnit testing. The function consists of :
    - a framework initialization part
    - a test suite specific part – i.e. a CUnitContext specific part
    - a testoutputter definition part
    - and a execution and result computation part – refered to as tail

  Each of these parts can be customized using a tag of the <<CUnitConfig>> configuration.
- **<testcontext-type>* setTestContext(<testcontext-type>* context)—**Since test cases may not have arguments in the CUnit framework,, they can not be invoked with the ‘me’-pointer by the test context. Hence, a static variable is required, that allows access to the test context data structure within test cases. Test cases can get access to this data structure using the test context function ‘theTestContext()’. Function ‘setTestContext()’ sets a static pointer variable, which then can be returned by ‘theTestContext()’.
- **<testcontext-type>* theTestContext()—** see above**.**
- **Init()—**initializer that, in particular, invokes ‘setTestContext()’ with the ‘me’-pointer in order to enable access to the test context data structure from within test cases (see above).

The customization tags of stereotype CUnitConfig are:

- **CUnitContextExecutionKind--** Possible values: ‘RestartExecutable’,’NoRestart’. This tag defines whether the application is restarted for each testcase, or all test cases are executed within a single invokation of the application. Default is ‘RestartExecutable’.
- **CUnitIncludePath—**defines the path to the headers of the CUnit framework. For path definition, a symbolic variable $CUNITINSTALLDIR can be used. This symbolic variable is textually substituted by the contents of tag CUnitInstallDir

upon “Update TestCase”, “UpdateTestContext”, and “Update TestPackage”, respectively. Default: “$CUNITINSTALLDIR/CUnit/Headers”.

- **CUnitInstallDir—**the full path to the installation directory of the CUnit framework. For definition of the path, envronment variables, e.g. “$(CUNITHOME)” can be used . Default: “$(CUNITHOME)”.
- **CUnitLibPath—**the full path to the CUnit framework library file. Default: “$CUNITINSTALLDIR/CUnit/lib/CUnit.lib”.
- **CUnitMainInit—** the initialization part of the cunitmain() function that will be generated by “Update TestCase”, “Update TestContext”, and “Update TestPackage”, respectively. For the default, please consult the TestingProfile.
- **CUnitMainOutputter—** test outputter specific initializations**.** Default: “$RTCAUTOGENERATE”. If CUnitMainOutputter contains exactly this string, TestConductor will automatically generate the respective code according to the chosen output format.
- **CUnitMainTail—** defines the execution and result computation part of ‘cunitmain()’. For the default, please consult the TestingProfile.
- **CUnitReportKind—**possible values: ‘xml’, ‘html’, ‘text’. This tag defines the result report format. Default: ‘html’
- **InvokeExecutable—**the content of this tag will be written to property C_CG.Configuration.<activeEnvironment> and defines how the application will be invoked.
  Default: “$executable $TestCase”, where “$TestCase” will be textually substituted by the “Update ...” functionality with the name of the selected test case or “all”, if a test context is going to be executed.
- **PostFrameworkThreadSegment—** the contents of this tag will be written to property CG.Configuration.PostFrameworkThreadSegment. Using this tag it can be customized how ‘cunitmain()’ will b invoked. Default: “char* tcname = argv[1]; cunitmain(tcname);”
- **ReportFilename—** the filename prefix of the report generated by CUnit. Default: “$CONFIGDIR/report”, where “$CONFIGDIR” is a symbolic variable denoting the code generation configuration refered to by the test context. “$CONFIGDIR” will be textually replaced by the “Update ...” functionality.
- **ResultFilename—** the filename for the overall ‘pass/fail’ result. A CUnit test case execution passes, iff all executed assertions pass; a CUnitContext execution passes, iff all test cases pass; a TestPackage passes, iff all CUnitContexts pass.
  Default : “$CONFIGDIR/result.txt”
- **XSLTFile---** full path to the xslt file using which a html report can be generated from a CUnit xml report.
  Default : “$CUNITINSTALLDIR/Share/CUnit-Run.xsl”

Stereotypes for CppUnit integration

Stereotype *CppUnitContext* can be applied to a class and sets some properties for CppUnit testing integration. You can change a test context to CppUnitContext – and vice versa - by right-clicking a test context and secting “Change to > CppUnitContext”.

Stereotype *CppUnitConfig* can be applied to a configuration and provides a set of tags for customization of CppUnit testing integration with Rhapsody. CppUnitConfig overrides

property CG.Configuration.StartFrameworkInMainThread, s.t. the Rhapsody framework ist started in a new thread and control returns to the main thread. Right after starting the framework either a single test case is invoked or all test cases of the test context (only for CppUnitContextExecutionKind == NoRestart).

“Update TestCase”, “Update TestContext”, and “Update TestPackage” with respect to a CppUnitContext (refering to a confioguration stereotyped <<CppUnitConfig>>) will instrument the CppUnitContext with a set of operations:

- **void setUp()—**CppUnit requires an initialization and a cleanup function for each test case/test suite. These functions are provided by TestConductor as prototypes, which can be used to add application or test specific code.
- **void tearDown()--**the test suite cleanup function.
- **void cppunitmain(char* tc_name)—**the main function for CppUnit testing. The function consists of :
    - a framework initialization part
    - a test suite specific part – i.e. a CppUnitContext specific part
    - a testoutputter definition part
    - and a execution and result computation part – refered to as tail

  Each of these parts can be customized using a tag of the <<CUnitConfig>> configuration.

The customization tags of stereotype CppUnitConfig are:

- **CppUnitContextExecutionKind--** Possible values: ‘RestartExecutable’,’NoRestart’. This tag defines whether the application is restarted for each testcase, or all test cases are executed within a single invokation of the application. Default is ‘RestartExecutable’.
- **CppUnitContextExecutionKindReuseTestFixtureforNoRestart** – Boolean (default False) – using this tag, the user can specify whether the TestFixture shall be reused for all test cases if CppUnitContextExecutionKind is 'NoRestart' or if a new TestFixture will be created for each test case.
- **CppUnitIncludePath—**defines the path to the headers of the CppUnit framework. For path definition, a symbolic variable $CPPUNITINSTALLDIR can be used. This symbolic variable is textually substituted by the contents of tag CppUnitInstallDir upon “Update TestCase”, “UpdateTestContext”, and “Update TestPackage”, respectively. Default: “$CPPUNITINSTALLDIR/include”.
- **CppUnitInstallDir—**the full path to the installation directory of the CppUnit framework. For definition of the path, envronment variables, e.g. “$ (CPPUNITHOME)” can be used . Default: “$(CPPUNITHOME)”.
- **CppUnitLibPath—**the full path to the CppUnit framework library file. Default: “$CPPUNITINSTALLDIR/lib/CppUnit.lib”.
- **CppUnitMainInit—** the initialization part of the cppunitmain() function that will be generated by “Update TestCase”, “Update TestContext”, and “Update TestPackage”, respectively. For the default, please consult the TestingProfile.
- **CppUnitMainOutputter— —** test outputter specific initializations**.** Default: “$RTCAUTOGENERATE”. If CUnitMainOutputter contains exactly this string, TestConductor will automatically generate the respective code according to the chosen output format.

- **CppUnitMainTail—** defines the execution and result computation part of ‘cppunitmain()’. For the default, please consult the TestingProfile.
- **CppUnitReportKind—**possible values: ‘xml’, ‘html’, ‘text’,’compilertext’. This tag defines the result report format. Default: ‘html’
- **InvokeExecutable—**the content of this tag will be written to property CPP_CG.Configuration.<activeEnvironment> and defines how the application will be invoked.
  Default: “$executable $TestCase”, where “$TestCase” will be textually substituted by the “Update ...” functionality with the name of the selected test case or “all”, if a test context is going to be executed.
- **PostFrameworkThreadSegment—** the contents of this tag will be written to property CG.Configuration.PostFrameworkThreadSegment. Using this tag it can be customized how ‘cunitmain()’ will b invoked.
  Default: “p_$TestContext->cppunitmain(argv[1]);”, where the term “$TestContext” will be textually substituted by TestConductor upon “Update ...”.
- **ReportFilename—** the filename prefix of the report generated by CppUnit.
  Default: “$CONFIGDIR/report”, where “$CONFIGDIR” is a symbolic variable denoting the code generation configuration refered to by the test context. “$CONFIGDIR” will be textually replaced by the “Update ...” functionality.
- **ResultFilename—** the filename for the overall ‘pass/fail’ result. A CppUnit test case execution passes, iff all executed assertions pass; a CppUnitContext execution passes, iff all test cases pass; a TestPackage passes, iff all CppUnitContexts pass.
  Default : “$CONFIGDIR/result.txt”
- **XSLTFile---** full path to the xslt file using which a html report can be generated from a CppUnit xml report.
  Default : “$CPPUNITINSTALLDIR/contrib/xml-xsl/report.xsl”

Test Definition for CUnit/CppUnit

Code and flow chart test cases can be used very similar to their normal usage. Instead of the RTC_ASSERT macros, for CUnit and CppUnit, CU_ASSERT macros and CPPUNIT_ASSERT macros, respectively, are used.

For CUnit also statechart test cases can be used similarly to their normal usage with TestConductor, except for using CU_ASSERT macros instead of RTC_ASSERT macros.

For CppUnit, usage of statechart test cases requires some manual adaptions of the test context and the statechart defining the test. The necessary adaptions are explained below. We recommend using code and flow chart test cases also for testing reactive behavior (cf. Testing reactive behavior with Code Test Cases, Testing reactive behavior with Flow Chart Test Cases on page 50 pp.).

Both, CUnit integration as well as CppUnit integration do currently not support SD test cases.

Using Statechart Test Cases with CppUnit

In the CppUnit framework assertions like CPPUNIT_ASSERT are realized by throwing an exception, when an assertion fails. This exception is caught by the framework and the failed assertion is reported. The entire mechanism relies on the assumption that the test case is executed in the same thread as the framework. CppUnit integration with TestConductor utilizes a test context as test fixture, i.e. the CppUnit framework is executed in the thread of the CppUnitContext. Statechart test cases are realized using a

separate test component owning the statechart, s.t. the statechart is exceuted in the thread of the test component. Since these threads are in general not the same, it is necessary to catch exceptions within the statechart and add failures to the testresult maintained by the CppUnitContext.

Necessary modifications for statechart test cases with CppUnit:

1. Add public attributes

   - CppUnit::TestSuite* suiteOfTests
   - CppUnit::TestResult* theTestResult

   to CppUnitContext

2. Overwrite tag CppUnitMainInit:

```
CPPUNIT_NS::TestResult testresult;
CPPUNIT_NS::TestResultCollector collectedresults;
testresult.addListener(&collectedresults);
std::ofstream outfile;

// Original: local variable
/* CppUnit::TestSuite *suiteOfTests = new
                  CppUnit::TestSuite("$TestContext");*/

//NEW: use CppUnitContext attribute
this->suiteOfTests = new
           CppUnit::TestSuite("$TestContext");

CPPUNIT_NS::TestRunner *testrunner = new
CPPUNIT_NS::TestRunner();

//NEW: initialize attribute of CppUnitContext
theTestResult = &testresult;
```

3. add “cppunit/TestResult.h” to property CPP_CG.Class.ImpIncludes of test component refered to by <<StatechartTestCase>> dependency of statechart test case

4. Instead of simply using e.g.

```
CPPUNIT_ASSERT(
 itsTCon->getItsCalculator()->get_result_op()==42),
```

   in a transition action, you now should write:

```
CPPUNIT_NS::Test* current_tcase = 0;
CppUnitVector<CPPUNIT_NS::Test*>& alltests =
            (CppUnitVector<CPPUNIT_NS::Test*>&)
 (itsTCon->suiteOfTests->getTests());
CppUnitVector<CPPUNIT_NS::Test*>::iterator it =
alltests.begin();
while (it != alltests.end()) {
   if((*it)->getName()=="SC_tc_0") {
        current_tcase = *it;
   }
```

```
    ++it;
}
try {
    CPPUNIT_ASSERT(
        itsTCon->getItsCalculator()->get_result_op()==42);
}
catch (CPPUNIT_NS::Exception e) {
        itsTCon->getTheTestResult()->addFailure(
            current_tcase,
            new CPPUNIT_NS::Exception(e));
}
```

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 test case, a test context or a test package.

TestConductor logs in the file "cl.log" in the project folder the command line actions together with the result[10] (SUCCEDED, FAILED or ERROR[11] for actions, TRUE, FALSE or ERROR for queries).

Note: -cmd=save needs to be defined in order to permanently actualize 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\CppTes`

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

```
tActions.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"`

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 test case, a test context or a test package.
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 “Test Case Execution”, ”Test Context Execution” and “Test Package Execution”.

Test Case Execution on Targets

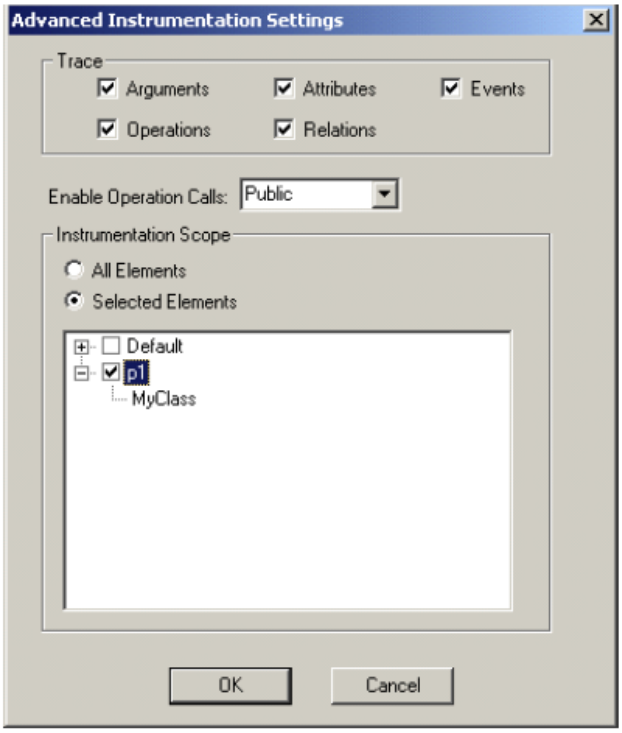
In addition to executing test cases on the host environment, test cases can also be executed on the target environment. The necessary steps are target environment specific and are further described in the following documents:

- Testing_with_RTC_on_a_Linux_Target.pdf (Linux)
- Testing_with_RTC_on_a_VxWorks_Target.pdf (VxWorks)
- Testing with TestConductor on a small target.pdf (generic environment)

Driving Operations Calls

Driving Operation Calls

To be able to call operation calls from the environment in TestConductor, we have to set the **Enable Operation Calls** option in the dialog **Advanced Instrumentation Settings** as *Public*, *Protected* or *All* and recompile/rebuild the model.



This setting controls the property `CG:Operation:AnimAllowInvocation`. Following are the details of the options that can be used:

- **None (Default)**—do not enable calls
- **Public**—enable calls if operation is public
- **Protected**—enable calls if operation is public or protected
- **All**—enable calls in all cases

Test Management

TestConductor is a fully integrated add-on solution for Rhapsody. All relevant test data like the test architecture, test cases and their test scenarios, 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

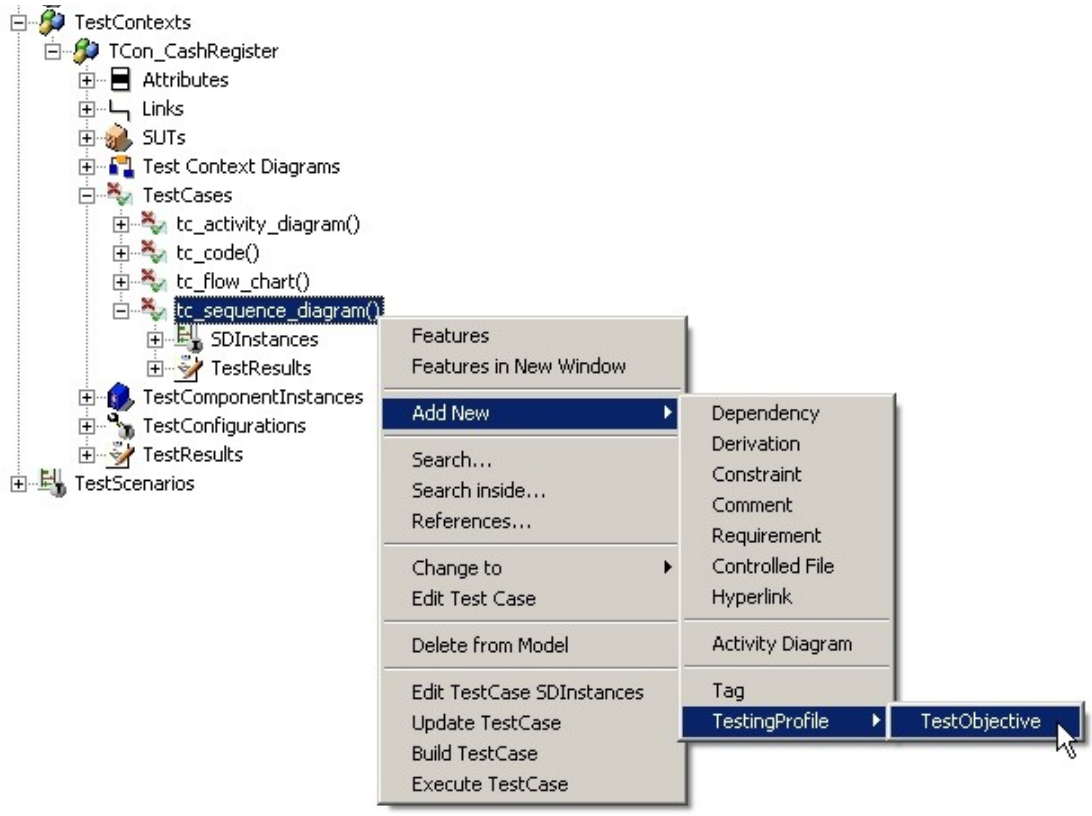
With this tight integration you have all the possibilities you already know from other elements like classes, package and so on, e.g.:

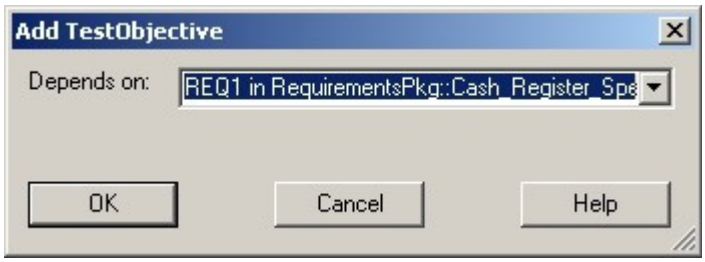
- Copy, paste, delete
- Create units for test components, test context, SUT and test component instances
- Load / unload test packages, test components, test context, SUT and test component instances
- Requirements management
- Configuration management
- Documentation

Linking Test Case to Requirements

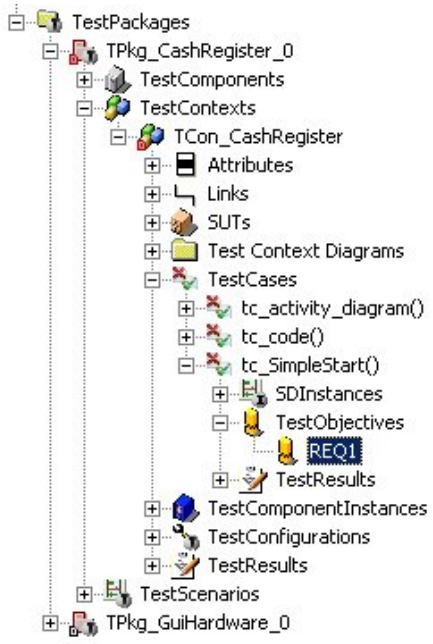
Test cases can be linked to their requirements which are defined in the model. This can be done by using *test objectives (TestObjective)* to link model elements to the related requirements.

- Add a new test objective to the test case “tc_SimpleStart” and select the requirement from the listed model elements.



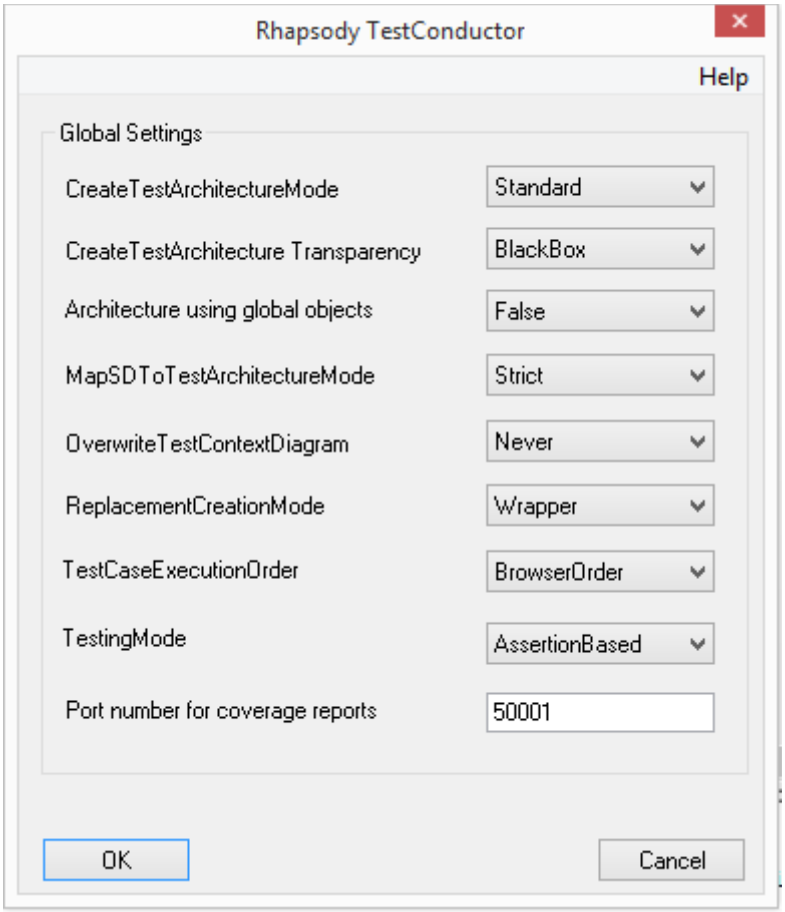


The result is a new test objective `REQ1` as an element of test case “tc_SimpleStart” which is linked to its requirement `REQ1`.



TestConductor Dialog

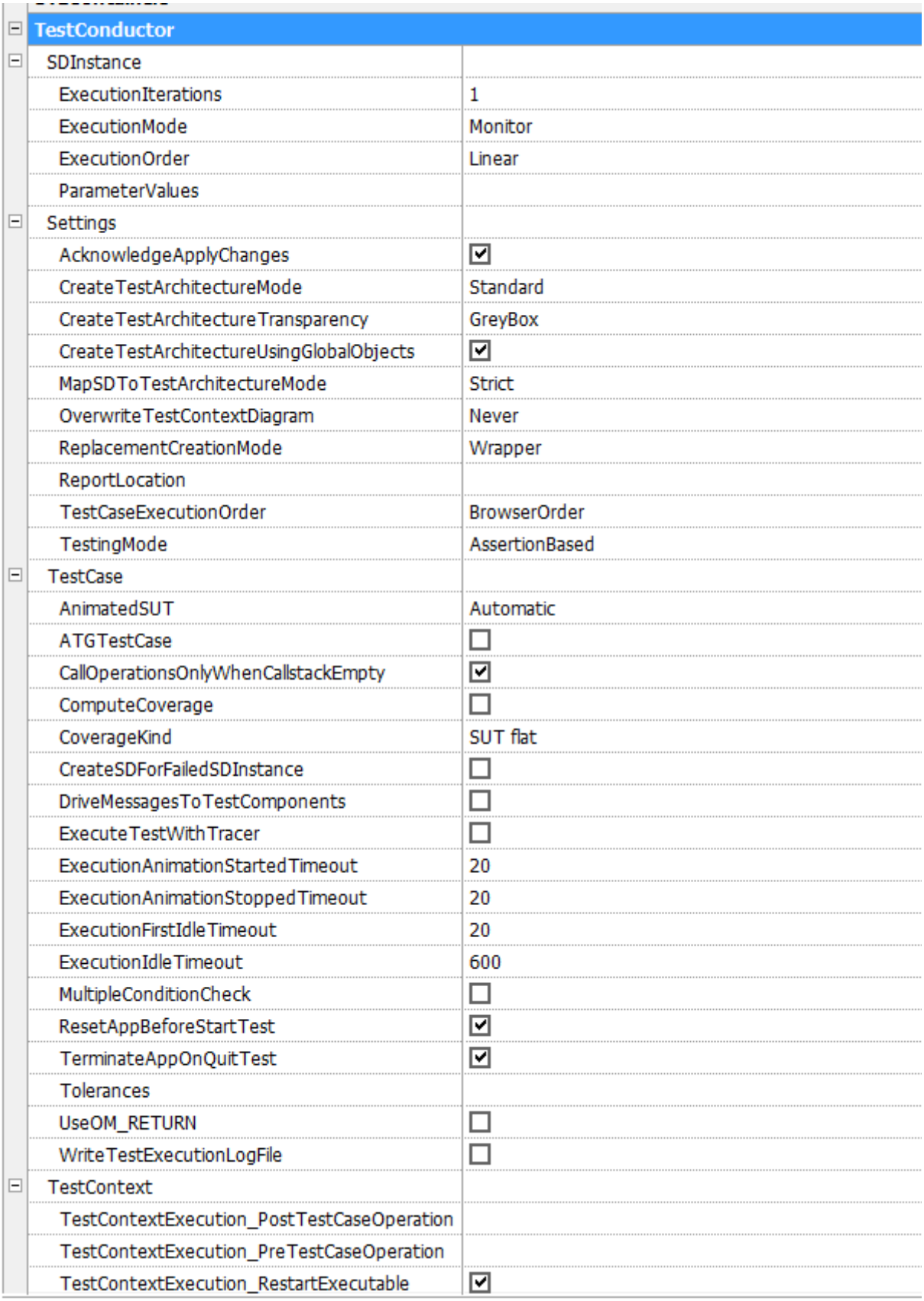
The TestConductor main dialog provides some global TestConductor settings and help functions by selecting **Tools > TestConductor** from the Rhapsody tools menu:



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

TestConductor Settings

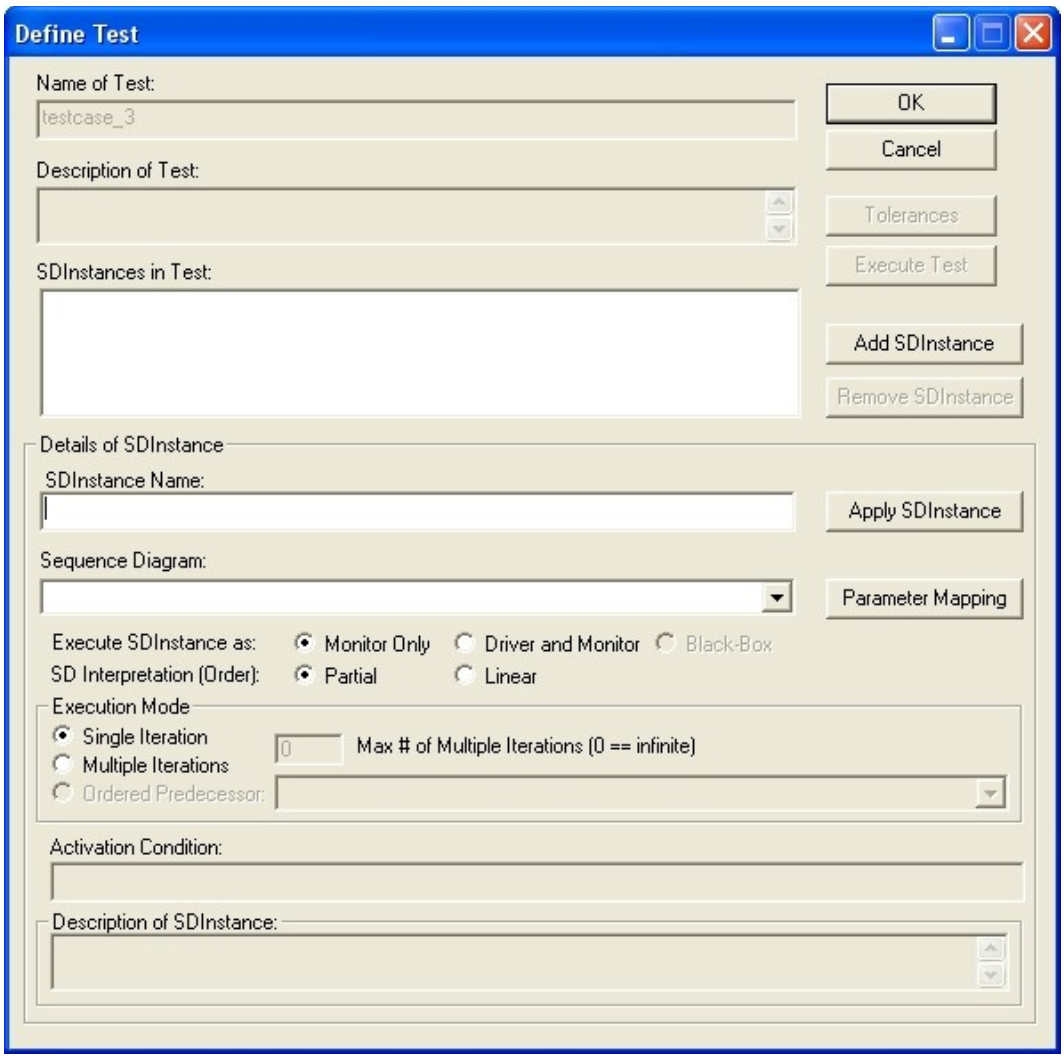
TestConductor provides a range of global and also test case specific settings. The settings are in most cases stored as properties in the model.



TestConductor	
SDInstance	
ExecutionIterations	1
ExecutionMode	Monitor
ExecutionOrder	Linear
ParameterValues	
Settings	
AcknowledgeApplyChanges	☑
CreateTestArchitectureMode	Standard
CreateTestArchitectureTransparency	GreyBox
CreateTestArchitectureUsingGlobalObjects	☑
MapSDToTestArchitectureMode	Strict
OverwriteTestContextDiagram	Never
ReplacementCreationMode	Wrapper
ReportLocation	
TestCaseExecutionOrder	BrowserOrder
TestingMode	AssertionBased
TestCase	
AnimatedSUT	Automatic
ATGTestCase	☐
CallOperationsOnlyWhenCallstackEmpty	☑
ComputeCoverage	☐
CoverageKind	SUT flat
CreateSDForFailedSDInstance	☐
DriveMessagesToTestComponents	☐
ExecuteTestWithTracer	☐
ExecutionAnimationStartedTimeout	20
ExecutionAnimationStoppedTimeout	20
ExecutionFirstIdleTimeout	20
ExecutionIdleTimeout	600
MultipleConditionCheck	☐
ResetAppBeforeStartTest	☑
TerminateAppOnQuitTest	☑
Tolerances	
UseOM_RETURN	☐
WriteTestExecutionLogFile	☐
TestContext	
TestContextExecution_PostTestCaseOperation	
TestContextExecution_PreTestCaseOperation	
TestContextExecution_RestartExecutable	☑

Sequence Diagram Properties

TestConductor provides settings concerning the usage and interpretation of sequence diagrams during test case execution. All following properties are the settings for the dialog **Define Test**:



These settings have to be done via properties on *SDInstance* level. Open the **Feature** dialog of a sequence diagram instance, select the **Properties** tab, switch in the dropdown combo box **View** to *All* and navigate to the metaclass `TestConductor::SDInstance`

SDInstance	
ExecutionIterations	1
ExecutionMode	Monitor
ExecutionOrder	Linear
ParameterValues	

`TestConductor::SDInstance::ExecutionIterations`

The required number of run-time instances can be set to multiple iterations with a concrete number.

Note: This property should not be set directly. Please use the **Multiple Iterations** setting in the **Define Test** dialog.

`TestConductor::SDInstance::ExecutionMode`

Driver invokes automatic driving of model execution after the test has been activated. TestConductor automatically injects events into the running Rhapsody model according to the specified sequence diagram. **Monitor** invokes manual driving of model execution. This means that, during test execution, you must inject input events manually using the Rhapsody animation tool or the project GUI (when available). TestConductor monitors the reception of these events and internal messages between system objects. **Blackbox** considers only those messages that originate at the system border (to be driven by TestConductor) or that go to the system border (to be monitored by TestConductor).

Note: This property should not be set directly. Please use the corresponding **Execute SDInstance as:** setting in the **Define Test** dialog.

`TestConductor::SDInstance::ExecutionOrder`

Linear—specifies that TestConductor should monitor the sequence diagram under test assuming that all events and messages are arranged in a strict sequence. The vertical drawing order of messages in sequence diagrams is used to compute an absolute sequence of events and messages (each message in the in sequence diagram has a unique predecessor and successor). **Partial**—specifies that TestConductor should monitor only the order of events located on the same line (instance line or message arrow).

Note: This property should not be set directly. Please use the corresponding **SD Interpretation (Order):** setting in the **Define Test** dialog.

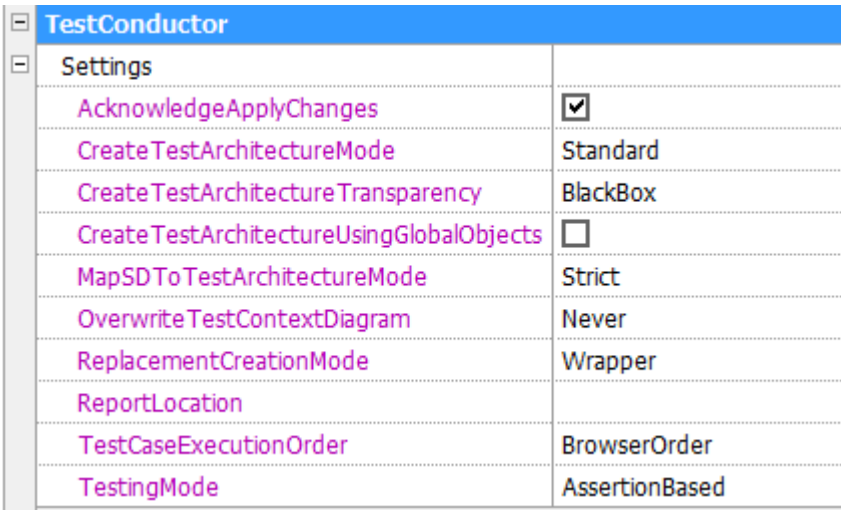
`TestConductor::SDInstance::ParameterValues`

For a parameterized Rhapsody sequence diagram, map its parameters to concrete values.

Note: This property shall not be set directly. Please use the button **Parameter Mapping** in the **Define Test** dialog.

General Properties

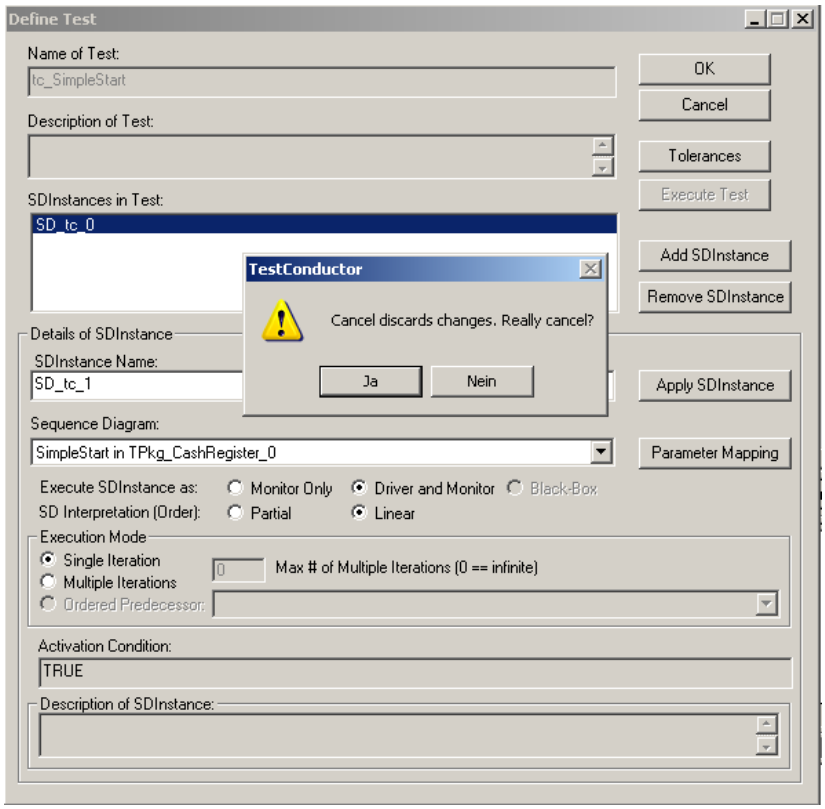
TestConductor provides some general settings that change the general behavior of TestConductor. These settings have to be done via properties on test package level. Open the **Feature** dialog of a test package, select the **Properties** tab, switch in the dropdown combo box **View** to *All* and navigate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

By default this property is switched on.



`TestConductor::Settings::CreateTestArchitectureMode`

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

By default this property has the value “Standard”.

`TestConductor::Settings::CreateTestArchitectureTransparency`

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

`TestConductor::Settings::CreateTestArchitectureUsingGlobalObjects`

Since Rhapsody 8.1.4, TestArchitecture creation can optionally use global objects instead of parts for SUT classes and TestComponent instances. 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 obejcts and singleton objects in test component roles. Thus, it is recommended to use global object instantiation if implicit objects or singleton objects are involved in the testing process.

`TestConductor::Settings::MapSDToTestArchitectureMode`

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

`TestConductor::Settings::overwriteTestContextDiagram`

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

By default this property has the value “Never”.

`TestConductor::Settings::ReportLocation`

With this property[12] TestConductor can be instructed to store test reports and results not in the default location directly underneath the test element (TestPackage, TestContext, TestCase) but at a location chosen by the user. The location has to be a (test-) package, which will be created if not existing yet. For nested packages the qualified name has to be specified using the delimiter '::' (e.g. “MyResults::Results_MR1”).

Affected by this property are Test Execution Results, Model Coverage Results, Requirement Coverage Results and Code Coverage Results. Underneath the test element a hyperlink will be created[13] targeting the actual result. If the property expression can not be

[12]Property will be evaluated not only on project but also also on package level.
[13]Hyperlink will be created only for test elements which can be written.

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:[14]

$TESTPACKAGENAME: Will be substituted by the name of the TestPackage[15] of the executed element.

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

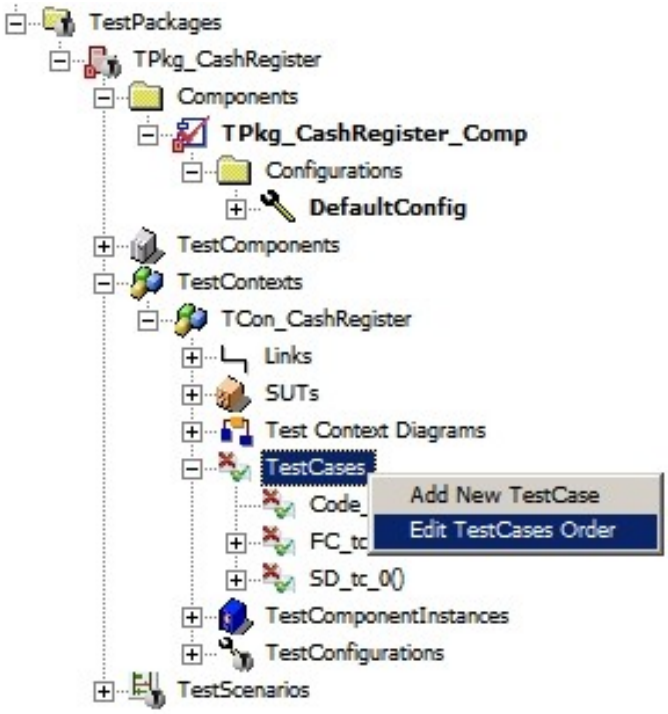
$TESTCASENAME: Will be substituted by the name of the executed TestCase. Will be ignored for TestPackage and TestContext results.

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

`TestConductor::Settings::TestCaseExecutionOrder`

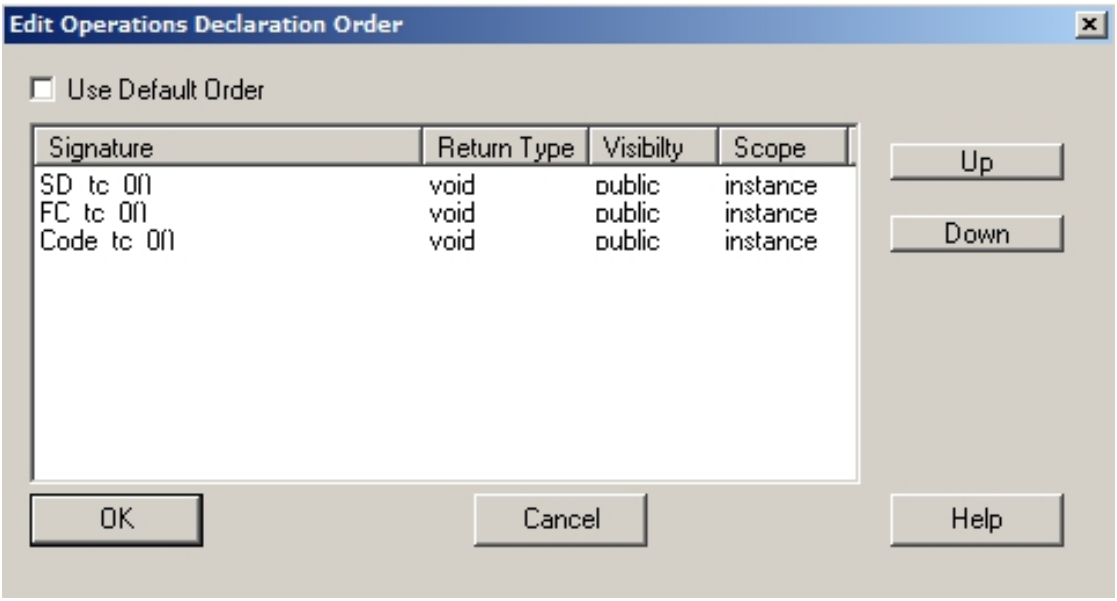
This property controls the execution order of TestCases when executing a TestContext. Possible values are “BrowserOrder” and “DeclarationOrder” , where “BrowserOrder” defines that TestCases areb executed in the same order as they are displayed in the browser. “DeclarationOrder” defines execution in a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

By default this property has the value “BrowserOrder”.



[14]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”)
[15]The outer TestPackage in assertion based mode

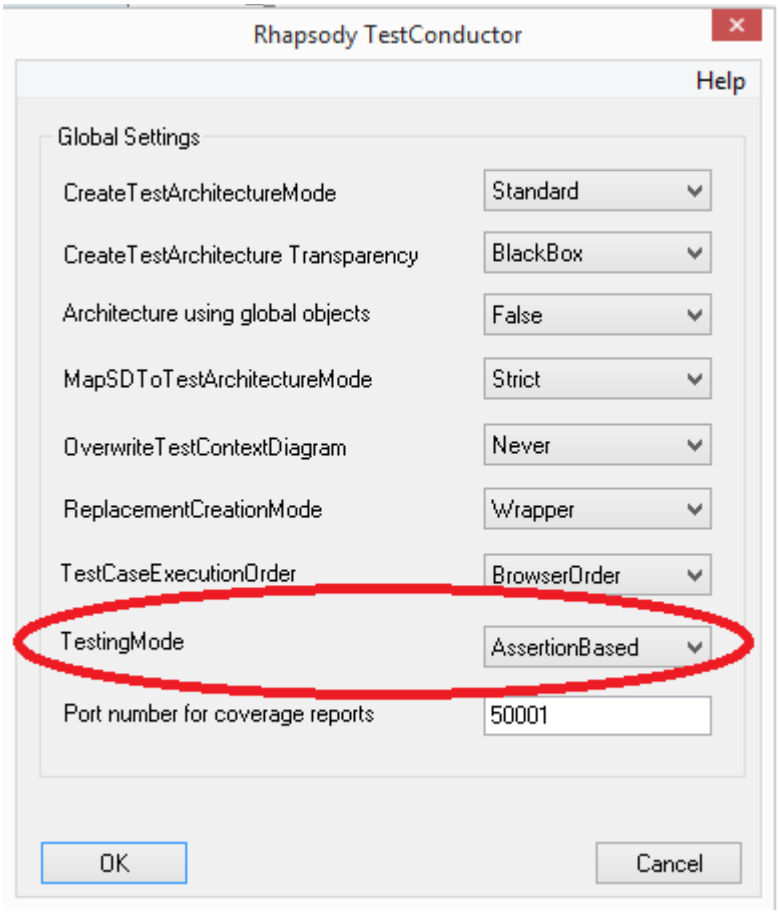
“Edit TestCases Order” opens a dialog using which the order of TestCases can be defined:



`TestConductor::Settings::TestingMode`

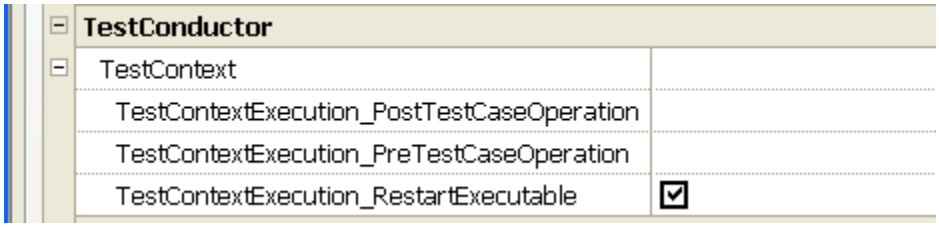
By default, new test architectures created with Rhapsody 7.6 or higher are created with testing mode set to assertion based testing, i.e., the property “TestConductor.Settings.TestingMode” is set to “AssertionBased”. For details regarding the testing modes supported by TestConductor see “Choosing between testing modes” on page 99.

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



Test Context Properties

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



`TestConductor::TestContext::TestContextExecution_RestartExecutable`

If this property is checked (true), for each test case during execution of the test context, the executable of the test context is restarted. If the property is not checked (false), the test cases are executed without restarting the executable after the previous test case has finished its execution.

`TestConductor::TestContext::TestContextExecution_PreTestCaseOperation`

If this property contains a name of an operation of the test context, for each test case during execution of the test context, **before** a test case is executed the operation specified

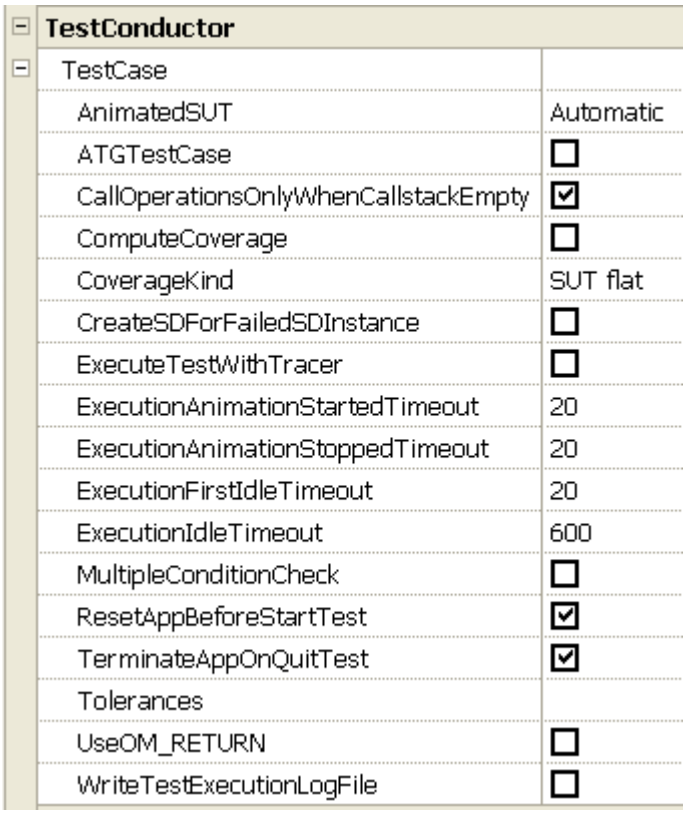
in this property is called automatically. In the operation specified in this property, one can initialize or reset some variables that are needed in the subsequent test case execution.

`TestConductor::TestContext::TestContextExecution_PostTestCaseOperation`

If this property contains a name of an operation of the test context, for each test case during execution of the test context, **after** a test case 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 test case execution.

Test Case Properties

Also some properties for test cases can be set by the user. Some of these properties are set directly by using the execution dialog, some properties you may set using the feature dialog of a test case. Open the **Feature** dialog of a test case, select the **Properties** tab, switch in the dropdown combo box **View** to *All* and navigate to the metaclass `TestConductor::TestCase`



`TestConductor::TestCase::AnimatedSUT`

This property controls the assumptions of TestConductor concerning the animation of the SUT classes. Depending on the fact that the SUT classes are animated or not, TestConductor uses different execution algorithms to control the execution of test cases that are needed in order to execute test cases properly. If this property is set to “Automatic”, TestConductor tries to automatically deduce if the SUT contains animation code or not, and chooses the right execution algorithm accordingly. If the property is set to “true”, TestConductor assumes that the SUT classes contain animation code. If the

property is set to false, TestConductor assumes that there is no animation code for the SUT classes.

Per default the property is set to “Automatic”.

`TestConductor::TestCase::ATGTestCase`

Normally TestConductor injects messages that are defined in a sequence diagram without time delays directly one after the other. In case this property is enabled, TestConductor waits with injection of messages until the system is idle.

This property is enabled automatically for test cases created and exported by ATG.

Per default the property is disabled.

`TestConductor:TestCase:CallOperationsOnlyWhenCallstackEmpty`

If this property is checked, TestConductor delays operation calls that refer to inputs of TestConductor so that these operation calls are made only when the call stack of the focus thread is empty.

If the property is cleared, all operation calls are made by TestConductor immediately even if the call stack of the focus thread is not empty.

Per default the property is disabled.

`TestConductor::TestCase::ComputeCoverage`

In case this property is enabled, TestConductor automatically computes and reports the model coverage achieved when executing the test cases.

Per default the property is disabled.

`TestConductor::TestCase::CoverageKind`

If `TestConductor::TestCase::ComputeCoverage` is enabled, `CoverageKind` defines how the coverage will be measured:

TestConductor supports four different kinds of coverage measures:

- SUT flat: Only coverage of the toplevel class of the SUT is measured, i.e. states, transitions, and operations of parts of the SUT are not considered. Coverage of model elements of test components is also not measured.
- SUT hierachical : Coverage of the SUT is measured in a hierarchical manner, i.e. also states, transitions, and operations of parts of the SUT are hierarchically regarded for coverage measure. Coverage of model elements of test components is again not measured.
- TestContext flat: Coverage is measured in terms of all states, transitions, and operations defined at the first decomposition level of the test context, i.e. all

states, transitions, and operations of the direct parts of the test context are considered.

- TestContext hierarchical : all states, transitions, and operations in the hierarchal structure of the test context are considered in coverage measure.

Per default the property is set to “SUT flat"..

`TestConductor::TestCase::CreateSDForFailedSDInstance`

In case this property is enabled, TestConductor automatically creates a failure sequence diagram (**Show as SD**) and stores it in the model.

Per default the property is disabled.

`TestConductor::TestCase::ExecuteTestWithTracer`

In case this property enabled, the execution of this test case will be done with activated tracer (trace #all all).

Per default the property is disabled.

`TestConductor::TestCase::ExecutionAnimationStartedTimeout`

Defines the time (in seconds) that TestConductor waits for the animated application to connect to Rhapsody. If the application does not connect to Rhapsody within the specified time, the test case execution is aborted. The default value is 20 seconds.

`TestConductor::TestCase::ExecutionAnimationStoppedTimeout`

Defines the time (in seconds) that TestConductor waits for the animated application to terminate after receiving the terminate command from TestConductor. If the application does not terminate within the specified time, TestConductor simply proceeds. The default value is 20 seconds.

`TestConductor::TestCase::ExecutionFirstIdleTimeout`

Defines the time (in seconds) that TestConductor waits for the animated application to become idle after giving the first “Go Idle” command. If the application does not become idle within the specified time, the test case execution is aborted. The default value is 20 seconds.

`TestConductor::TestCase::ExecutionIdleTimeOut`

In case a timeout is defined (> 0) and the application does not show any activity for the defined timeout (in seconds) the execution of this test case is interrupted.

The testing profile defines a global timeout, which can be overwritten for every test package, test context and test case. This default value in the testing profile is 600 seconds.

Setting this property to zero means that no timeout is enabled.

`TestConductor::TestCase::MultipleConditionCheck`

TestConductor can be configured to check the reached condition and following conditions without system activity, until one condition mark evaluates to `FALSE`. To change the default TestConductor behaviour change the property `TestConductor::TestCase::MultipleConditionCheck` of the test case to `TRUE`. For further information read the chapter Condition Marks at page *167*.

Per default the property is `FALSE`.

`TestConductor::TestCase::ResetAppBeforeStartTest`

In case this property is enabled, TestConductor will reset the application to the initial state of the model for each test case execution. Normally this property is set using the test execution dialog for sequence diagram based test cases.

Per default the property is enabled.

Note: This property is available for sequence diagram test cases only. This property is currently not interpreted for source code, flow chart and statechart test cases.

`TestConductor::TestCase::TerminateAppOnQuitTest`

This property controls the behavior of TestConductor after quitting a test. In case this property is enabled, the application terminates after quitting the test. Otherwise only TestConductor quits.

Per default the property is enabled.

Note: This property is available for sequence diagram test cases only. This property is currently not interpreted for source code, flow chart and statechart test cases.

`TestConductor::TestCase::Tolerances`

This property is an internal property where TestConductor stores tolerance definitions defined in the sequence diagram test definition dialog. User should not set this property directly.

Note: This property should not be set directly. Please use the corresponding **Tolerances** button in the **Define Test** dialog.

`TestConductor::TestCase::UseOM_RETURN`

In case this property is enabled, TestConductor checks return values by evaluating a specific animation message that is generated by the application if the operation whose return value should be checked uses the animation macro OM_RETURN. If this property is disabled, TestConductor can only check return values for operation calls that originate from TestComponents.

Per default the property is disabled.

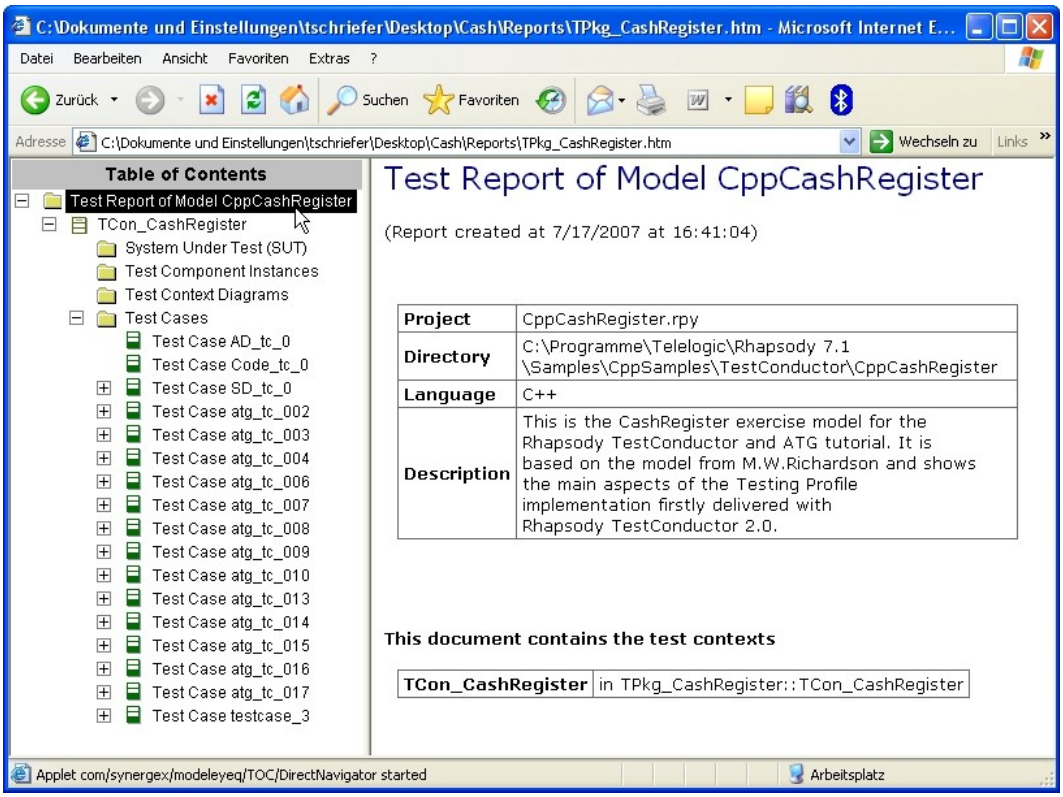
`TestConductor::TestCase::WriteTestExecutionLogFile`

TestConductor generates a log file of the test case execution if this property is enabled. TestConductor stores this log file (`RTC_log.txt`) into the folder `C:\tmp`. The folder must exist and the user must have write access to this folder.

Per default the property is disabled.

Generating Test Reports with Rhapsody ReporterPLUS

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



Note: The report templates currently will not show pictures of subscenarios or linked subscenarios of test cases. Only the top level diagrams of scenarios and flow charts are currently displayed.

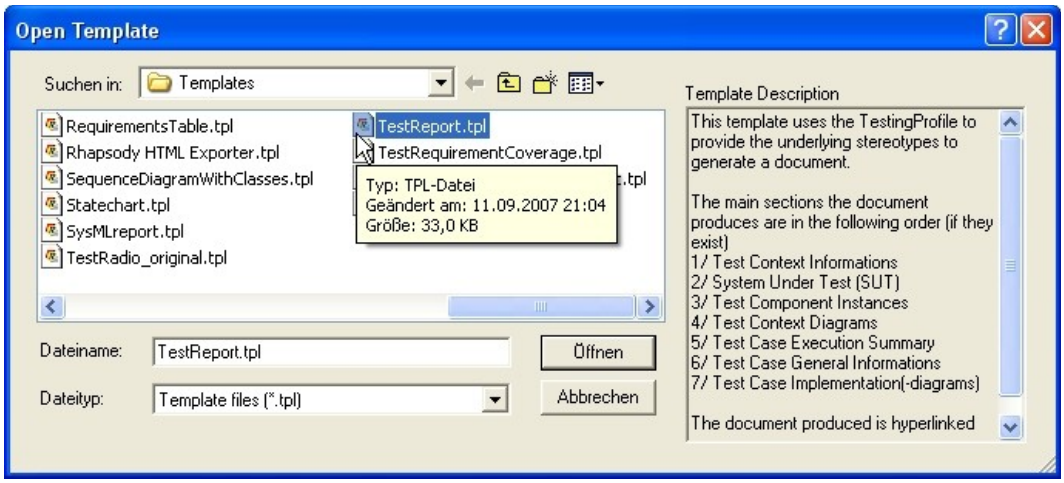
Executing the Test Report

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

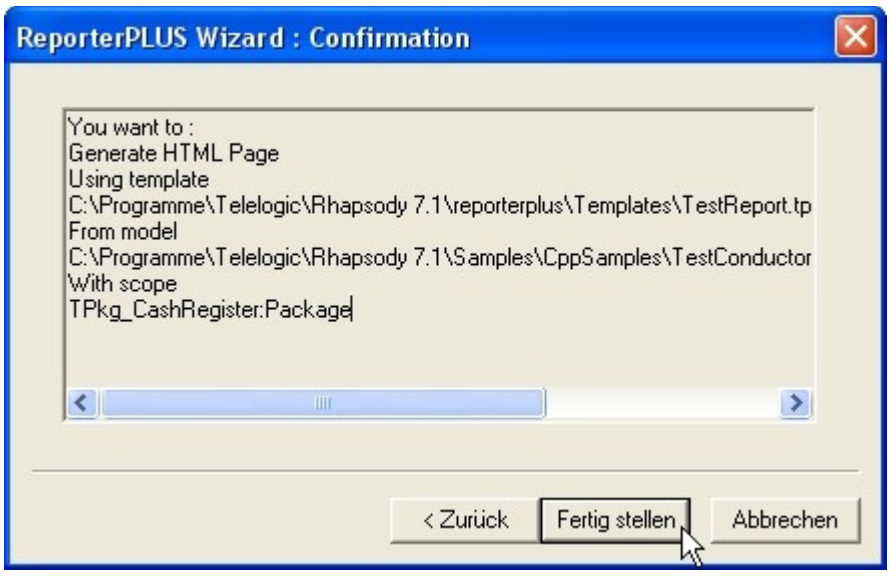
- In case you want to create the report only for a selected test package and the containing test packages, select in the Rhapsody browser a test package and choose from the menu **Tools > ReporterPLUS > Report on selected package…**

- In case you want to create the report for all test packages in the model choose from the menu **Tools > ReporterPLUS > Report on all model elements…**
- In the Rhapsody ReporterPLUS wizard **Select Task** specify the export file format your report shall be displayed in and click **Next>**.

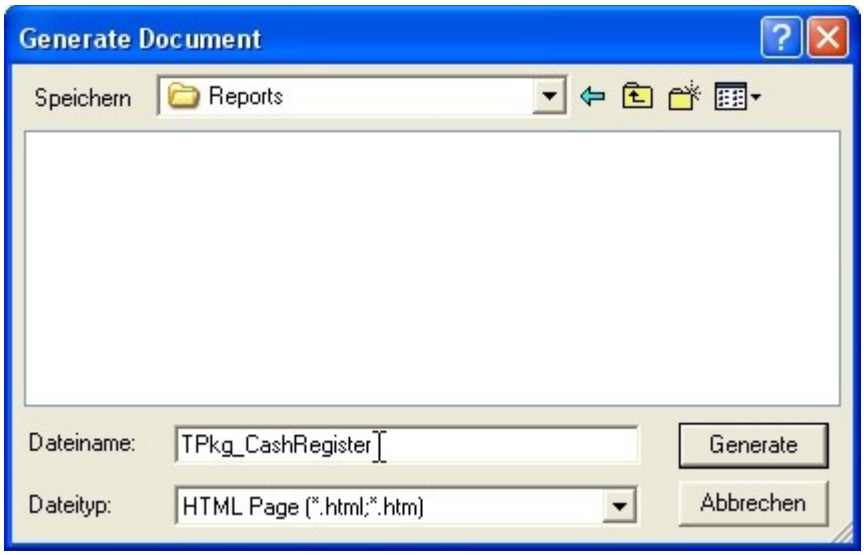
- In the Rhapsody ReporterPLUS wizard **Select Template** check the currently active template. In case the template "`TestReport.tpl`" is not active click on "**…**", open it from the folder "`reporterplus\Templates`" in your Rhapsody installation folder and click **Next>**.



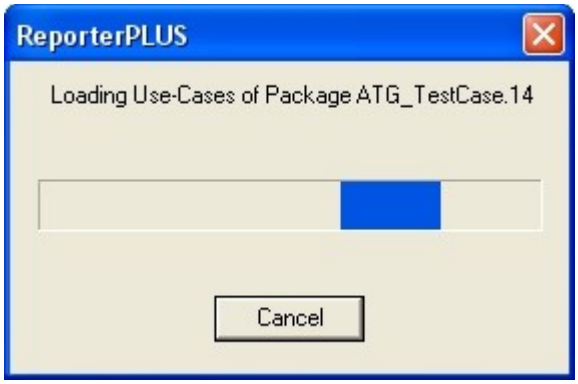
- The Rhapsody ReporterPLUS wizard **Confirmation** gives an overview about the selected options. Click the button **<Back** to change the options. Click **Generate** to start the execution of the Rhapsody ReporterPLUS report generation.



- In the dialog **Generate Document** specify a path and a name for the document to generate and click the button **Generate**.



- Rhapsody ReporterPLUS will show the progress during creating the document and start the corresponding application to show the test report.

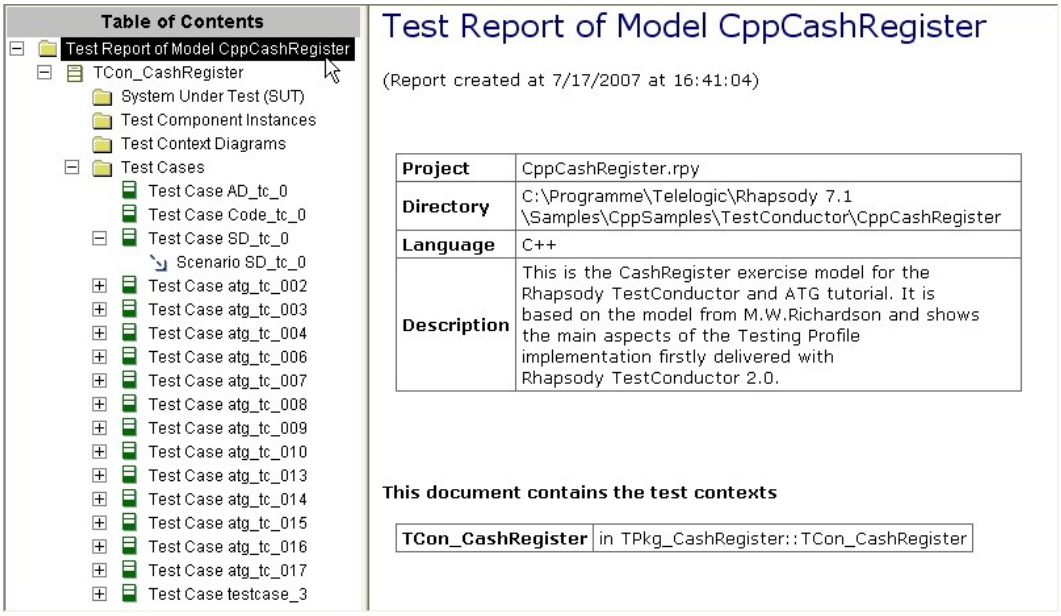


Using the HTML Test Report

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

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

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



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

Conceptual this report lists all test contexts of the specified test package(s) during creation. For each test context you will find information about

- the system under test
- the test component instances
- the test context diagrams
- the test cases and their execution status

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

Using the Test Requirement Coverage Report

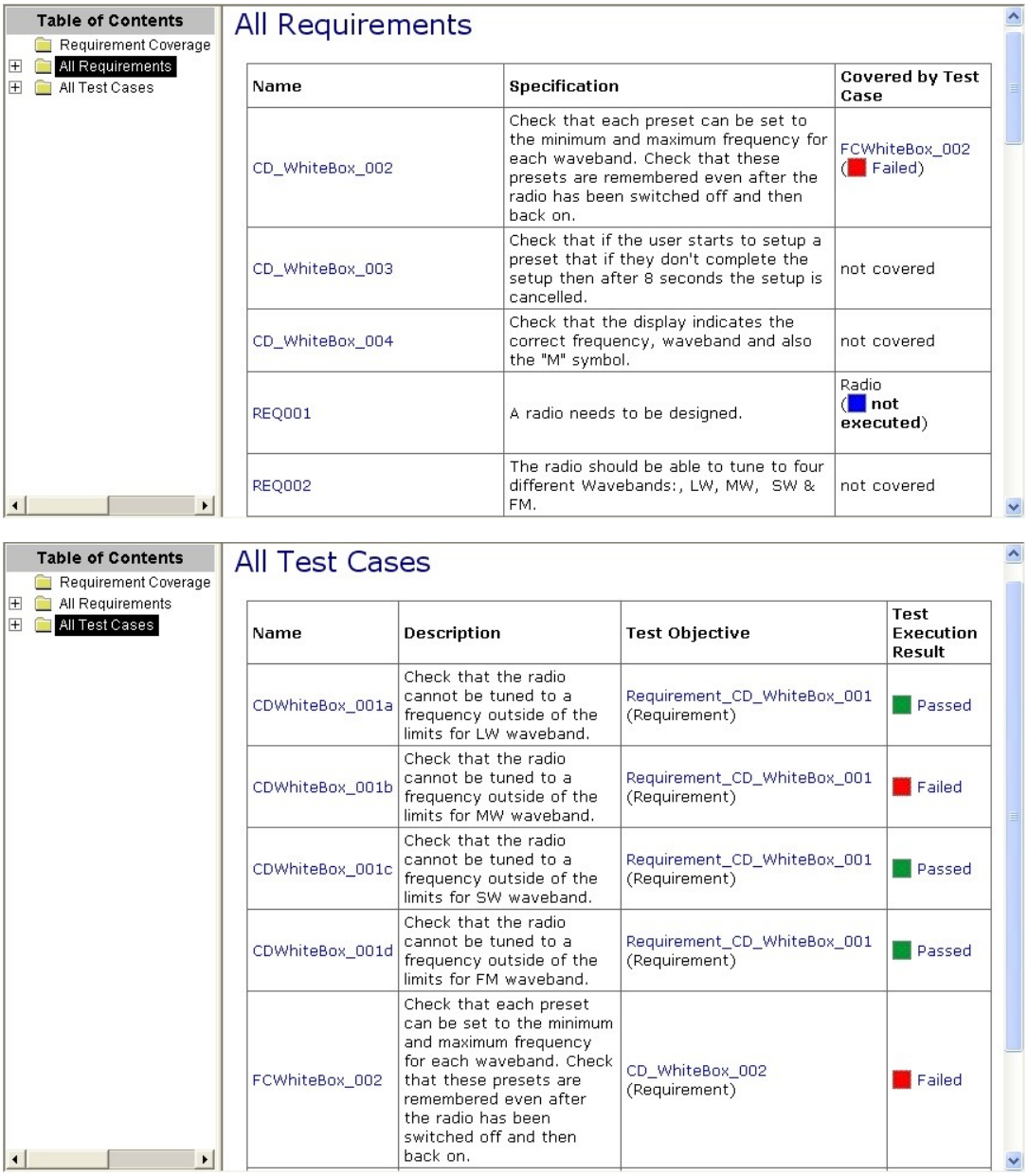


Table of Contents
- Requirement Coverage
- All Requirements
- All Test Cases

All Requirements

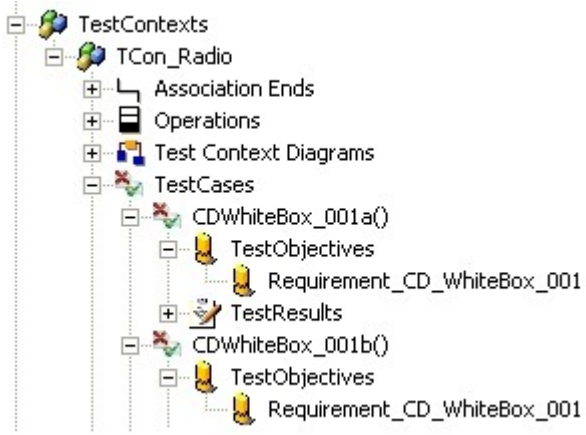
Name	Specification	Covered by Test Case
CD_WhiteBox_002	Check that each preset can be set to the minimum and maximum frequency for each waveband. Check that these presets are remembered even after the radio has been switched off and then back on.	FCWhiteBox_002 (Failed)
CD_WhiteBox_003	Check that if the user starts to setup a preset that if they don't complete the setup then after 8 seconds the setup is cancelled.	not covered
CD_WhiteBox_004	Check that the display indicates the correct frequency, waveband and also the "M" symbol.	not covered
REQ001	A radio needs to be designed.	Radio (**not executed**)
REQ002	The radio should be able to tune to four different Wavebands:, LW, MW, SW & FM.	not covered

Table of Contents
- Requirement Coverage
- All Requirements
- All Test Cases

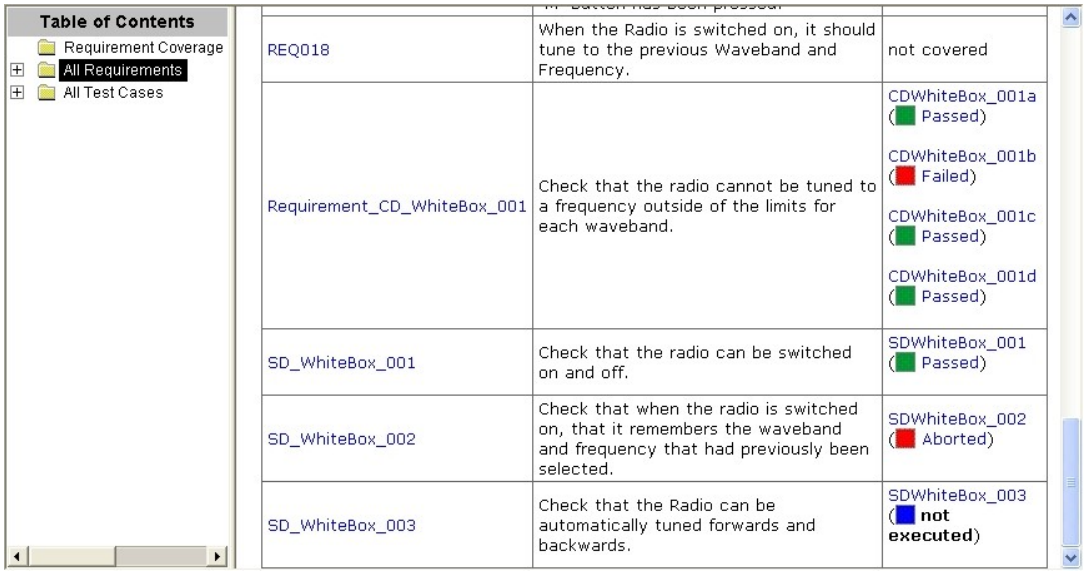
All Test Cases

Name	Description	Test Objective	Test Execution Result
CDWhiteBox_001a	Check that the radio cannot be tuned to a frequency outside of the limits for LW waveband.	Requirement_CD_WhiteBox_001 (Requirement)	Passed
CDWhiteBox_001b	Check that the radio cannot be tuned to a frequency outside of the limits for MW waveband.	Requirement_CD_WhiteBox_001 (Requirement)	Failed
CDWhiteBox_001c	Check that the radio cannot be tuned to a frequency outside of the limits for SW waveband.	Requirement_CD_WhiteBox_001 (Requirement)	Passed
CDWhiteBox_001d	Check that the radio cannot be tuned to a frequency outside of the limits for FM waveband.	Requirement_CD_WhiteBox_001 (Requirement)	Passed
FCWhiteBox_002	Check that each preset can be set to the minimum and maximum frequency for each waveband. Check that these presets are remembered even after the radio has been switched off and then back on.	CD_WhiteBox_002 (Requirement)	Failed

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

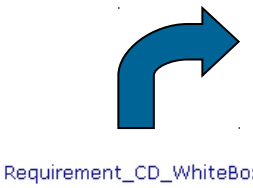


This requirement coverage report focus especially on the dependency between a requirement and a test case. The test requirement coverage report gives another view on the model. At a glance the user is able to verify, that e.g. the requirement "`Requirement_CD_WhiteBox_001`" is covered by the test cases `CDWhiteBox_001a`, `CDWhiteBox_001b`, `CDWhiteBox_001c` and `CDWhiteBox_001d`, where `CDWhiteBox_001b` is currently `FAILED` and in result the requirement "`Requirement_CD_WhiteBox_001`" is not fulfilled.



In opposite to the view "**All Requirements**", the report also shows a table with "**All Test Cases**" of the model. The "**All Test Cases**" view is assistant to check, whether a test case has a test objective.

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



Requirement_CD_WhiteBox_001

Requirement Requirement_CD_WhiteBox_001

Description	no description
Specification	Check that the radio cannot be tuned to a frequency outside of the limits for each waveband.
Package	Radio_TestPlan
Full Path	RequirementsPkg::Radio_TestPlan.Requirement_CD_WhiteBox_001
Covered by Test Case	CDWhiteBox_001a (Passed), CDWhiteBox_001b (Failed), CDWhiteBox_001c (Passed), CDWhiteBox_001d (Passed),
Anchored Elements	CDWhiteBox_001a (Operation), CDWhiteBox_001b (Operation), CDWhiteBox_001c (Operation), CDWhiteBox_001d (Operation),

Customizing the Test Report

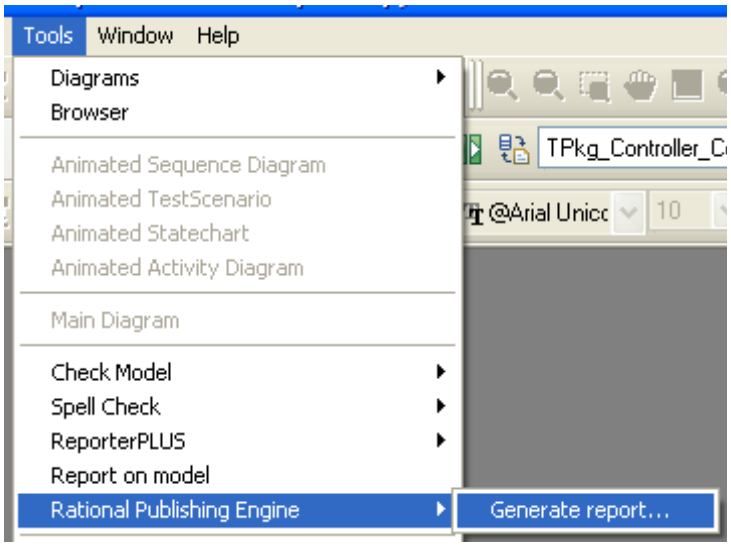
The test report template is customizable to fit specific users requirements. Follow the Rhapsody ReporterPLUS documentation how to adapt it to your needs.

Generating Test Reports with Rational Publishing Engine

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

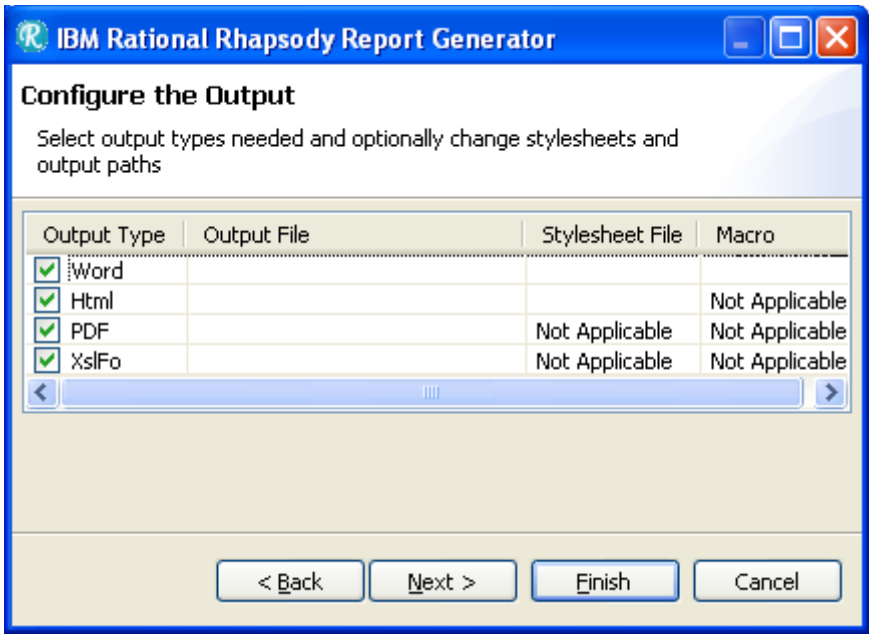
Creating the Test Report

- Choose from the menu **Tools > Rational Publishing Engine > Generate report…**



- Select the RPE template which should be used for report generation. The template "`TestRequirementCoverage.dta`" must be selected to create a requirement coverage report.

- Specify which types of output files should be created and where they should be saved.



- Then RPE automatically creates the selected reports.

Test Requirement Coverage Report

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

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

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

All Requirements

Name	Specification	Covered By Test Case
Mission	<<No Model Data>>	• TPkg_TheCalc::TCon_TheCalc_Architecture::TCon_TheCalc.SD_tc_plus PASSED for TPkg_TheCalc::TPkg_TheCalc_MIL_Comp::MIL_DefaultConfig • TPkg_TheCalc::TCon_TheCalc_Architecture::TCon_TheCalc.SD_tc_minus PASSED for TPkg_TheCalc::TPkg_TheCalc_MIL_Comp::MIL_DefaultConfig • TPkg_TheCalc::TCon_TheCalc_Architecture::TCon_TheCalc.SD_tc_mult_div PASSED for TPkg_TheCalc::TPkg_TheCalc_MIL_Comp::MIL_DefaultConfig
Notation	<<No Model Data>>	• TPkg_TheCalc::TCon_TheCalc_Architecture::TCon_TheCalc.SD_tc_2 FAILED for TPkg_TheCalc::TPkg_TheCalc_MIL_Comp::MIL_DefaultConfig
OperationErrors	<<No Model Data>>	• not covered
Division by zero	<<No Model Data>>	• TPkg_TheCalc::TCon_TheCalc_Architecture::TCon_TheCalc.SD_tc_1 PASSED for TPkg_TheCalc::TPkg_TheCalc_MIL_Comp::MIL_DefaultConfig

Requirement Division by zero

Description	It is an unrecoverable error if operand register 2 is 0 when operation is 4 (=/)
Specification	no description
Package	RequirementsPkg
Full Path	RequirementsPkg::Division by zero
Covered by Test Case	• TPkg_TheCalc::TCon_TheCalc_Architecture::TCon_TheCalc.SD_tc_1: PASSED for TPkg_TheCalc::TPkg_TheCalc_MIL_Comp::MIL_DefaultConfig
Anchored Elements	no anchored elements

The test cases specified in the model are listed, too. Again detailed information are available for each test case.

All Test Cases

Name	Description	Test Objectives	Test Execution Result
SD_tc_plus	no description	Mission (Requirement)	PASSED for TPkg_TheCalc::TPkg_TheCalc_MIL_Comp::MIL_DefaultConfig
SD_tc_1	no description	Division by zero (Requirement)	PASSED for TPkg_TheCalc::TPkg_TheCalc_MIL_Comp::MIL_DefaultConfig
SD_tc_2	no description	Notation (Requirement)	FAILED for TPkg_TheCalc::TPkg_TheCalc_MIL_Comp::MIL_DefaultConfig
SD_tc_minus	no description	Mission (Requirement)	PASSED for TPkg_TheCalc::TPkg_TheCalc_MIL_Comp::MIL_DefaultConfig
SD_tc_mult_div	no description	Mission (Requirement)	PASSED for TPkg_TheCalc::TPkg_TheCalc_MIL_Comp::MIL_DefaultConfig

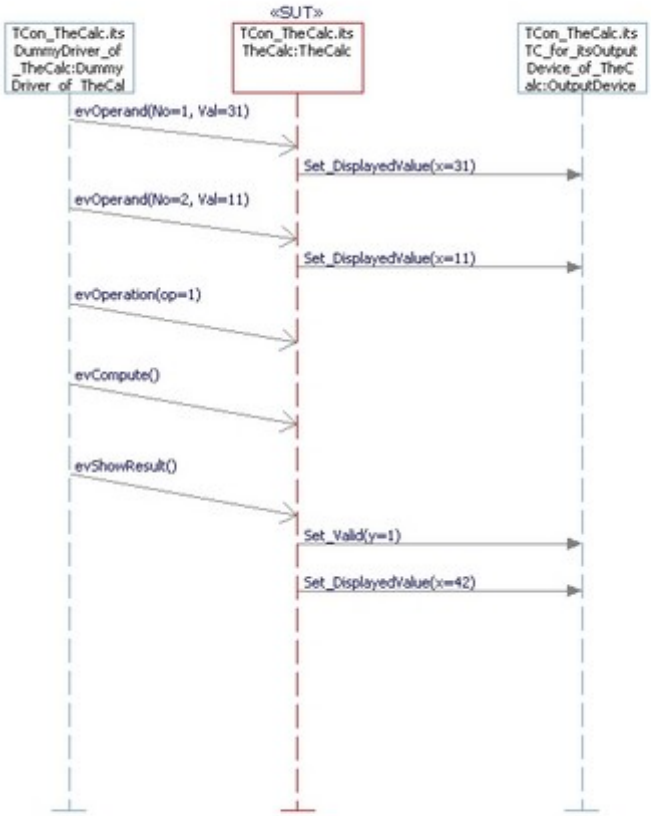
SD_tc_plus

Description	no description
Test Execution Result	**Passed** for TPkg_TheCalc::TPkg_TheCalc_MIL_Comp::MIL_DefaultConfig

Implementation

itsCSC_SD_tc_plus.GEN(evTCStart);

Test Scenario SDTestScenario_0



Test Objectives

Name of Test Objective	Anchored Model Element	Metaclass
Mission	Mission	Requirement

Creating Report Templates

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

Using the TestConductor API

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

- VBA Scripts
- 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. A typical sequence would look as follows (using VBA):

…

```
Set app = GetObject(, "rhapsody.Application")

Set proj = getProject()

Set testcase = proj.findElementsByFullName("TestPackageA.TestContextB.TestCaseC")

‘ highlight the selected element

testcase.highLightElement();

‘ now one can execute a TestConductor command

app.runHelper(“Execute TestCase Sync”)
```

…

The sample "CppSamples/TestConductor/TestConductorAPI" shows how to access the TestConductor API from within VBA scripts and Java programs. Additionally, the sample "CppSamples/TestConductor/CppTestAutomationSample" shows how to use the API in order to automate your testing workflows..

Available TestConductor API Commands

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

Applicable to TestCase elements:

- "Edit TestCase SDInstances"
- "Update TestCase"
- "Build TestCase"
- "Execute TestCase"
    - Performs asynchronous TestCase execution, i.e., the function returns immediately and the execution of the TestCase is performed in a separate thread. The API script has to ensure itself (e.g. by waiting a

specified amount of time) that the TestCase execution has finished before additional TestConductor API commands can be executed.

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

Applicable to TestContext elements

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

Applicable to TestPackage elements

- "Create TestContext"
- "Update TestPackage"
- "Clean TestPackage"
- "Build TestPackage"
- "Execute TestPackage"
    - Performs asynchronous TestPackage execution, i.e., the function returns immediately and the execution of the TestPackage is

performed in a separate thread. The API script has to ensure itself (e.g. by waiting a specified amount of time) that the TestPackage execution has finished before additional TestConductor API commands can be executed.

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

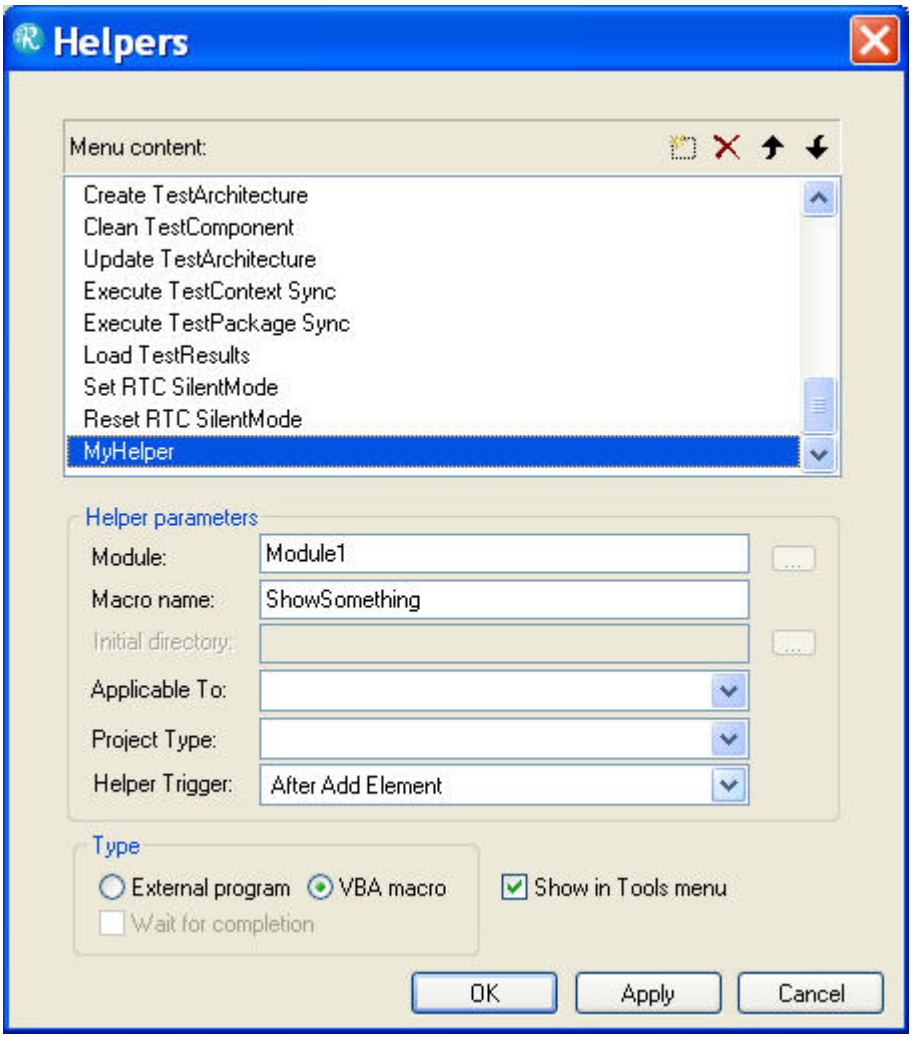
Applicable to Class elements

- "Create TestArchitecture"

Defining Callbacks for TestConductor functions

In addition to using the TestConductor API directly, one can also execute automated scripts after certain TestConductor actions like e.g. creating test architectures. In order to do this, one can use triggered helpers as provided by Rhapsody. For instance, to specify that after test architecture creation a 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 VBA script or by an external program that uses the Rhapsody API.



- Now, when doing “Create TestArchitecture”, after the test architecture has been created the specified helper is invoked automatically.

Besides “Create TestArchitecture”, helpers with helper trigger “After Add Element” are also invoked automatically for all other TestConductor functions that create new elements, like e.g. “Create Code TestCase”.

Advanced Test Definition

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.

Graphical Feature Support

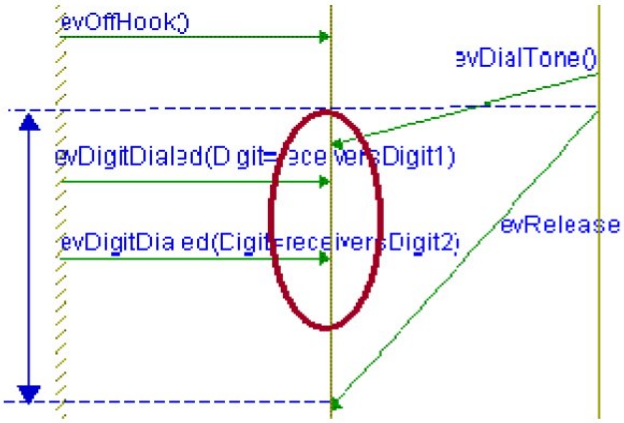
TestConductor supports the standard UML sequence diagram elements, as available in the Rhapsody sequence diagram editor. However, some of these elements are not yet fully supported.

TestConductor supports the following graphical features:

- Test component lines, which specify classes with driver operations or stub operations
- Test context lines, which specify the boundary of the system under test including their test components
- Environment lines, which specify the boundary of a system under test
- Actor instance lines for reactive actor classes (those containing state charts). These classes represent external behaviour against the system under test.
- Object instance lines, which specify the communication behaviour inside the system under test
- Horizontal and slanted message arcs between object instances (including actor instances), which specify events, triggered operations, operation calls, and their argument values. Unspecified messages (messages with realization *unspecified*) and unrealized message (messages with Stereotype *unrealized*) are ignored.
- Messages to itself, which specify that the source and the target of events and operation calls is the same object instance.
- Dataflow messages among object instances.
- Condition marks, which specify synchronization points in a sequence diagram
- Events originating at the environment axis, which specify that external events trigger the system under test.
- Only assertion based testing mode: Interaction operators “opt”, “alt”, “loop”, “break”, “consider”, “parallel”

Synchronous and Asynchronous Messages

Rhapsody supports the concepts of *synchronous* and *asynchronous* messages. Both of these concepts can be used when you define and execute tests.



Note the following:

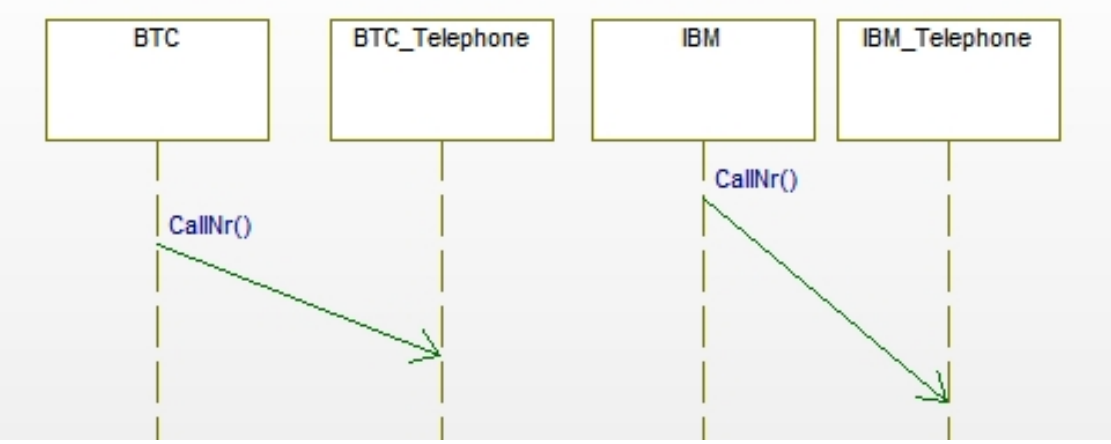
- Only event messages, which are asynchronous, can be interfered by another message.
- Operation calls are synchronous and do not admit any interference.

TestConductor associates for every event message in a sequence diagram two actions—sending and receiving. In opposite to event messages TestConductor associates only one action to operation calls and dataflows. During a test execution with TestConductor, you can drive a specified sequence diagram and monitor (in the execution dialog) the total number of actions and those that passed successfully.

Linear and Partial Order

TestConductor can interpret a sequence diagram either in *linear order* or in *partial order* mode. To understand why partial order interpretation of sequence diagrams is sometimes necessary to specify monitors, consider the following example. Assume that the companies `CompanyA` and `CompanyB` want to set up a conference call. You want to monitor the situation that both parties are eventually connected to the conference call. The following sequence diagram specifies that each party dials a conference `CallNr()`. Regardless of the order the parties dial and connect, the monitor must be fulfilled whenever both parties have connected. In the sequence diagram every message `CallNr()` specifies two ordered actions:

- Sending the `CallNr()` event by a party
- Consumption of the `CallNr()` event by the telephone corresponding to the calling party

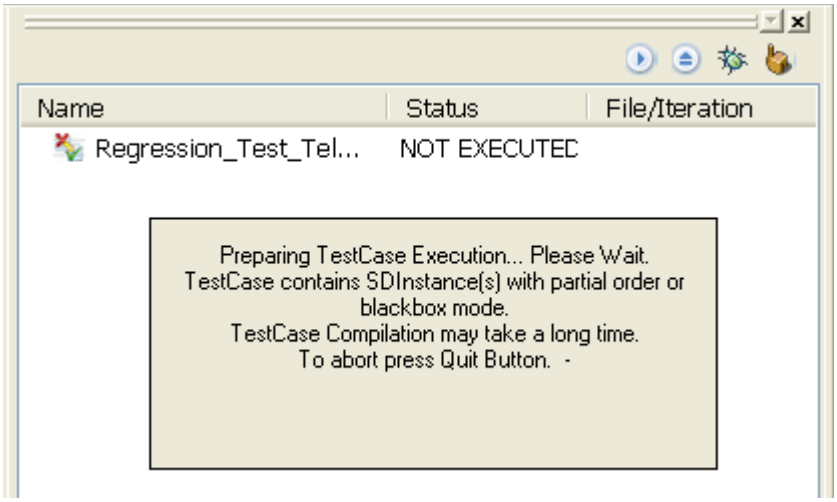


If you had only linearly ordered monitor sequence diagrams, you could not express the required independency of the connection order. Note that there are six possible dialing-and-connection orders for the parties:

```
   (CompanyA_Dial - CompanyB_Dial - CompanyA_Connect -
CompanyB_Connect)
   (CompanyA_Dial - CompanyB_Dial - CompanyB_Connect -
CompanyA_Connect)
   (CompanyA_Dial - CompanyA_Connect - CompanyB_Dial -
CompanyB_Connect)
   (CompanyB_Dial - CompanyA_Dial - CompanyB_Connect -
CompanyA_Connect)
   (CompanyB_Dial - CompanyA_Dial - CompanyA_Connect -
CompanyB_Connect)
   (CompanyB_Dial - CompanyB_Connect - CompanyA_Dial -
CompanyA_Connect)
```

Every sequence diagram interpreted in linear order could specify only one of these possible connection orders (for example, the linear order of the connections shown in the sequence diagram considered above is "`CompanyA_Dial - CompanyB_Dial - CompanyB_Connect - CompanyA_Connect`", because the evaluation order is from top to bottom). Hence, with linear order you must define six different monitor sequence diagrams. Note that five of these monitors would lead to a failure during testing; only one would pass in every test execution. If you interpret this sequence diagram in partial order, it represents all the possible (six) orders. This is due to the fact that you do not enforce any order between pair wise independent sending and receiving of the `CallNr()` events. Sending and receiving of an event on the `CompanyB` side is independent from the `CompanyA` side.

Test execution with partial order might result in extreme compilation times. TestConductor has a facility to interrupt the execution when it takes too long.



By pressing the “Abort” icon in the icon toolbar aborts the compilation and test case execution.

Note: Partial order set together with the **driver and monitor** option implies that driving the input events is independent from monitoring the internal messages. To avoid the arising nondeterminism, TestConductor first drives inputs and then monitors internal messages. **TestConductor chooses one valid order of messages to be driven (in particular, this order changes in general when the same sequence diagram test case is executed repeatedly).** Such nondeterminism does not exist for linear order interpretation, because it is a precise order between all messages in a sequence diagram. Also note that there is no nondeterminism for monitor only, because you decide when you inject all inputs, and TestConductor monitors internal messages as they appear in the running model.

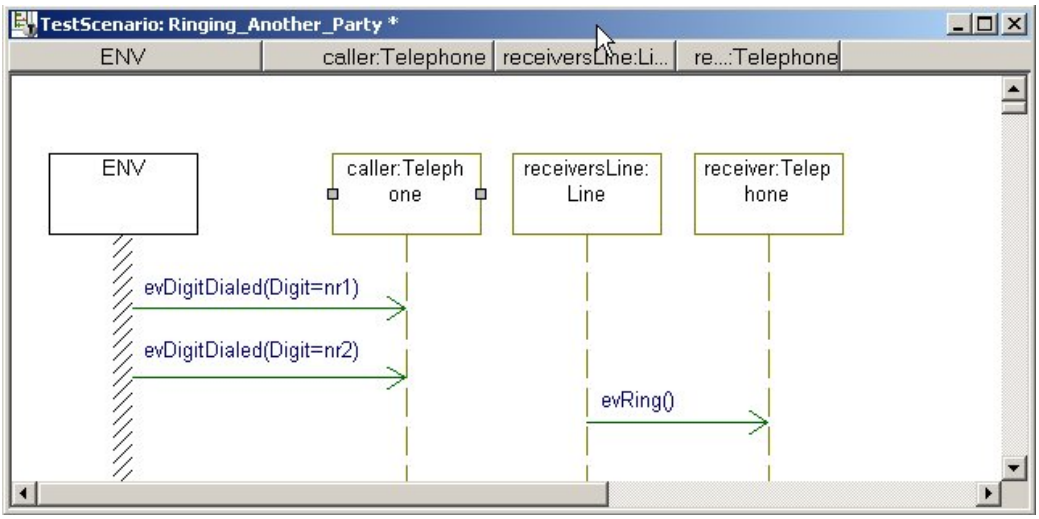
Parameters

One of the most important aspects of reusing sequence diagrams is the possibility to parameterize them. By using *parameters* such as “X” and “Y” as object names for sequence diagram instances, all combinations of objects of the corresponding classes can be treated within one sequence diagram. You must instantiate these parameters with different concrete objects of the system.

Parameters are used to specify sequence diagrams, which can be used as test patterns or as generic sequence diagrams in test definitions. Parameterized sequence diagrams can be used more than once in the same test configuration, or they can be used in various contexts in different test configurations. Parameters can be applied for instance names and for argument lists of events and operations. Instance names in a Rhapsody sequence diagram must be either concrete names or parameters. For example, if an instance line is labelled “`X1:Telephone`”, `X1` is a parameterized object instance name of class Telephone that will be mapped to a concrete object instance name when the sequence diagram is instantiated as part of a test definition. In other words, `X1` can be mapped to `PBX[0]->itsTelephone[0]`. Parameters are useful when you are defining multiple tests with a similar structure, such as the PBX sample where `Telephone 1` can connect to `Telephones 2`, `3`, and `4`. Using parameters, you can specify sets of similar tests by

specifying one common sequence diagram for these cases. To manually generate multiple test cases, simply bind the sequence diagram parameters to various concrete values.

In the following example, the sequence diagram contains the parameters `caller`, `receiversLine`, `receiver`, `nr1`, and `nr2`. The first three parameters represent parameterized instance names, whereas the last two describe attribute values for parameterized events. Due to the concept of parameters, this sequence diagram can be used as a test pattern to specify and execute caller-receiver tests for the pairs of telephones. This is done by instantiating the sequence diagram several times.



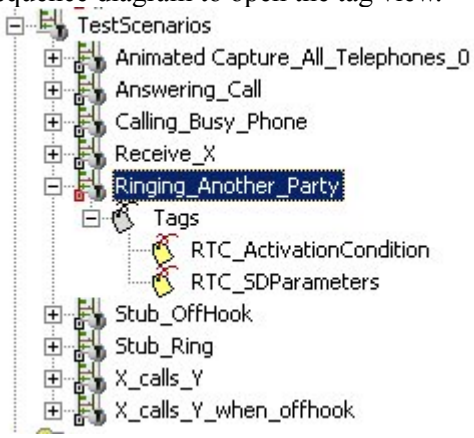
Defining Parameters

TestConductor supports test definitions based on sequence diagrams, whose instances either have a concrete or *parameterized name*. Parameterized name means that it is not a valid, or concrete, object name as usually used in Rhapsody. You can also use *an anonymous class name* that is without a concrete name or parameter. In this case, in accordance with Rhapsody, the class name is internally expanded to the unique concrete object instance. During test execution, sequence diagrams are animated in relation to the default names. Note that parameters have no default values. You can specify parameters for a sequence diagram by declaring them in the Tag *RTC_SDParameter* which is available for each test scenario sequence diagram.

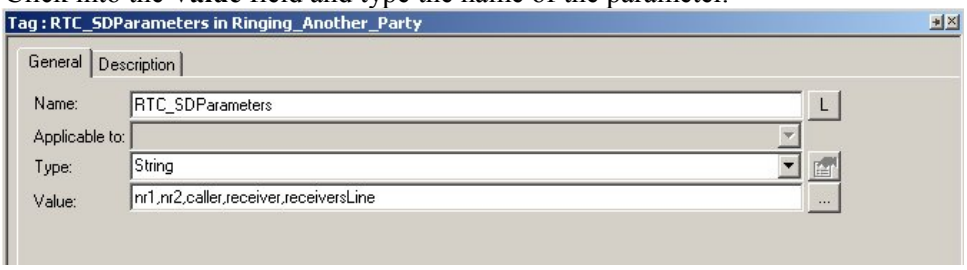
To declare parameters for a sequence diagram do the following:

1. Open a Rhapsody sequence diagram in a Rhapsody project.
2. In the names pane, specify the objects names of the classes `Telephone` and `Line`. Give a parameterized name, such as `caller:Telephone`. Give the concrete names for another instance depicted in the sequence diagram like `PBX[0]->itsTelephone[0]: Telephone`. You can leave an instance “anonymous” like `Line`. Rhapsody considers such a specification as a concrete class instance with the default name `PBX[0]-> itsLine[0]:Line`.

3. In the Rhapsody browser, click on the cross beside of the name of the test scenario sequence diagram to open the tag view.



4. Open the **Feature** dialog of the corresponding RTC_SDParamters tag
5. Click into the **Value** field and type the name of the parameter.



Note: Make sure that you type the identical names of parameters as specified in the current sequence diagram. TestConductor cannot determine misspelling.

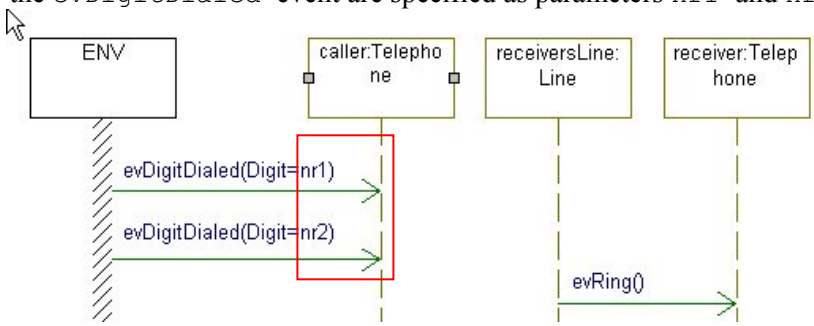
Note: TestConductor adds properties to the sequence diagrams when models are opened, in case these properties were not added before. This is why existing models with sequence diagrams are marked as changed (red icon) along with the sequence diagrams when projects are loaded for the first time after TestConductor was installed.

If a sequence diagram contains two or more parameters, separate their names using commas, then click **OK**. The following figure shows how to specify multiple parameters.

You can apply parameters to message argument lists to specify more flexible, generic sequence diagrams as templates in test definitions. Parameterized arguments of messages are used, for example, when input stimuli correspond to parameterized object names in the same sequence diagram or in the same test configuration.

To extend the parameter list of a sequence diagram with parameterized arguments, do the following:

1. Open the sequence diagram in the Rhapsody sequence diagram editor and specify event or operation arguments as parameters inserting their parameterized names in the object pane. As an example, in the following figure, values of the `Digit` argument of the `evDigitDialed` event are specified as parameters `nr1` and `nr2`



2. Using the Rhapsody browser, open the **Feature** dialog of the corresponding RTC_SDParamters tag and extend the list of the parameters typing "`nr1,nr2`" in addition to the existing parameters in the **Value** field.
3. Click **OK** to accept the change of the parameter list.

The specification defined with the generic "Ringing_Another_Party" sequence diagram, says that whenever a calling telephone is taken off the hook and dials an extension, the receiving telephone rings. Note that the sequence diagram does not specify which telephone is calling, which one is the receiver, nor the extension dialed.

Parameter Mapping

You can consider Rhapsody sequence diagrams with parameters as "classes of sequence diagrams", whereas sequence diagrams with parameters mapped to real objects represent "instances of sequence diagram classes." One parameterized sequence diagram can be used in various contexts: in different test configurations, or in the same test configuration with different parameter mappings. It catches several requirements similar in structure (order of messages) and different only in the names of the involved instances.

As an example, the "Ringing_Another_Party" sequence diagram can specify that `Telephone 1` calls `Telephone 4`. To do this, map its parameters to the following object names in the PBX model:

```
caller: PBX[0]->itsTelephone[0]
receiversLine: PBX[0]->itsLine[3]
receiver: PBX[0]->itsTelephone[3]
nr1: 1
nr2: 4
```

The following table lists the extension for each telephone.

Telephone	Extension
Telephone 1	11
Telephone 2	12
Telephone 3	13
Telephone 4	14

In this example, mapping parameter `nr2` to `3` instead of `4` leads to the "concrete" specification corresponding to "Whenever Telephone 1 dials the extension of Telephone 3, Telephone 4 rings". TestConductor will show that this specification cannot be met by the real behaviour of the model.

Note: During execution parameter values containing quotes will consistently be stripped, e.g. the expression "OK" will be converted to OK and ""OK"" will be converted to "OK".

Using Time Interval for Delay Driving from Environment and TestComponents

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

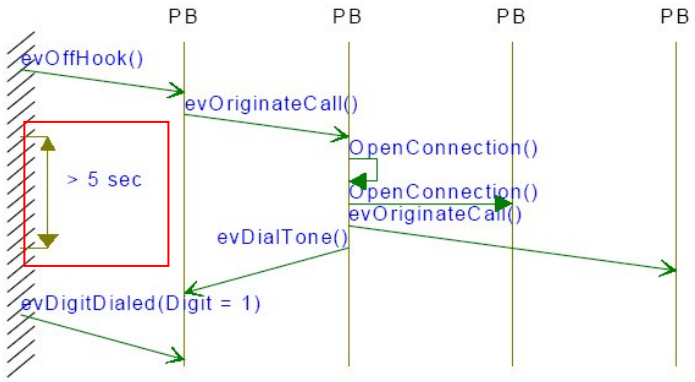
Note: TestConductor will regard only time intervals between messages, if driving messages are defined from the ENV line and the time interval definition is also specified on the ENV line or if driving messages are defined from a TestComponent instance line and the time interval definition is specified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

Time delays will be specified with the time interval notation in sequence diagrams. TestConductor supports time intervals if they are associated with system border or TestComponent instance lines. The label of a time interval specifies the time unit (`ms, sec, min`) and a time value. Essentially, there are two slightly different Time Interval annotations with a slightly different execution semantics. The first variant uses the following syntax:

Syntax: `> 5 sec`

Here, TestConductor must wait at least 5 seconds before it may drive the next message. Other time interval formats are **"**`> 500 ms`**"** and "`> 5 min`**".** TestConductor creates a timer in the tested application which elapes after the amount of time specified in the Time Interval.

The start point of a time interval is always associated with the next message point above the time interval (on any instance line). The end point of a time interval is always associated with the next message point below the time interval (again on any instance line).



After driving `evOffHook()` and observing `evOriginateCall()` TestConductor must wait 5 seconds before it may drive `evDigitDialed(Digit=1)`.

TestConductor must monitor all system reactions before `evDigitDialed(Digit=1)`, including `evDialTone()`.

The second variant of Time Intervals are those which uses the following syntax for time annotations:

Syntax: `>> 5 sec`

When using this syntax, in contrast to the "> 5 sec" case TestConductor does not create an own application timer when starting the time interval. Instead TestConductor will use the time of the tested application. As a result, TestConductor will only proceed if the tested application time increases at least the specified amount of time. In contrast to the "> 5 sec" syntax TestConductor may proceed later than the specified amount of time, since the tested application time might increase to a larger amount of time than the specified time interval.

TestConductor also allows that time intervals overlap if several messages to be driven are constrained via time intervals. This means, TestConductor will manage several timers for the driven messages at the same time, no matter if they are specified on the same instance line or on different instance lines. For every time interval there always exists a unique predecessor and successor message to be driven in the sequence diagram.

Activation Conditions

Activation conditions are used to specify the point in time during model execution when sequence diagram instances become activated. You can use activation conditions to model a predecessor order between several sequence diagram instances in a test definition. Activation conditions can specify a starting point of sequence diagram instance simulation, such as event sending or event receiving, which in turn can be a result of the behavior defined by another sequence diagram. TestConductor supports conditional expressions for events and conditions in the following form:

```
ObjectName->CondName(Parameters)
```

In this syntax:

- `ObjectName` is a parameterized or concrete name of a class instance or an ENV (environment variable), which can be represented by the system border.
- `CondName` is a particular kind of event, state, or method action.
- `Parameters` is a state of a state chart, or the name of an event or method, and the receiver of this event or method, depending on the `CondName`.

The exact syntax is described under Syntax for Activation Conditions / Condition Marks (see page *255*) in the appendix.

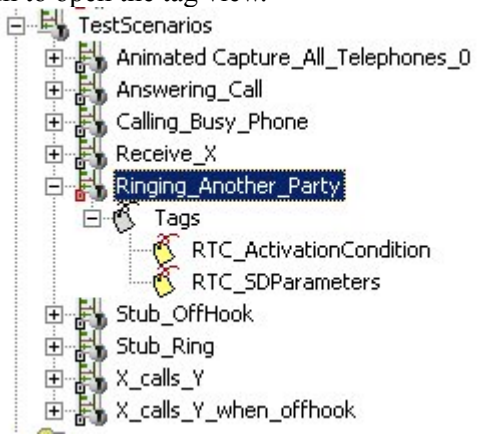
Note: Rhapsody does not perform any static syntax checks on these conditions.

You can associate exactly one activation condition with every sequence diagram. The trivial activation conditions are *TRUE* and *FALSE*. Every sequence diagram instance used in a test inherits the activation condition specified in the property dialog of the sequence diagram.

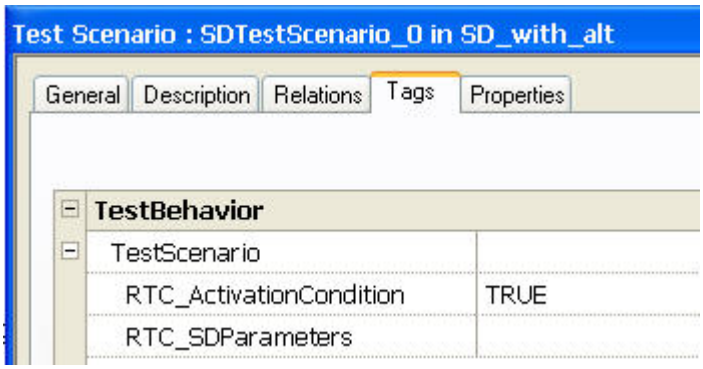
Defining an Activation Condition

Activation conditions are stored as additional tag *RTC_ActivationCondition* in the corresponding test scenario sequence diagram. Activation conditions can be defined with respect to the condition language definition, as follows:

1. In the Rhapsody browser, click on the cross beside of the name of the test scenario sequence diagram to open the tag view.



2. Open the **Feature** dialog of the corresponding RTC_ActivationCondition tag
3. Click into the **Value** field and type the condition. You can specify one activation condition.



5. Click **OK**.

Note: To make activation conditions visible in the sequence diagram, you can draw notes with their descriptions.

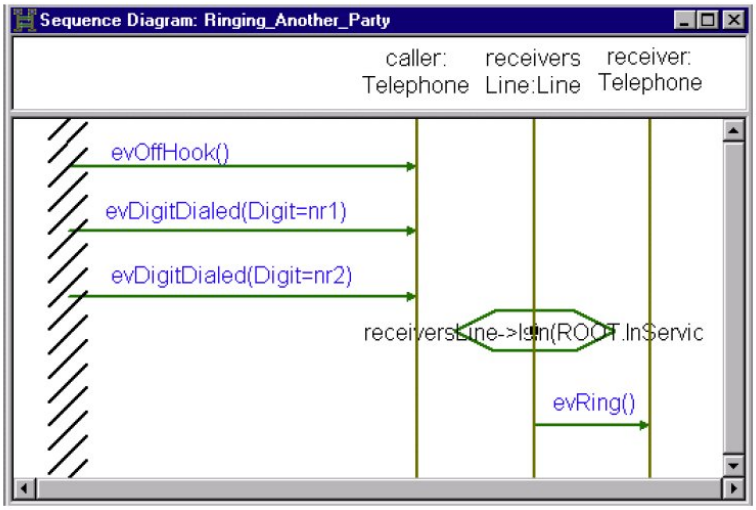
Condition Marks

TestConductor enables you to specify conditions for condition marks on instance lines with the same syntax as activation conditions. Condition marks in sequence diagrams can play the following two roles:

- Synchronize several sequence diagram instances executed concurrently.
- Specify a stubbing behaviour which can appear during execution.

As an example, you can add the following condition mark for the instance of the class `Line` in the “Ringing_Another_Party" sequence diagram:

```
receiversLine->IsIn(ROOT.InService)
```

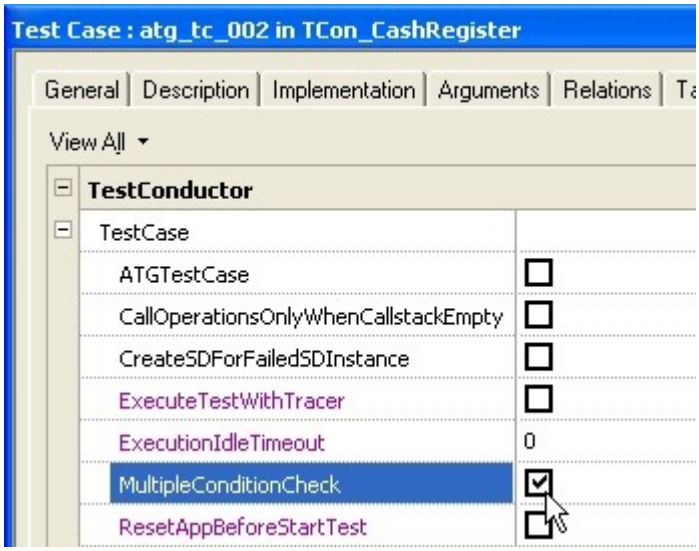


Testing the requirements specified by this sequence diagram, TestConductor will drive the first three events. After that, it will proceed only if the condition of the condition mark has the value `TRUE`. Otherwise, some other activities in the system must be performed to change the value of the condition. You can specify these activities using other sequence

diagrams driven by TestConductor. They can also be driven manually, if it has not been yet implemented as a part of the system. Changing the value of the specified condition to `TRUE` will trigger TestConductor to continue monitoring and driving this sequence diagram.

In case there are two or more condition marks defined in a row, TestConductor will check the first only. TestConductor will evaluate each of the following condition marks with a new system activity, if the previous condition mark was `TRUE`. This is the default TestConductor behaviour.

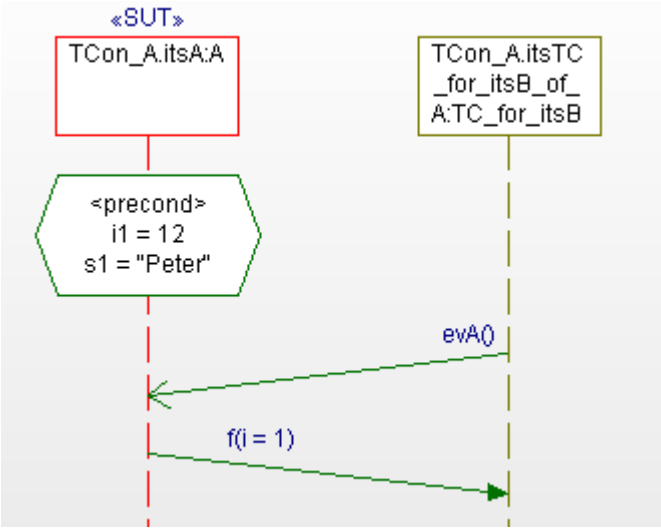
TestConductor can be configured to check the reached condition and following conditions without system activity, till one condition mark evaluates to `FALSE`. To change the default TestConductor behaviour change the property `TestConductor::TestCase::MultipleConditionCheck` of the test case from `FALSE` to `TRUE`.



Note: TestConductor will ignore condition marks during test execution when the syntax of the condition mark is not valid.

Preconditions (for SysML/Harmony)

For SysML/HARMONY models, i.e for SysML models that contain the HARMONY profile, TestConductor provides a special kind of condition, so-called preconditions. With preconditions, in SysML/HARMONY models one can set attributes of SUTs to specifed values. This is useful whenever the behavior of the SUT depends on values of local attributes. In order to define a precondition in a test scenario, add a condition on the life line of the SUT instance that contains the attribute, write "`<precond>`" into the first line of the condition's text, and specify the value the attribute should have in the next line:



In the example depicted above, a precondition is specified that defines value “12” for the attribute “i1” and value “Peter” for attribute “s1” of block A. When executing the test case, and TestConductor reaches the precondition, it sets the specified values for the attributes. When the test case continues, now the behavior of the SUT reflects the new values for the attributes. Currently, the usage of preconditions is restricted to SysML/HARMONY models. If multiple attributes should be set by a precondition, the attribute value specification must be separated by newlines in the condition mark.

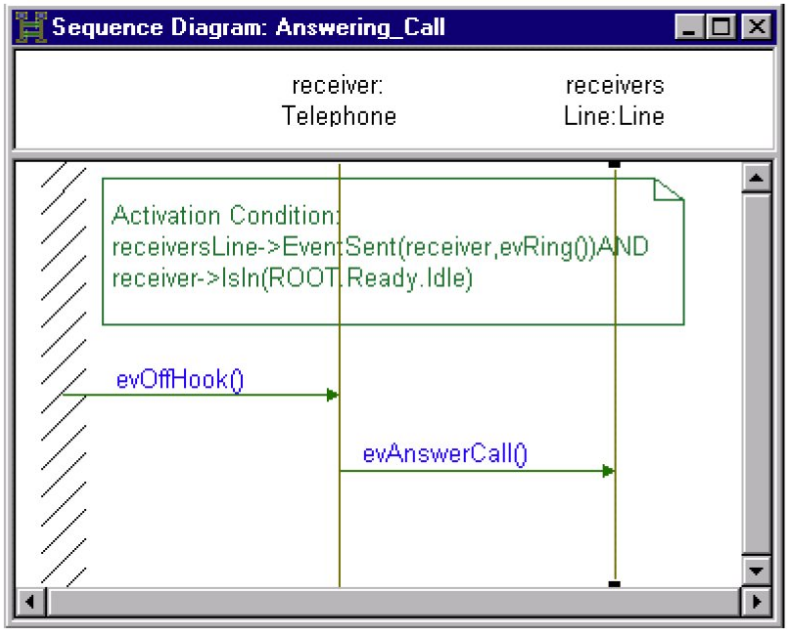
Use Cases of Activation Conditions

This section describes some examples that use activation conditions. The main three purposes of activation conditions are as follows:

- To specify the starting point of sequence diagram simulation.
- To specify that one sequence diagram can be activated only when another sequence diagram has already been activated or fully traversed (during simulation).

Specifying the Starting Point of Simulation

Activation conditions specify a point in time when the corresponding sequence diagrams must be activated. Consider the parameterized “Answering_Call” sequence diagram shown in the following figure:



This sequence diagram can be used to test, whether any telephone can properly answer a call. This property will be checked starting in the system state specified in its activation condition:

- When the object defined as `receiversLine` has sent the event `evRing()` to the corresponding `Telephone` receiver.
- When the object defined as `receiver` stays in its basic state `Idle`.

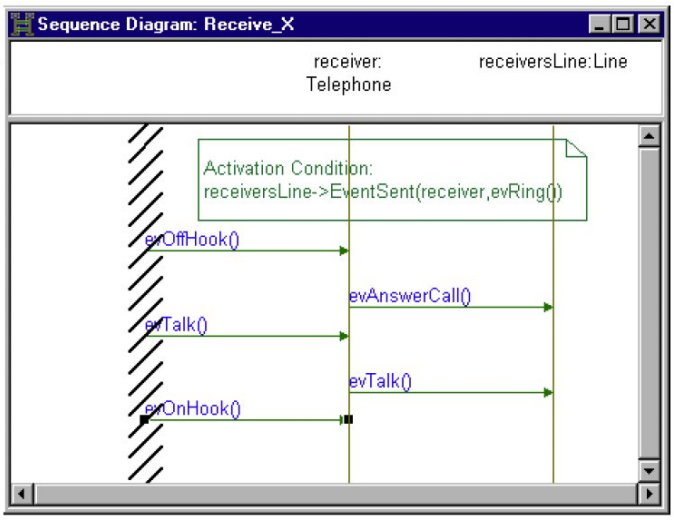
Specifying Ordered Predecessors

Through activation conditions, you can define a *predecessing order* between instances of different sequence diagrams checked during the same test execution.

Example 1: Exact Predecessing

Consider two sequence diagrams that will be stimulated one after another:

- “Ringing_Another_Party” (shown on page *161*)
- “Receive_X”, shown in the following figure:



Note that the exact order can be set only between "concrete" sequence diagram instances, rather than parameterized sequence diagrams. Consider the following parameter mapping for the "Receive_X" sequence diagram:

```
receiver: PBX[0]->itsTelephone[2]
receiversLine: PBX[0]->itsLine[2]
```

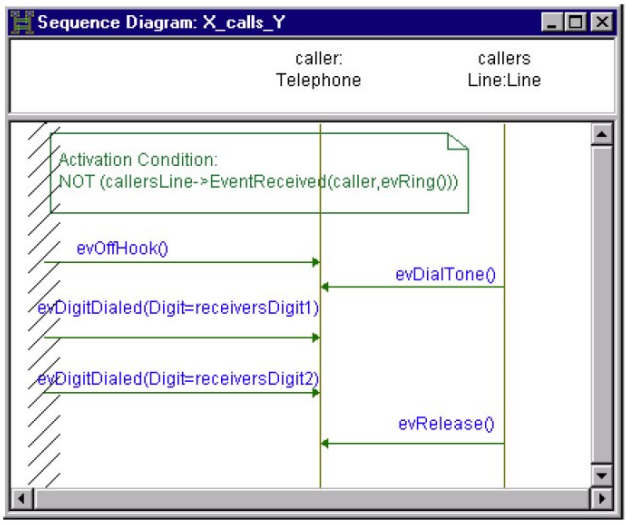
The activation condition of this sequence diagram specifies the starting point when `Line 3` has sent the `evRing` event to its `Telephone 3`. This condition can become `TRUE` when the corresponding instance of the "Ringing_Another_Party" sequence diagram (with the similar parameter mapping) has been fully traversed.

Although the sequence diagrams "Ringing_Another_Party" and "Receive_X" have similar parameter names—`receiver` and `receiversLine`—they can be mapped to different values. In such a case, two sequence diagram instances will be unordered. Therefore, parameter names in sequence diagrams can be considered as local variables with values in the scope of the corresponding sequence diagrams.

Example 2: Interleaving the Execution of Two Sequence Diagrams

The following two sequence diagrams are activated during a test execution one after another:

The "X_calls_Y" sequence diagram, shown in the following figure:



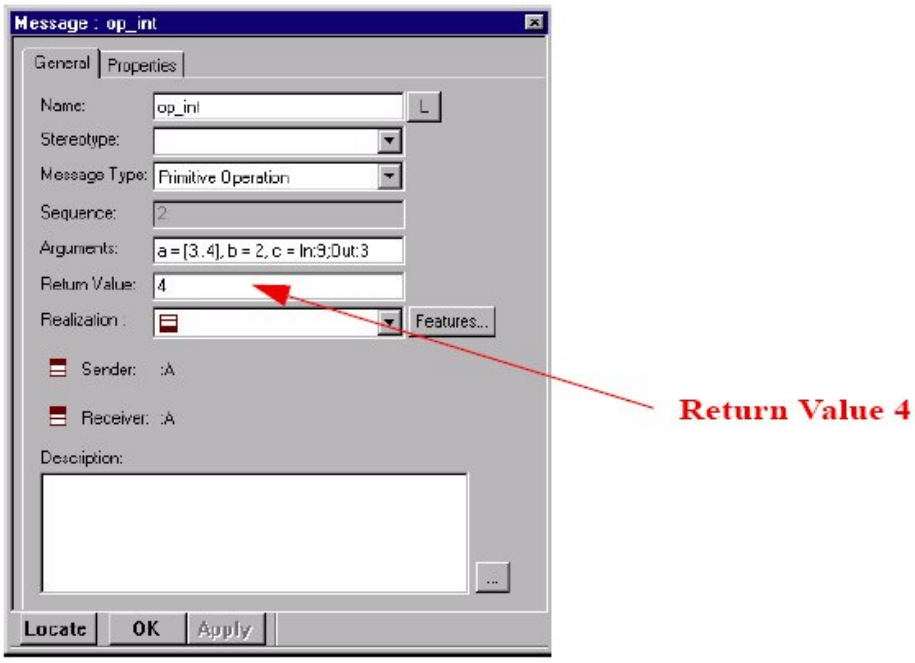
This can be used to test whether any telephone can start and finish a communication. Moreover, this property will be checked only starting from the specified state of the system—when the object defined as `callersLine` has not received the event `evRing` from the corresponding telephone `caller`.

An instance of the “Receive_X” sequence diagram, described on before can be activated after the corresponding instance of the “X_calls_Y” sequence diagram has been partially traversed. To obtain this order between sequence diagram instances, the mapping for the parameters `receiversDigit1` and `receiversDigit2` from the “X_calls_Y” sequence diagram must correspond to the extension number of the `Line` name mapped to the parameter `receiversLine` from the “Receive_X” sequence diagram.

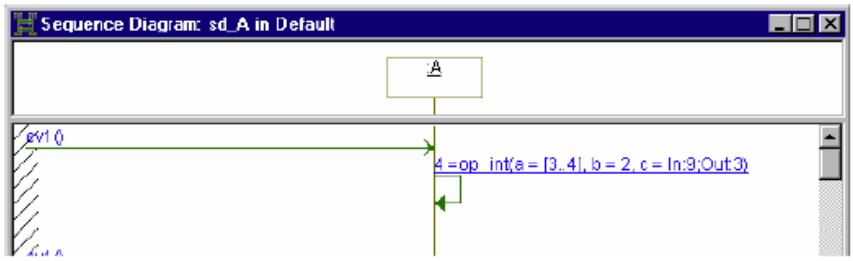
Note that the predecessing order is defined implicitly. During test execution, containing instances of these two sequence diagrams, Test Conductor first activates an instance of “X_calls_Y”, drive the events `evOffHook`, `evDigitDialed`, and monitor the event `evDialTone`. After driving the event `evDigitDialed(Digit= receiversDigit2)`, TestConductor activates the corresponding instance of the “Receive_X” sequence diagram. It monitors the event `evRelease` only after the instance of the “Receive_X” sequence diagram has been fully traversed. The exact order of the sequence diagram instance execution is derived from the system behaviour, but is also bounded by the activation condition.

Specifying Return Values and Output Values

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.



Consider operation `4 =op_int(a = [3..4], b = 2, c = In:9;Out:3)` in the following sequence diagram. It returns integer values. Assume we specify integer value `4` as the return value.



TestConductor will monitor the actual values as specified in the dialog when an operation call returns and will check if the actual return value conforms to the specified value or not.

Note: Using Macro `OM_RETURN()`: TestConductor is using Rhapsody's animation capabilities to perform test execution. If an operation returns a value then this value is by default *not animated* in Rhapsody. In order to get animation information about returning operations it is *mandatory* to use a special Rhapsody macro `OM_RETURN()` instead of statement return() for the purpose of test execution. The macro is pre-defined in "`\Share\LangCpp\aom\aommacro.h`". In the above example suppose that operation body of `op_int(int a, int b, int c)` simply contains one statement „`return 4;`". This must be replaced by `OM_RETURN(4);` to be able to check such return values with TestConductor. Since this special macro is only needed for testing purposed it is already embedded into `#ifdef`-statements. The `#ifdef` statement guarantees that the

macro is only used for testing purposes, while the standard return-statement is used when generating non-animated code.

Note: Using Macro `OM_RETURN_VOID`: If an operation returns with a void value, then TestConductor can check that the return indeed happens when using `OM_RETURN_VOID`.

Note: Using Macro OMREPLY(): Triggered operations returning values is realized using reply().TestConductor can check that the return indeed happens when using `OMREPLY()`.

Note: output parameters of type `uchar` and `long double` are not supported.

Note: range specification for return values (e.g. `"[1..4]"`) are not supported.

If an actual return value does not conform to a specified value, then a red message is drawn. The message is labelled with

`"<Specified operation and its parameter> Operation Call returned - Return value does not match. Expected values are: <Expected operation and its parameter list>”.`

For example:

`“4=op_int(a=1,b=2,c=3) Operation Call returned - Return value does not match. Expected values are: 5=op_int(a=1,b=2,c=3)”.`

Note: If we have pointer types or structures as output and in/out parameter types then serialization functions must be added to the macro in order to be able to test the value with TestConductor.

Note: If we have pointer types or structures as return types then serialization functions must be added to the return macro in order to be able to test the value with TestConductor

Specification of the Output and in/out Values

Suppose we consider an operation `m(int p1, int p2, int p3, int p4)`, where `p1` and `p2` are input parameters and `p3` is an output parameter, and `p4` is an in/out parameter. In a sequence diagram users can specify the expected input parameter values and the expected output and in/out parameter values. Output and in/out Test Execution parameters are realized with call-by-reference. For instance, a sequence diagram message "`m(p1= 3, p2 = 5, p3 = 7, p4 = 9)`" specifies that operation `m()` is called in the model with input values `p1=3` and `p2=5`, and with references to `p3` and `p4`, i.e. `m(3,5,&p3,&p4)`. Note that `&p4` is an in/out parameter and hence is used as an input in the operation `m()`, too. Here, `&p4` provides the value '`9`' for the call. The call returns with value `p3=7` and `p4=?`.

The in/out parameter is specified in a sequence diagram with both input and output parameters. The format of specifying an in/out parameter is

```
<parameter> = In:<in_value>;Out:<out_value>
```

Message "`m(p1 = 3, p2 = 5, p3 = 7, p4 =In:9;Out:12)`" specifies that `m()` is called with "`Input p1=3”, "Input p2=5”, "in/out p4=9”`. Message `m()` returns with "`Output p3=7, in/out p4=12`”. Both values for in/out parameter `p4`, the input part and the output part are specified.

Output value checking can not be done for operations which originate from the environment line and are generated by TestConductor. Checking of output values is supported for all operations that originate from TestComponents, and for all operations that do not start at the environment line and whose called operation uses OM_RETURN to return values to the caller.

Users can record animated sequence diagrams. The animated sequence diagrams trace the parameter values when operations are called, but they do not show the values of output and in/out parameters when operations return. Hence, animated sequence diagrams can not be used to check values of output parameters and in/out parameters. Users have to modify animated sequence diagrams in order to extend it with relevant output information which is not provided by Rhapsody's sequence diagram animation.

Suppose we consider an operation `m(int p1, int p2, int p3, int p4)`, where `p1` and `p2` are input parameters and `p3` is an output parameter, and `p4` is an in/out parameter. An animated sequence diagram might show "`m(p1 = 3, p2 = 5, p3 = *, p4 = 9)`". In order to check output parameter `p3` and the output value of p4 when `m()` returns users must add the required information. Example: "`m(p1 = 3, p2 = 5, p3 = 7, p4 = In:9;Out:12)`".

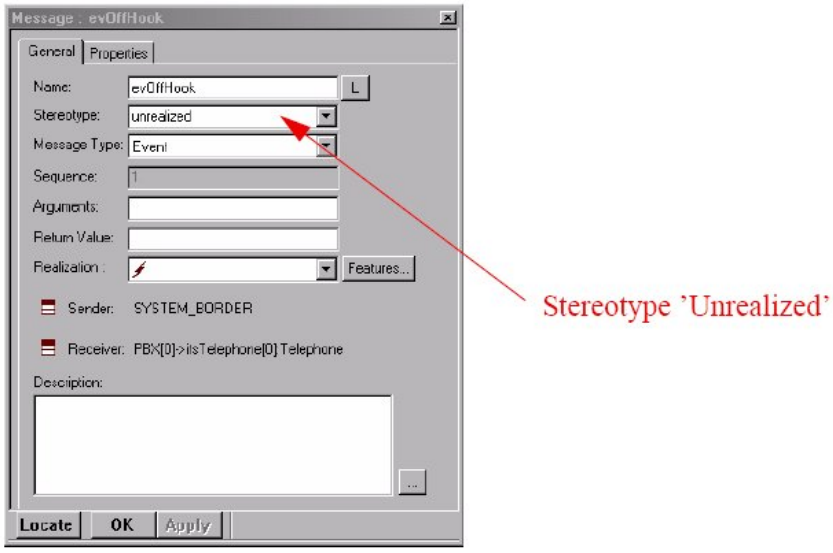
Note: Out or in/out values are only taken into account by TestConductor if also a return value is given in the message specification (value or “don’t care”-star). That must also be done for operations that do not have a specified return type (void operations). Hence, the In:..;Out:… specification should only be used if a return value has been defined, too. Otherwise the test execution will fail.

Note: Out values for some specific out arguments are currently not usable if the corresponding setting of the property CPP_CG::Type::Out specifies a pointer-type instead of a reference-type.

Note: During execution parameter values containing quotes will consistently be stripped, e.g. the expression “OK” will be converted to OK and ““OK”” will be converted to “OK”.

Ignoring Unrealized Messages

Messages with stereotype *unrealized* are filtered out and ignored in the test execution.



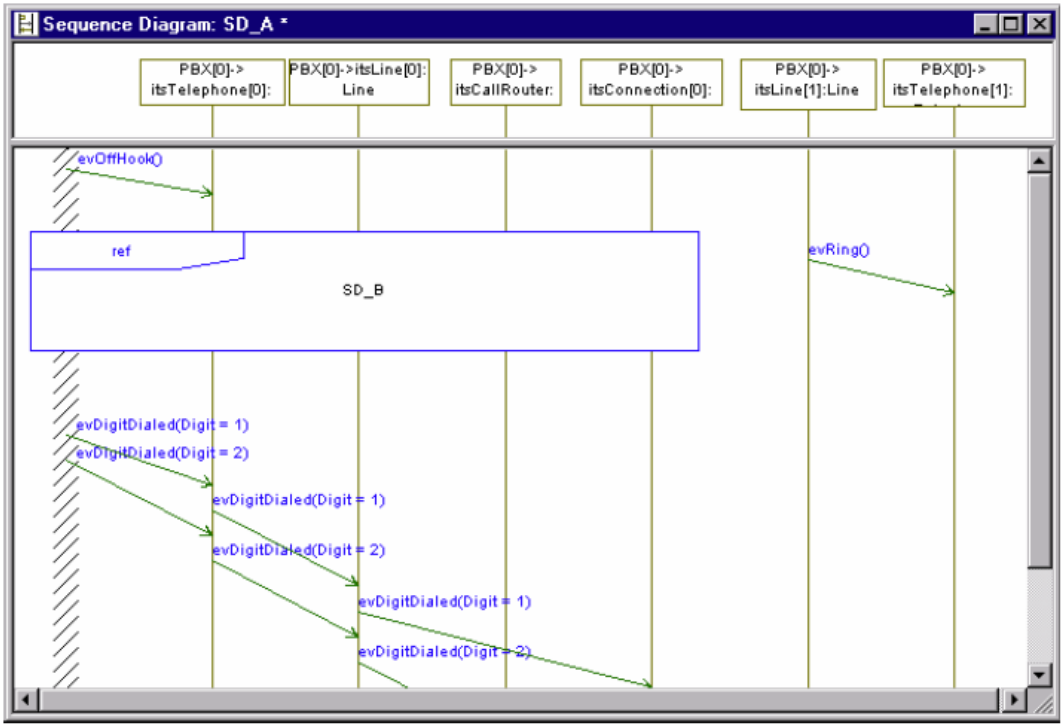
Open the **Features** dialog of the message then specify **Stereotype** as *Unrealized*. When you are executing the test, we get a user warning that the message is ignored in the test execution.

Reference Sequence Diagram

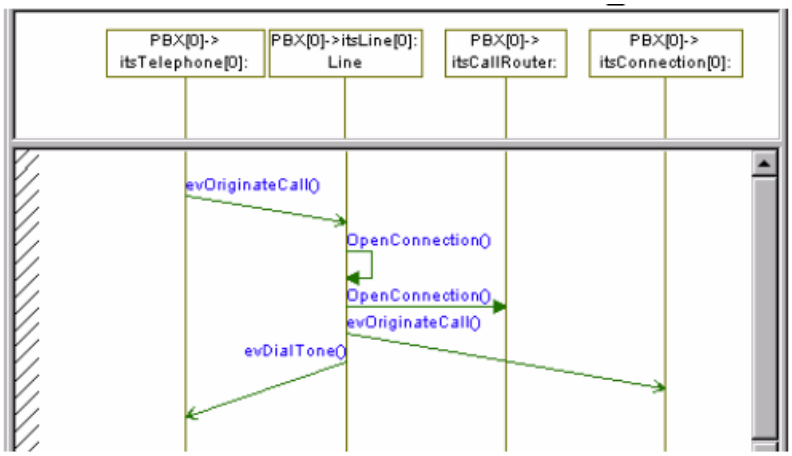
Interaction occurrences and their corresponding reference sequence diagrams are specified within Rhapsody. Defining tests with TestConductor is not affected by interaction occurrences, since interaction occurrences are features inside sequence diagrams, while tests are defined on the basis of sequence diagrams listed in the Rhapsody browser. If sequence diagrams used in a TestConductor test contain interaction occurrences, then this is not relevant for the test definition but it clearly has impact on the test execution.

TestConductor will substitute interaction occurrences with the scenarios specified in the corresponding reference sequence diagrams for test execution. 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 and reference sequence diagrams. Whenever an interaction occurrence is reached, then the scenario as specified in the reference sequence diagram is tested. Test control starts with the main sequence diagram, and when a reference sequence diagram is reached, the control goes into a reference sequence diagram, and as the execution of the reference sequence diagram is completed, the control returns back into the main sequence diagram.

Consider the following main sequence diagram, “SD_A”, which has a reference to the sequence diagram, “SD_B”.



This interaction occurrence refers to a sequence diagram with name “SD_B”, as seen below.



In the sample sequence diagrams above testing sequence diagram “SD_A” with reference sequence diagram “SD_B” leads to the same result as if the interaction occurrence would have been replaced with the scenario in “SD_B”.

The scenario which is going to be tested is:

- `EvOffHook` (SD_A)
- `EvOriginateCall` (SD_B)
- `OpenConnection` (SD_B)
- `OpenConnection` (SD_B)
- `EvOriginateCall` (SD_B)

```
– EvDialTone                (SD_B)
– EvRing                    (SD_A)
– EvDigitDialed       (SD_A)
– EvDigitDialed             (SD_A)
– EvDigitDialed       (SD_A)
...
```

Note: Interaction occurrences are drawn on lifelines. Those lifelines have to be contained in the reference sequence diagram.

TestConductor does not care if:

- reference sequence diagram *does not contain the same life lines* as surrounded by the interaction occurrence
- reference sequence diagram contains *fewer* life lines
- reference sequence diagram contains *more* life lines
- reference sequence diagram contains *other* life lines

TestConductor just considers the provided life lines and the specified messages as relevant test scenario and expects exactly those messages when the SUT is executed. For instance, if the above shown sequence diagram “SD_B” does not contain the life line to the right hand side, then message `evOriginateCall` going to this life line is not part of the test.

Show As SD draws one new sequence diagram with all the messages which have been monitored (*green* colour) or which are supposed to be monitored (*blue* colour), and also failed messages (*red* colour). If a test contains a sequence diagram with one or more interaction occurrences, then TestConductor draws still only one new sequence diagram which shows all the relevant messages of the main sequence diagram and also the messages from the entire referenced sequence diagram.

I case a TestConductor test is executed in linear order a situation which must be taken care of is, when there is an additional message on the same level as of the reference sequence diagram. Consider sequence diagram “SD_A” with the interaction occurrence. To the right hand side of the interaction occurrence there is an additional message `evRing`, which is independent from the interaction occurrence. In partial order execution this will be considered as parallel. In linear order execution, TestConductor must determine a total order on all messages. In sequence diagrams without interaction occurrences, this order is determined graphically from top to bottom in a sequence diagram. In the case above, the graphical order between messages in “SD_B” and between `evRing` is not specified. Hence, TestConductor can not establish a total order based on the graphical information. In this situation, TestConductor follows the following rules:

1. TestConductor considers all messages from top to bottom in total order unless the upper boundary (graphically) of an interaction occurrence is reached.
2. Then all messages in the reference sequence diagrams are considered in total order
3. Then the messages to the right hand and left hand side of an interaction occurrence are considered in total order (if those messages do exist).
4. If reference sequence diagrams contain new interaction occurrences then the same rules apply.

If several interaction occurrences appear in one sequence diagram then the same rules apply, i.e. there is a total order on interaction occurrences which is derived from the graphical order.

If an interaction occurrence is not yet realized by a reference sequence diagram, then this interaction occurrence is ignored for actual test execution.

If reference sequence diagrams are used to specify lifeline decomposition, then this is also ignored by TestConductor for test execution.

Life Line and Part Decomposition

Life Line Decomposition Support for Testing

Life line decomposition and their corresponding reference sequence diagrams are specified in Rhapsody. For instance, consider sequence diagram “MainSD” (Figure 1) which references “RefSD” *(Figure 2)*.

The system border life line specifies the environment of the sequence diagram. Here, we have four messages from the system border going to a logical object `Tel0`. `Tel0` has not been realized to a concrete class or object in the model. It is just a logical name for an arbitrary `telephone (<unspecified>)`. It is a decomposed life line. We set the decomposed life line to “RefSD” as shown in the diagram. Messages `evOffHook, evDigitDialed and evOnHook()` are sent to `Tel0` (the messages are also `<unspecified>`). The `MappingPolicy` property of its life line is set to ObjectAndDerivedFromRefSD.

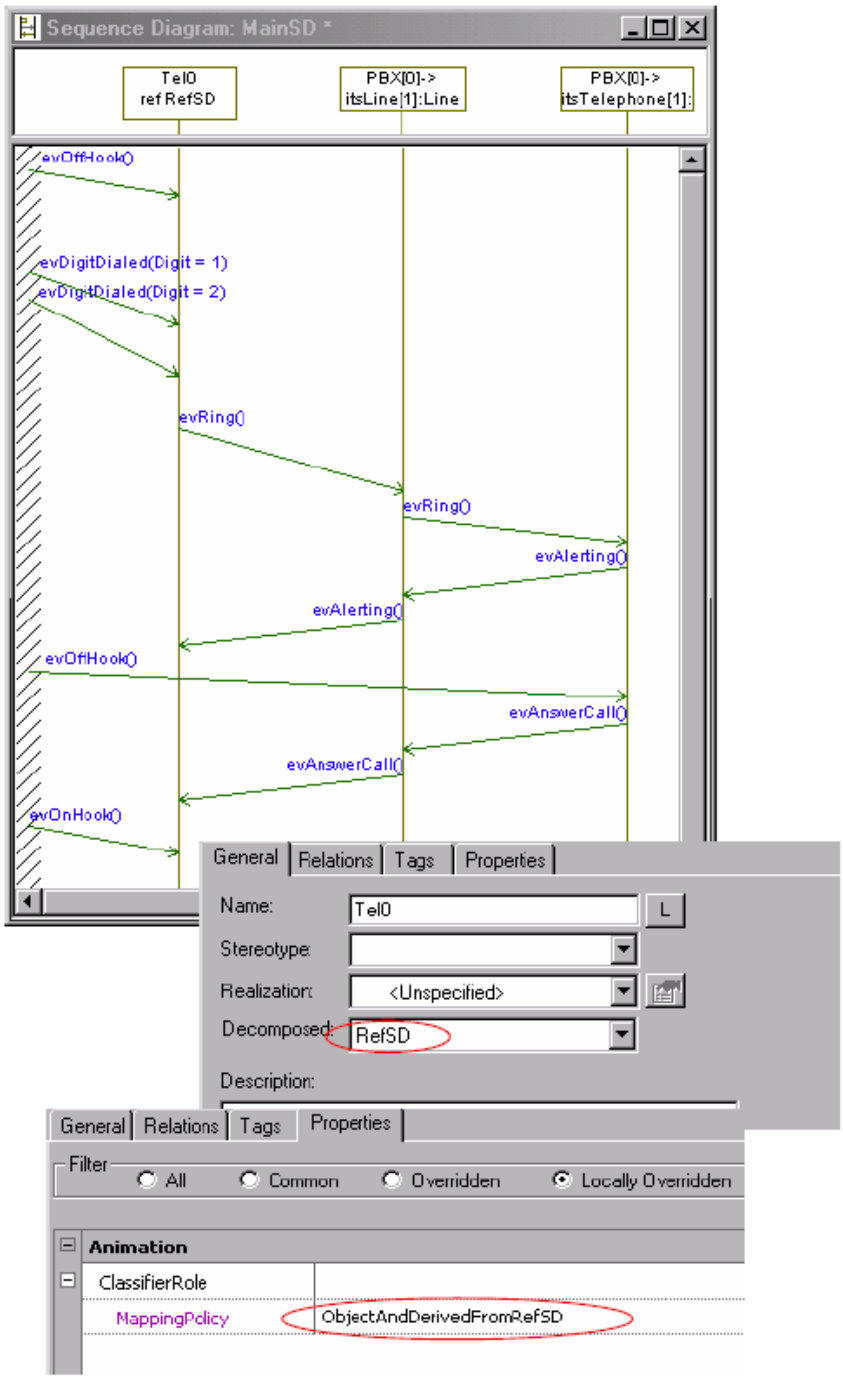


Figure 1 : MainSD

In the "RefSD", we can see that the messages that come from the system border of this "RefSD" do match with the messages in the "MainSD" (`evOffHook()`, `evDigitDialed()`, `evOnHook()`). In the "MainSD", these messages go from the system border to the `Tel0` life line. `Tel0` is internally realized by the concrete objects `PBX[0]->itsTelephone[0]`, `PBX[0]->itsLine[0]` and `PBX[0]->itsConnection[0]` which also exchange some internal messages.

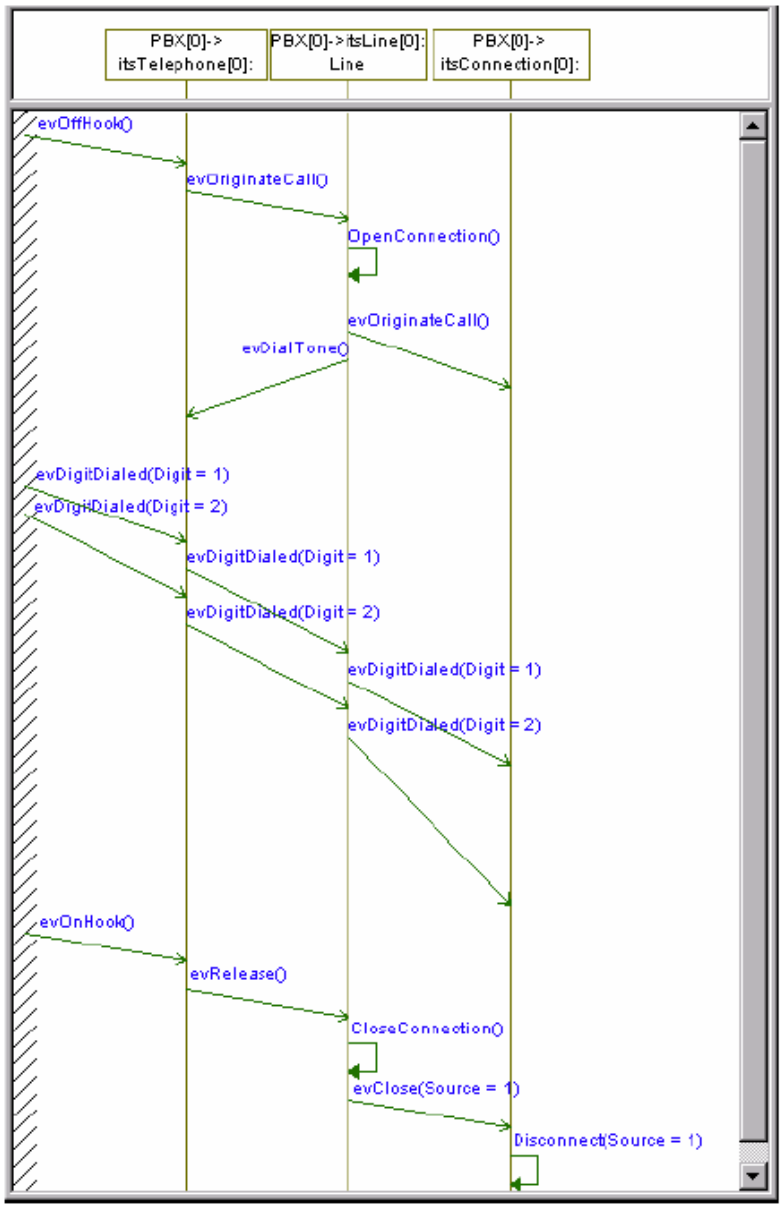


Figure 2 : RefSD

We consider only the “MainSD” while defining the test in TestConductor. For actual test execution, TestConductor will execute the “MainSD” and check if the messages sent to/from `Tel0` in the “MainSD” are received/sent by any of the instances in the “RefSD”. TestConductor knows only senders/receivers of the “RefSD”, i.e., TestConductor knows only the instances in the “RefSD” but TestConductor does not know about the internal messages between the instances in the “RefSD”. When message are sent to/from `Tel0` in the “MainSD”, Testconductor only checks if these messages are received/sent by the instances present in the “RefSD”.

In the sample, testing “MainSD” with reference sequence diagram “RefSD” leads to the following order of messages that will be checked by TestConductor

- System border sends `evOffHook()` to `Tel0` in the MainSD
- `evOffHook()` is received by one of the instances in the RefSD

- System border sends `evDigitDialed(Digit = 1)` to `Tel0` in the MainSD
- `evDigitDialed(Digit = 1)` is received by one of the instances in the RefSD
- System border sends `evDigitDialed(Digit = 2)` to `Tel0` in the MainSD
- `evDigitDialed(Digit = 2)` is received by one of the instances in the RefSD
- `evRing()` is sent by one of the instances in the RefSD
- `evRing()` is received by `PBX[0]->itsLine[1]` in MainSD
- Messages `evRing()` and `evAlerting()` occur in the MainSD
- `evAlerting()` sent by `PBX[0]->itsline[1]` to `Tel0` in MainSD
- `evAlerting()` is received by one of the instances in the RefSD
- Messages `evOffHook()` and `evAnswerCall()` occur in the MainSD
- `evAnswerCall()` sent by `PBX[0]->itsline[1]` is sent to `Tel0` in MainSD
- `evAnswerCall()` is received by one of the instances in the RefSD
- System border sends `evOnHook()` to `Tel0` in the MainSD
- One of the instances in the RefSD receives `evOnHook()` in RefSD

Note: Limitation - Type of message arguments going to decomposed life lines are not known. All arguments are treated as input arguments.

In order to drive messages that are directed to decomposed life lines, a receiver instance must be specified. Open the features dialog of the decomposed life line, click on **Tags** tab, add a new tag *RTC_receiver* (if not available) and also a value like `Telephone[0]` as shown in Figure 3.

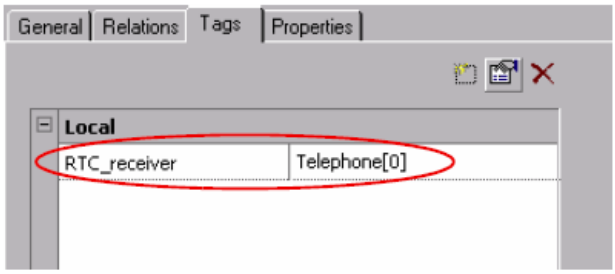


Figure 3 : Features dialog

The following rules are applied by TestConductor in order to drive those messages.

1. If an instance line is not decomposed
   - not realized messages to such a life line are filtered out with a warning
   - if the life line is not realized the test is not executed

2. If a life line is decomposed into *ObjectAndItsParts*
   - if the life line is not realized the test is not executed
   - if the life line is realized then for each driven message the tag RTC_receiver is used to define the proper receiver of the message.
   - if the tag is not defined then the message is sent to the instance the life line is realized to.

3. If an instance line is decomposed into *ObjectAndDerivedFromSD*
   - tag RTC_receiver is used to define the receiver instance of driven messages
   - if the tag is not defined then the message is sent to the instance the life line is realized to
   - if the tag is not defined and the message is not realized then the message is filtered out

4. If an instance line is decomposed into *Smart*
   - if a reference sequence diagram has been defined then see 3.
   - otherwise see 2.

Part Decomposition Support for Testing

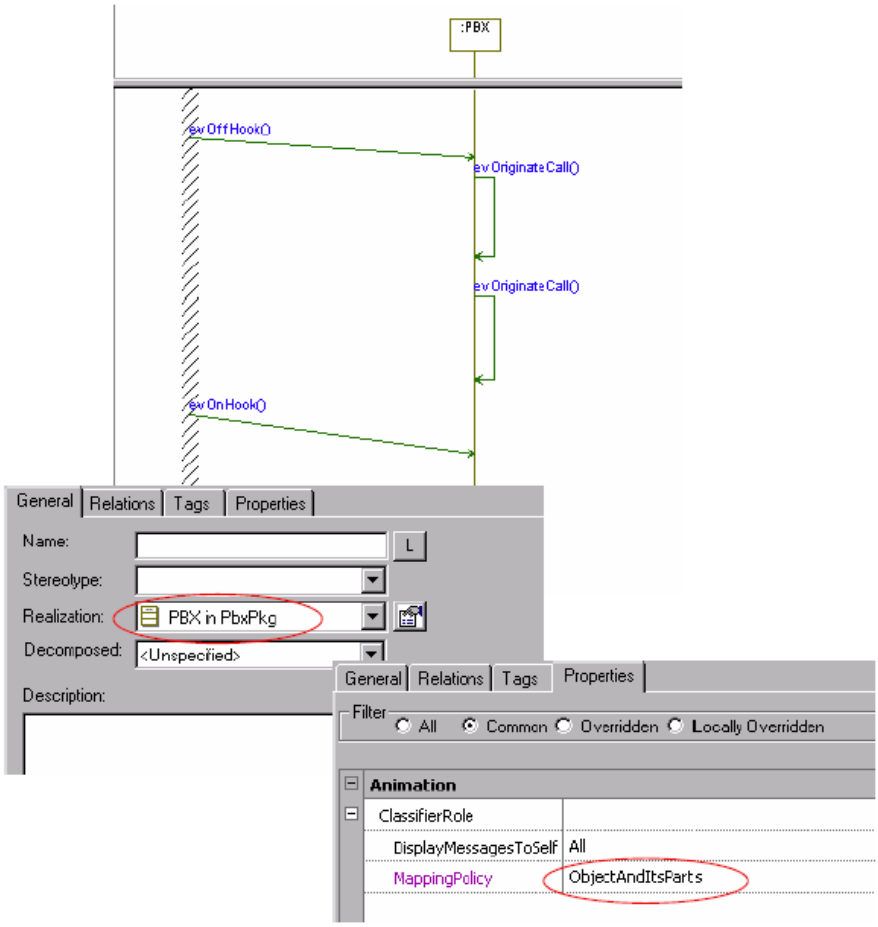
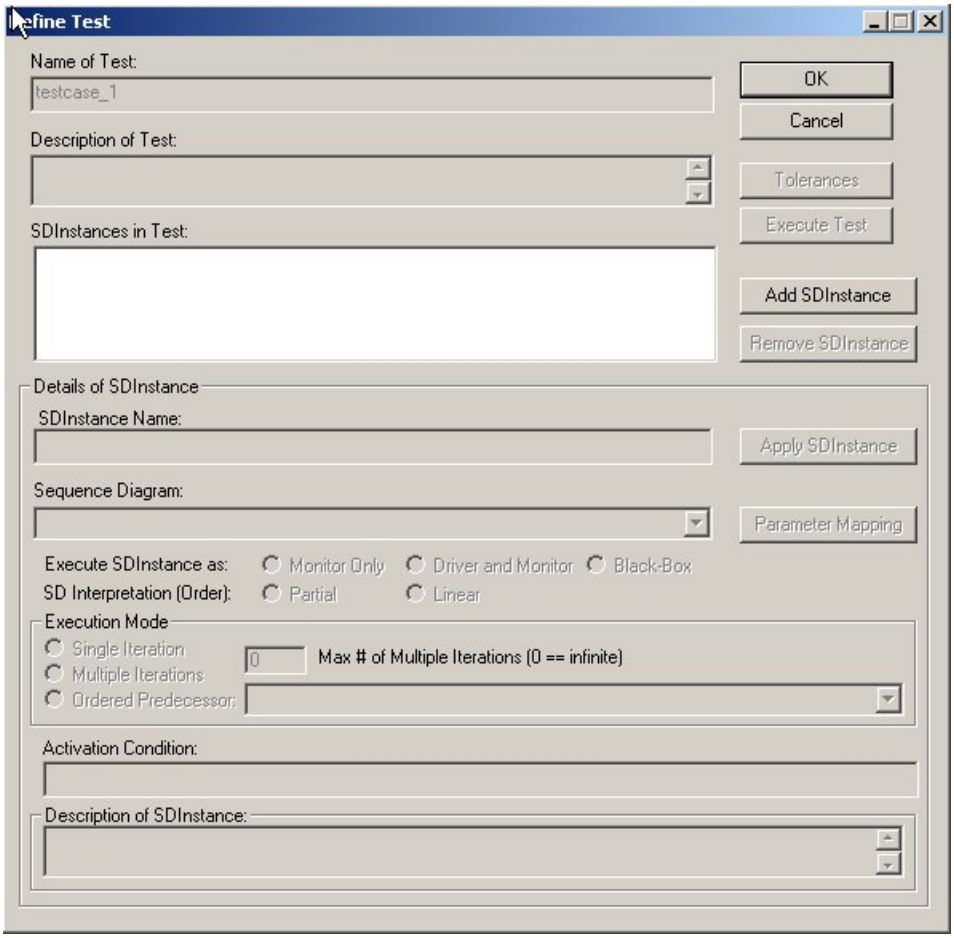


Figure : ObjectSD

Life lines can represent objects and its parts. Consider the Sequence diagram "ObjectSD" above. In the features tab for life line PBX, we have class `PBX` as Realization and ObjectAndItsParts as `MappingPolicy`. Instance line PBX represents object `PBX` and its parts. `evOffHook()` and `evOnHook()` are sent to the parts of `PBX` from the environment. TestConductor treats these messages as going to object `PBX` or any of it parts. `evOriginateCall()` is an internal message of `PBX`, which is sent between the internal parts of `PBX`. In other words, TestConductor takes a black box view for life lines with part decomposition.

Advanced Sequence Diagram Test Definition

The TestConductor test definition dialog enables you to define and configure advanced sequence diagram test cases. Using the dialog box, you can define a name of the test, a description and you can add several sequence diagram instances to the test case. The sequence diagram instances are marked as **Monitor Only**, **Driver and Monitor** or **Black-Box** and parameters are bound to concrete values. In addition, for every sequence diagram instance, you choose the interpretation order (**Linear** or **Partial**) and execution mode. The **Execution Mode** specifies whether the sequence diagram instance must be tested one time or repeatedly in a cycle. You can order sequence diagram instances with **Single Iteration** or in an **Ordered Predecessor** order.



Defining a Sequence Diagram Test

There are four steps in defining a test using the **Define Test** dialog:

1. Create the sequence diagram test case.
2. Define a new sequence diagram instance.
3. Map the parameters.
4. Close the dialog.

The following sections describe these steps in detail.

Creating a Sequence Diagram Test Case

There are three possible ways to define a sequence diagram test case:

1. Right-click on the test context and select **Create SD TestCase**. This creates automatically a new test scenario sequence diagram with lifelines of all classes (SUT and test components) of the test context.
2. Right-click on the test context and select **Add New > TestingProfile > TestCase**.

For the second way you have to use the **Define Test** dialog (shown on page *185*). Use sequence diagrams could be sequence diagrams from the analysis phase, a recorded

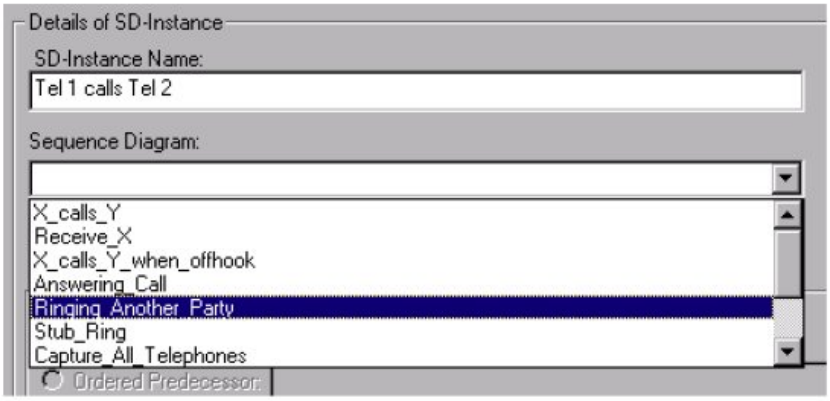
animated sequence diagram from manually driven animation, or a newly drawn test scenario sequence diagram.

Adding a New Sequence Diagram Instance

When you add an sequence diagram instance to a test case definition, you select and reference a sequence diagram from the Rhapsody repository, define a name for that particular instance in the test configuration, and bind the parameters to concrete values (if parameters are used in the sequence diagram). TestConductor automatically extracts the defined activation condition of the referenced sequence diagram from the Rhapsody repository and displays it in read-only mode in the text field.

To add a sequence diagram instance to the list, do the following:

1. In the **Define Test** dialog box, click **Add SD Instance**.
2. The fields **SD-Instance Name**, **Sequence Diagram**, and **Description of SD-Instance**, and the radio buttons **Execution Mode**, **SD Interpretation (Order)**, and **Execute SD-Instance** become enabled so that you can enter data.
3. In the **SD-Instance Name** field, type a descriptive name. For example, “Tel 1 calls Tel 2”.
4. The **Sequence Diagram** drop down list includes all the sequence diagrams from all packages specified in the project. From this list, select one sequence diagram. The following figure shows the list of sequence diagrams for the PBX example.



Note: You do not have to save the sequence diagrams before using them to define and execute tests because the created sequence diagrams are immediately part of the model. The read-only field **Activation Condition** shows the corresponding value for the specified sequence diagram. You can change this value by editing the tag *RTC_ActivationCondition* of the corresponding sequence diagram.

5. In the field **Execute SD-Instance as**, select one of the following options:

   - **Driver and Monitor**—Invokes automatic driving of model execution after the test has been activated. In other words, TestConductor automatically injects events into the running Rhapsody model according to the specified sequence diagram.
   - **Monitor Only**—Invokes manual driving of model execution. This means that, during test execution, you must inject input events manually using the Rhapsody animation tool or the project GUI (when available). TestConductor monitors the reception of these events and internal messages between system objects.

- **Black-Box**—Considers only those messages that originate at the system border (to be driven by TestConductor) or that go to the system border (to be monitored by TestConductor). The remaining messages are not considered because they are internal to the system.

6. In the field **SD Interpretation (Order)** select one of the following options:

   - **Linear**—Specifies that TestConductor should monitor the sequence diagram under test assuming that all events and messages are arranged in a strict sequence. The vertical drawing order of messages in sequence diagrams is used to compute an absolute sequence of events and messages (each message in the sequence diagram has a unique predecessor and successor).
   - **Partial**—Specifies that TestConductor should monitor only the order of events located on the same line (instance line or message arrow).

     Note that partial order set together with driver and monitor implies that driving the input events is independent from monitoring the internal messages. To avoid the arising nondeterminism, TestConductor first drives inputs and then monitors internal messages. **TestConductor chooses one valid order of messages to be driven (in particular, this order changes in general when the same sequence diagram test case is executed repeatedly).** Such nondeterminism does not exist for linear order interpretation, because it is a precise order between all messages in a sequence diagram. See chapter Linear and Partial Order (on page *158*), for the explanation of partial order. Note that there is no nondeterminism for monitor only, because you decide when you inject all inputs, and TestConductor monitors internal messages as they appear in the running model.

7. In the **Execution Mode** field, select one of the radio buttons:

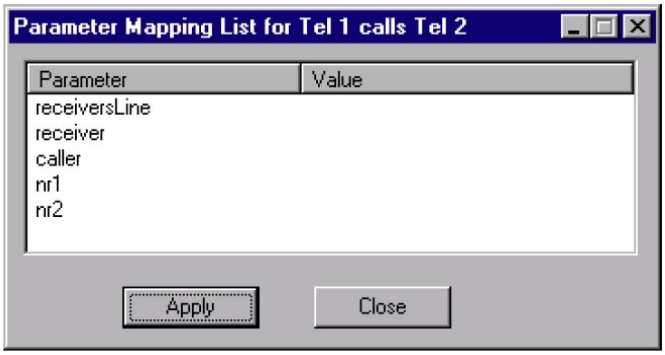
   - **Single Iteration**—Drives the sequence diagram instance only once. TestConductor will generate only one run-time instance of the sequence diagram.
   - **Multiple Iteration**—Drives the sequence diagram instance in a cycle. This option is defaulted to 0 which implies infinite execution of an sequence diagram instance if the activation condition of the corresponding sequence diagram is set to `TRUE`. When a concrete number is supplied here, it implies the number of times the sequence diagram instance will be executed. In batch mode execution, the number 10 helps to avoid infinite looping of tests.
   - **Ordered Predecessor**—Specifies the execution order between two sequence diagram instances. From the drop-down list, select an available sequence diagram instance that must be executed before the current sequence diagram instance is activated.

8. If desired, specify a description in the **Description of SD-Instance** field. This field does not influence test execution, but can be used to describe the purpose of the specific sequence diagram instance.

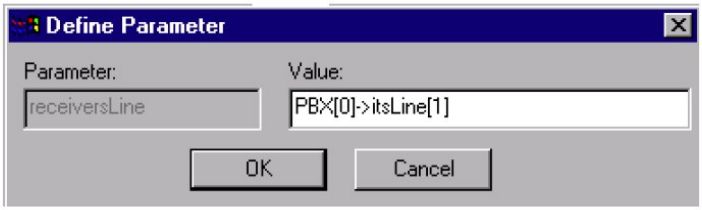
Mapping Parameters

For a parameterized Rhapsody sequence diagram, map its parameters to concrete values as follows:

1. Click **Parameter Mapping** to display the parameter mapping list for the sequence diagram. For a “concrete” sequence diagram, this list is empty. The following figure shows the parameter list for the `Tel 1` calls `Tel 2` sequence diagram.

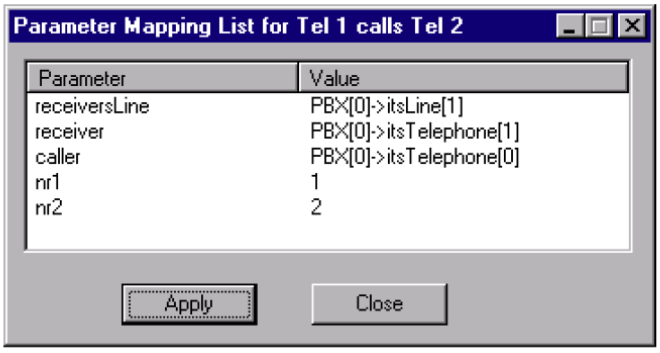


2. Double-click on the name of the parameter to map. The **Define Parameter** dialog is displayed, which enables you to bind the parameter to a concrete value in the current sequence diagram instance.
3. In the **Value** field, type an object name of the corresponding class, or a value for a message argument.



Click **OK** to add the specified parameter value to the list of the parameter mappings or click **Cancel** to discard the changes.

3. Repeat *Step* 2 and *Step 3* to bind all the parameters in the list to concrete values. The following figure shows the completed list.



5. Click **Apply** to bind the values to the parameters and dismiss the dialog, or click **Close** to dismiss the dialog without binding the parameters to new values. You return to the **Define Test** dialog.
6. To add the current definition of the created sequence diagram instance to the test, click **Apply SD**. The sequence diagram instance is accepted as part of the test configuration.

If you do not apply the instance to the test, but continue with another sequence diagram instance, TestConductor automatically applies the first instance for you. If you dismiss the complete test case definition dialog, the sequence diagram instance definition is discarded.

Note: For each sequence diagram in the repository, you can add many sequence diagram instances to a test (for example, with different parameter values). At any time, you

can easily modify any of the information specified for a given test. For example, you could add other sequence diagram instances, or specify another instance testing mode.

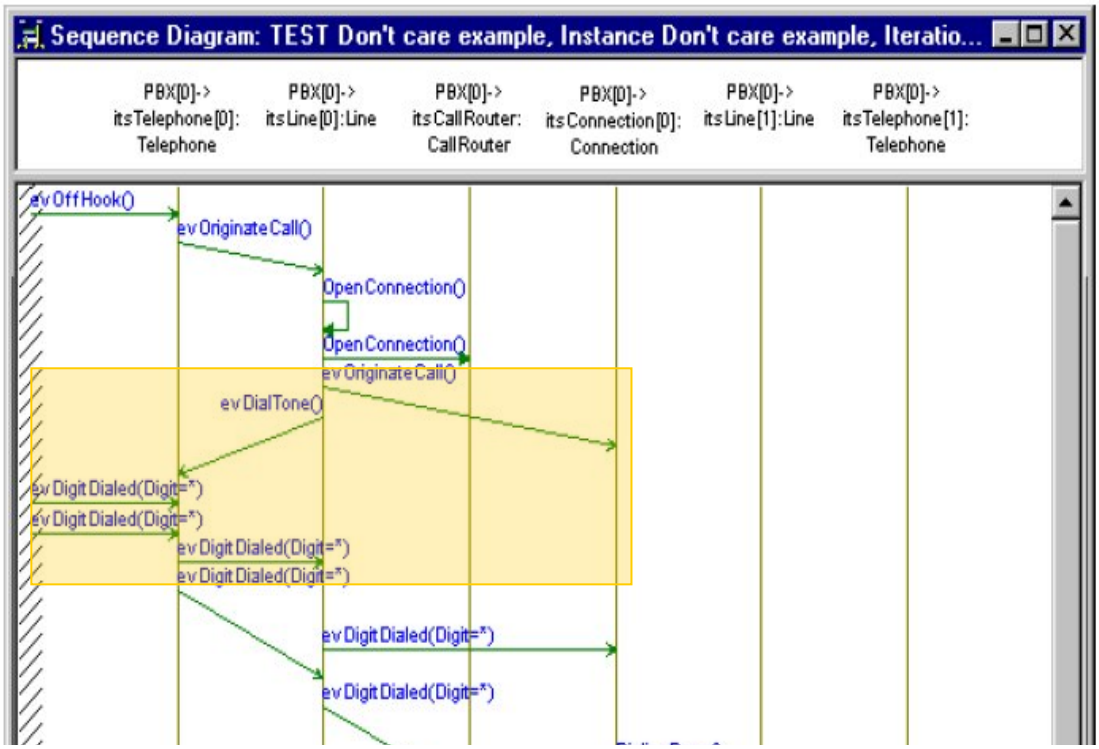
Don't care values, Ranges, and Tolerances

Don't care values

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

- Messages carry values that change whenever you re-run your application (sensor values, etc.). TestConductor should not compare the actual values with the specified values.
- Message parameter is a pointer to a structure. TestConductor can not compare the actual values in the structure.
- Some specific parameter values are not interesting at all for your test. You can switch on/off monitoring and checking of actual parameter values. For every message playing a role in your test you specify don't care either
- For a whole test, or
- For a single message instances in the used scenarios.

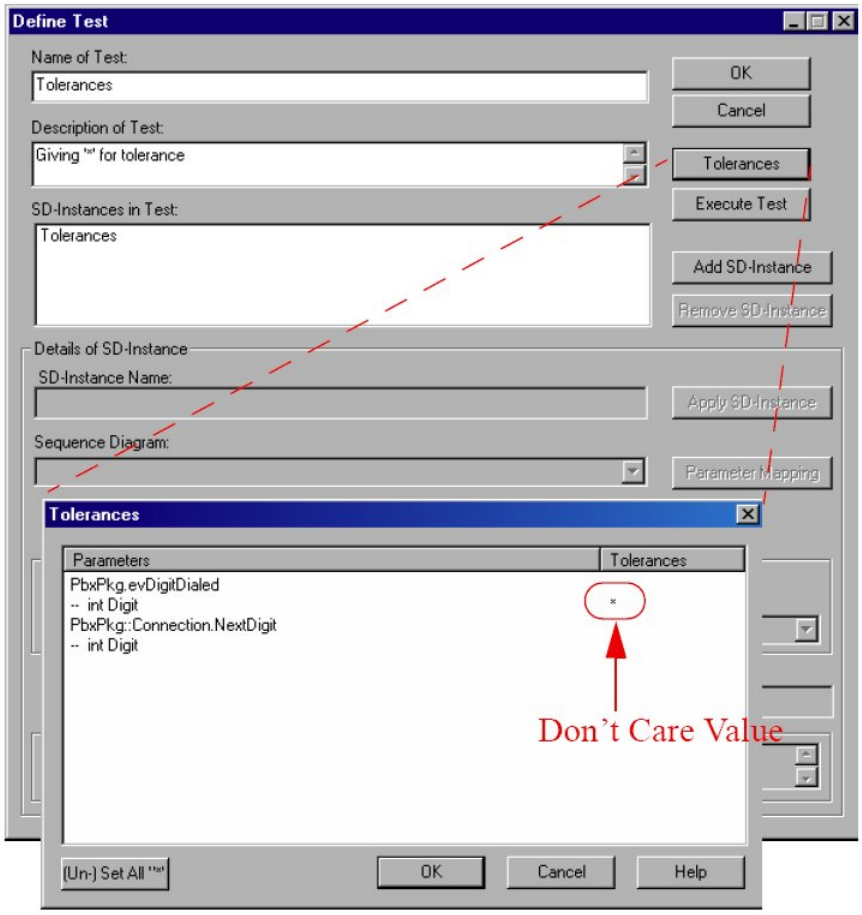
You can even switch on/off monitoring of parameter values for every single parameter of a message



To specify tolerances as don't care values:

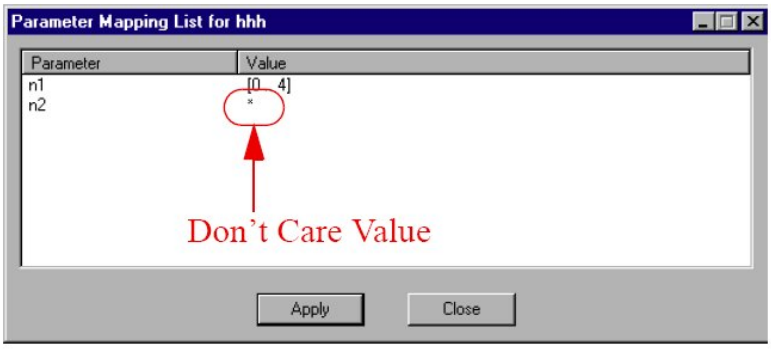
- Replace the parameter values for message instances in the sequence diagrams with the '*' symbol (see picture above), or

- Press the **Tolerances** button within the **Define Test** dialog



- The table lists all messages of all sequence diagrams used in the test
- The don't care values in the table 'override' concrete values in sequence diagrams
- Double-click on a parameter to set/unset '*' for the parameter
- Double-click on a message to set/unset '*' for all parameters of the message
- Click on **(Un-)Set All "*"** to set/unset '*' for all parameters of all messages
- Don't care information are stored with the test
- **Show As SD** also shows use of don't care values

Don't care '*' can also be assigned to the variables used in sequence diagrams. Open the parameter mapping window and assign '*' to the variables which you want to set as don't care which is equivalent to specifying '*' in the sequence diagram.



Note: Do not use '*' for messages that are driven by TestConductor!

Note: You must not inject an event into your application with '*' as value for an input parameter

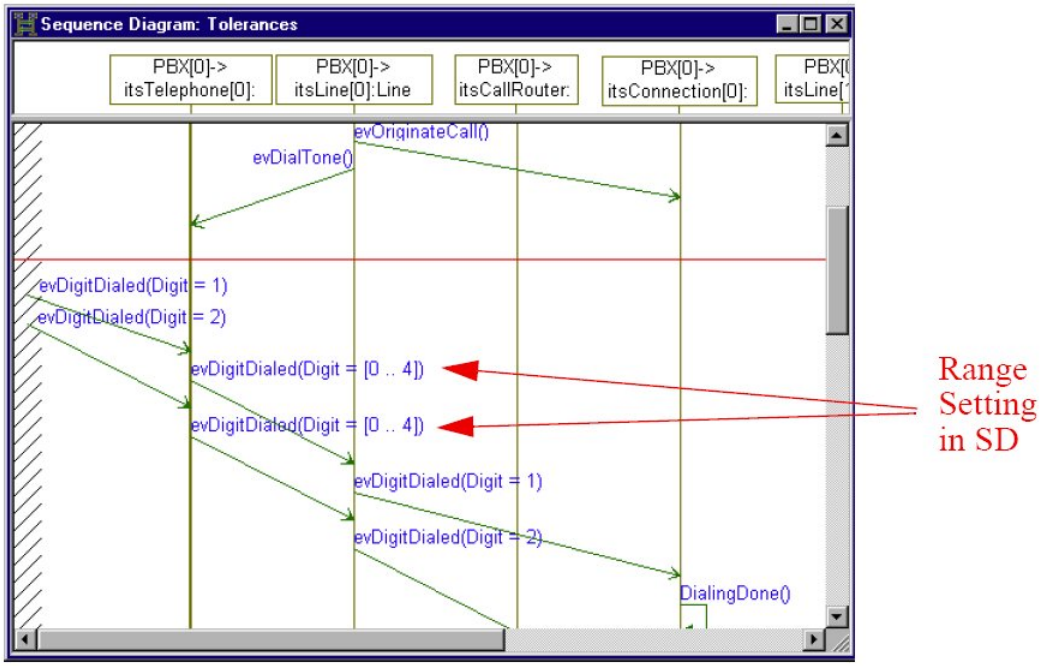
Range Setting

Range setting allows monitoring and checking if concrete values of message instances are in a given specified range. Checking ranges is required if messages have parameters that carry values which deviate from run to run. Speed and temperature are good examples 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. Users must be able to specify that they do not care about specific single values, but about certain value ranges throughout testing. Similar to 'don't care' settings shown in the previous section, we use the same **Tolerances** dialog to specify the ranges also.

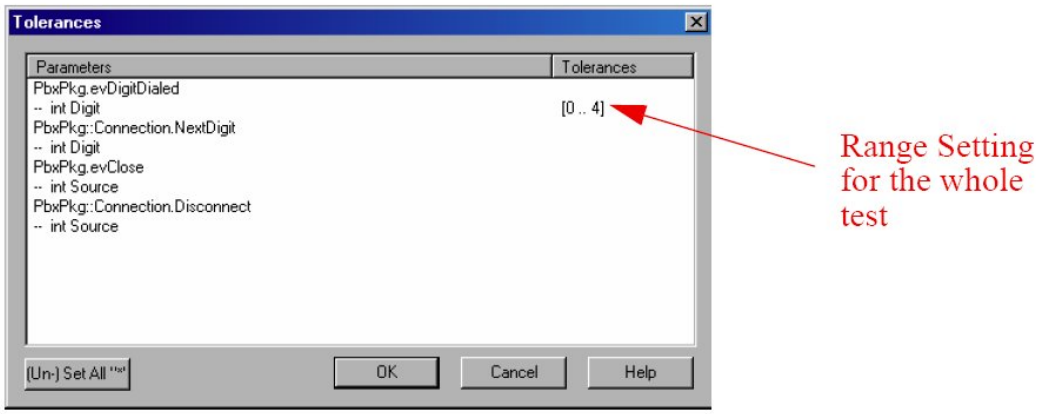
- For every single message instance in a sequence diagram users can specify which parameter should be treated as range of values. A special notation will be used to indicate ranges instead of specific values. Notation:
  `[<lower_value> .. <upper_value>]`

Users can express "`m(p1=1, p2=*, p3=[1.5 .. 1.7])`" to state that `p1` must equals '`1`', `p2` is "don't care", `p3` must be in the range between '`1.5`' and '`1.7`'. In the PBX model, we could use the range of `[0..4]` for the digit of the message `evDigitDialed` in specified sequence diagram.

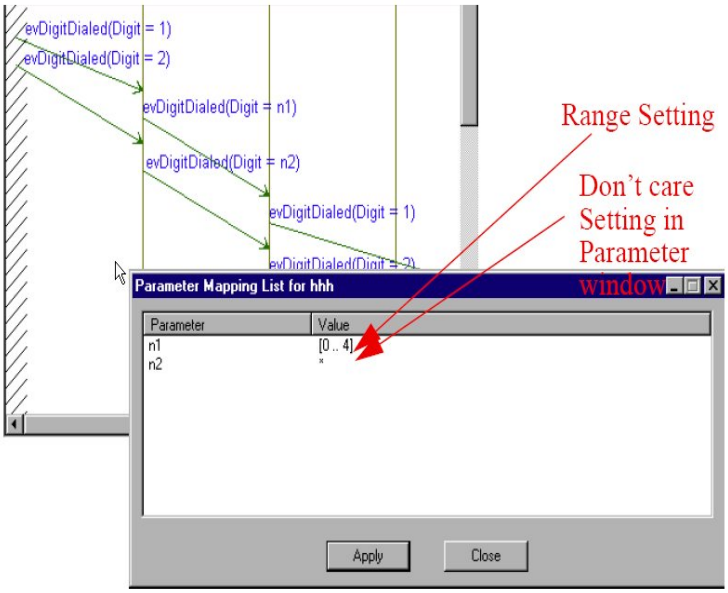
Note: *lower_value* and *upper_value* may be of scalar types like `integer`, `long`, `double` etc.



Alternatively, users may want to specify one specific range of values for a given message parameter for a whole test. This might for instance be desired if a certain measured sensor value globally must be in a certain range. E.g. a measured temperature must always be in the range between 0 and 100 degree celsius. Otherwise it is considered to be an error. For the PBX model, we set the range of `[0..4]` for the digit of the message `evDigitDialed()` in the **Tolerances** dialog as shown below.



The range for the messages which has a parameter as a variable can also be specified in the parameter mapping dialog as shown in the figure below. If we have `n1` and `n2` as variables in the sequence diagram, we can set the range for variables in the parameter setting dialog.



Tolerances

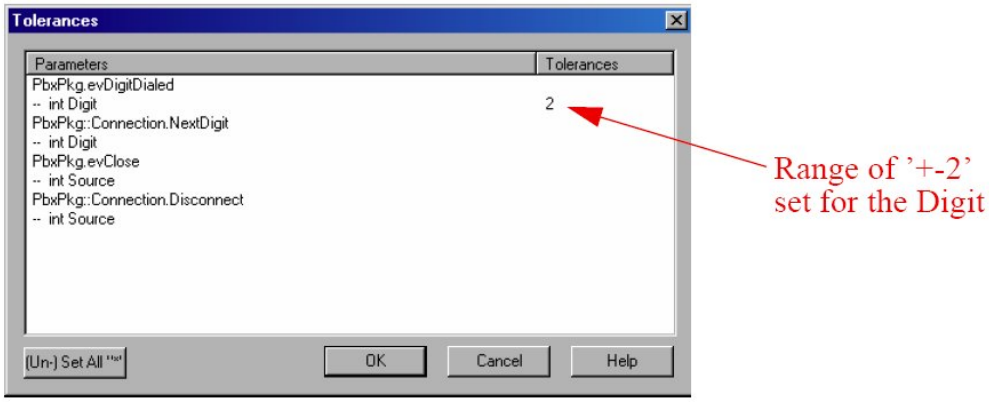
Users may want to specify a tolerance for a message parameter for the whole test. Suppose that a model contains a message `M(temperature p)`. In a recorded animated sequence diagram several instances of `M` might occur, because temperature is measured periodically. E.g. `M(p=27.6)`, `M(p=29.2)`, `M(p=31.1)`, etc. If such a recorded sequence diagram is used for a test, the user must either manually specify a range of values for every single message instance of `M` in the recorded sequence diagram, e.g. `M(p=[27.4..27.9])`, `M(p=[29.0..29.8])`, `M(p=[31.0..31.5])` or we could define a global tolerance for parameter `p` of message `M` in the whole test, e.g. "`p = +-0.5`", meaning that the concrete values in the message instance might have a deviation of '`±0.5`' from the specified values.

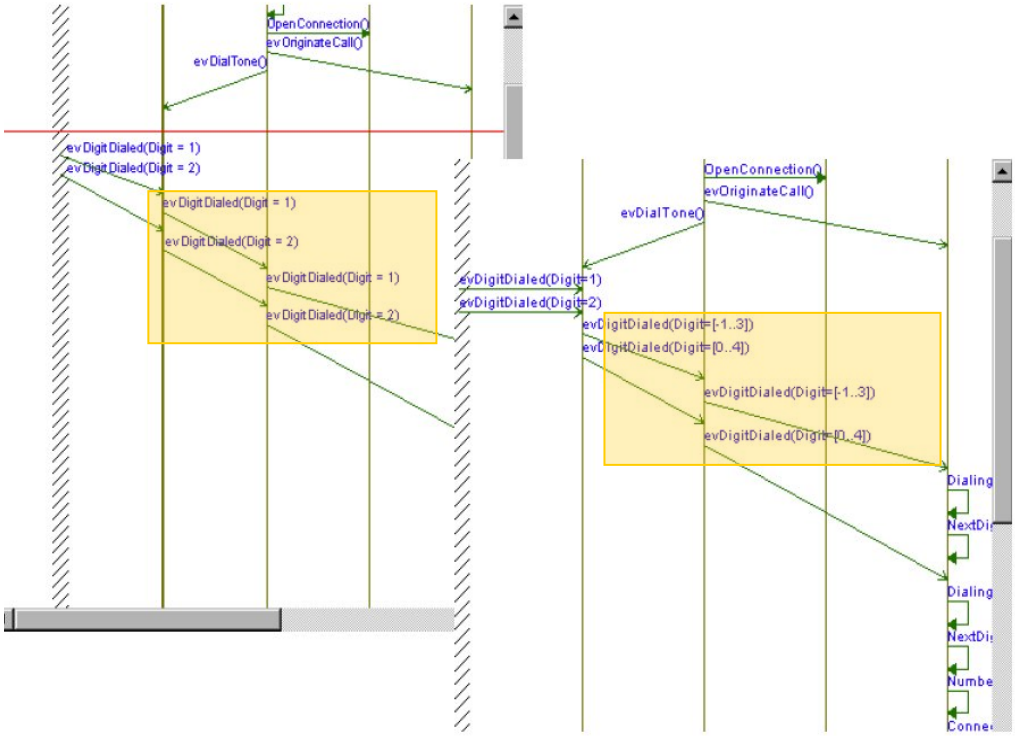
Note: Tolerances can be specified on a per test basis in the table. Users cannot specify parameter tolerances in the sequence diagram.

Note: Tolerances cannot be specified in the parameter mapping dialog.

Note: Tolerances apply to both the parameter values and to parameter ranges.

Setting the tolerance of '`+-2`' for the parameter digit in the PBX model is shown in the following figure. Message `evDigitDialed(Digit = 1)` is seen by TestConductor as `evDigitDialed(Digit = [-1 .. 3])`, which is a range of '`±2`' and `evDigitDialed(Digit = 2)` is seen by TestConductor as `evDigitDialed(Digit = [0 .. 4])`, which is a range of '`±2`' as specified as the tolerance.





Priority rules for the Tolerances

TestConductor will apply priority rules on the parameter values for test execution in the following order:

1. If in the Tolerances table a parameter is set as don't care '*' this will be applied for test execution
2. If don't care'*' is set in the sequence diagram, this will be applied
3. If a range of values has been specified in the **Tolerances** dialog, it will be applied for test execution
4. If a tolerance has been specified in the **Tolerances** dialog this will be applied for test execution
5. Range setting in the parameter mapping dialog or the range setting in the sequence diagram will be applied.
6. Next the value setting in the parameter mapping window or values as specified in the sequence diagrams are used for testing

Note: Value ranges and tolerances can not be applied to messages driven by TestConductor, since driving always requires concrete values.

Note: Value ranges and tolerances can be used only for pre-defined scalar types `int`, `long`, `float`, etc. such that TestConductor can apply standard compare operations (`<`, `>`, `=`) for the checking.

Note: Ranges of values and tolerances can not be applied to structured types or user defined enumeration types.

Syntax for Tolerances

The syntax for specifying don't care values, range values and tolerances is as follows:

- Don't care: `*`
- Range value: `[<lower_value> .. <upper_value>]`
- Tolerances: `<tolerance_value>`

where *lower_value* and *upper_value* and *tolerance_value* can be of pre-defined scalar types `int`, `long`, `float`, etc. such that TestConductor can apply standard compare operations (`<`, `>`, `=`) for the checking. While don't care values and range values can be specified in specification sequence diagrams, in the **Parameter mapping** dialog and in the **Tolerances** dialog, tolerance values can be specified only in the **Tolerances** dialog.

Exiting the Define Test Dialog Box

There are two ways to exit the **Define Test** dialog:

- Click **OK** to save the test.
  If you click **OK**, TestConductor automatically adds all your test modifications to the current model.
  Alternatively, you can add the current test to the model and exit the editor by pressing Enter, but only if the **Description of Test** and **Description of SD-instance** fields are not currently active. If you press Enter in the description fields, it adds a line-feed in the description.

  Note that the TestConductor dialog accepts any test definition, even if it is incomplete (for example, you did not specify a sequence diagram instance). If you try to execute an incomplete test configuration, TestConductor displays an error message.
- Click **Cancel** to discard the test.
  To ignore all changes made during the test definition session, click **Cancel**. TestConductor prompts you to confirm the lost changes; click **Yes**.

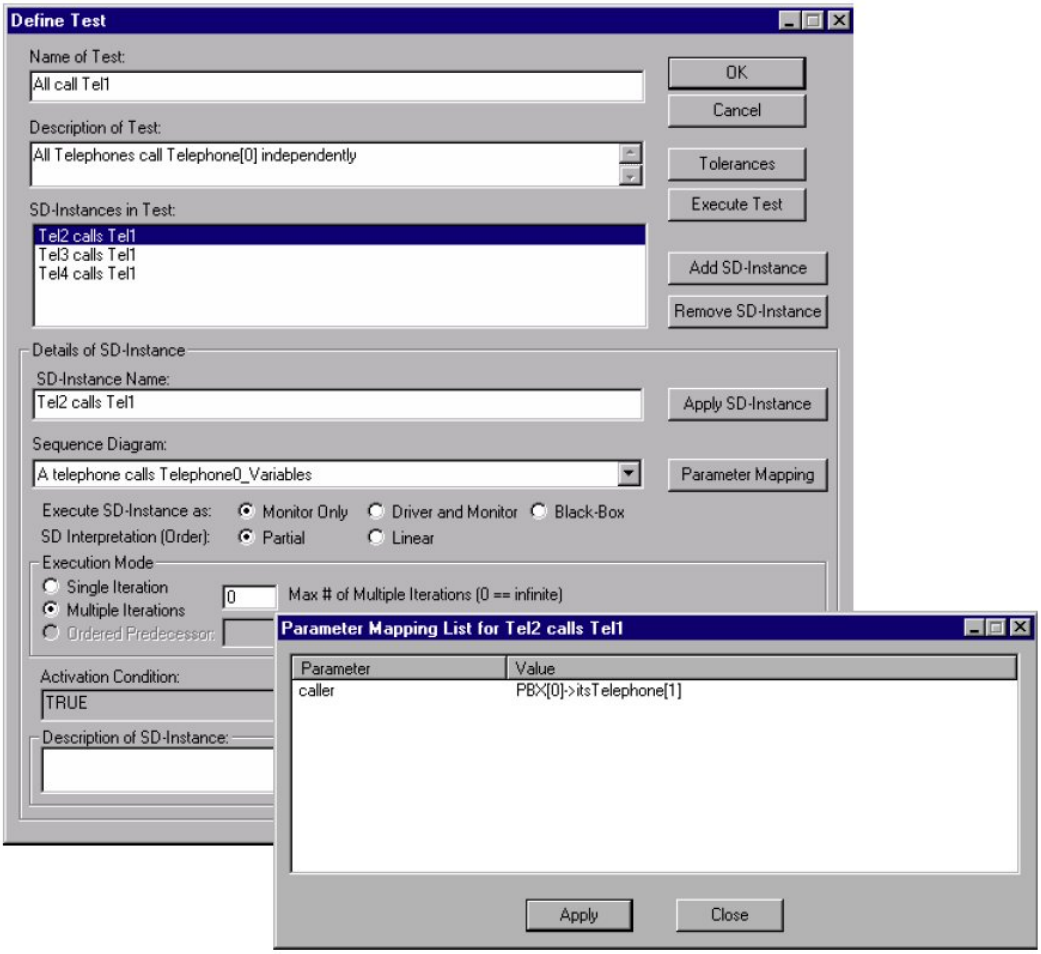
Note: It is not possible to execute tests directly from the **Define Test** dialog.

Use Cases of Sequence Diagram Test Cases

This section shows some sample test cases including different combinations of sequence diagram instance settings (execution mode, sequence diagram interpretation order with monitor or driver), as well as combinations of different sequence diagram instances to be executed in one test with different modes.

Simple Monitor

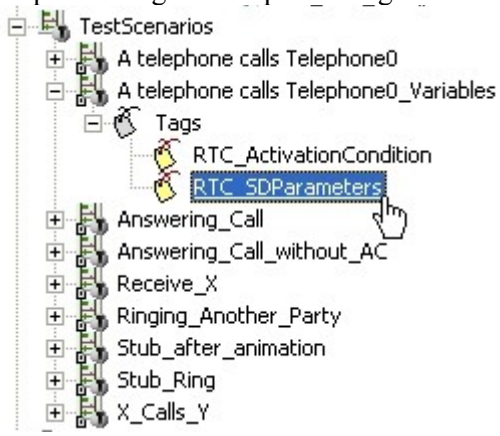
This example explains how to define a simple watchdog. The following figure shows a test configuration with independent sequence diagram instances to be driven manually, infinitely many times. TestConductor monitors whether the computed order of messages corresponds to that specified in the sequence diagrams.



To define this watchdog, do the following:

1. Modify the “A telephone calls Telephone[0]” sequence diagram to make it generic:
   - In the sequence diagram editor, replace the concrete object name `PBX[0]-> itsTelephone[1]:Telephone` with the parameter `caller:Telephone`.

2. Select in the Rhapsody browser the test scenario “A telephone calls Telephone0_Variables” and click on the cross beside of the name of the test scenario sequence diagram to open the tag view.



3. Open the **Feature** dialog of the RTC_SDParamters tag
4. Select the **General** tab, click into the **Value** field and type `caller`, the name of the parameter.

5. Apply the changes and close the **Feature** dialog

To define a new test case and connect the sequence diagram, do the following:

6. Select the test context and choose from the context menu **Add New > TestingProfile > TestCase**
7. Rename the newly created test case to “All_call_Tel1”
8. Select the test case “All_call_Tel1” and choose from the context menu **Edit TestCase SDInstance**
9. Verify the name of the test “All call Tel1” and add the description “All telephones call Telephone[0] independently.”
10. Click **Add SD-Instance**. Type the name of the sequence diagram instance “Tel2 calls Tel1” and select the sequence diagram “A telephone calls Telephone[0]” from the drop-down list.
11. Select the following radio buttons:
    Monitor Only execution
    Partial order, to set manual driving

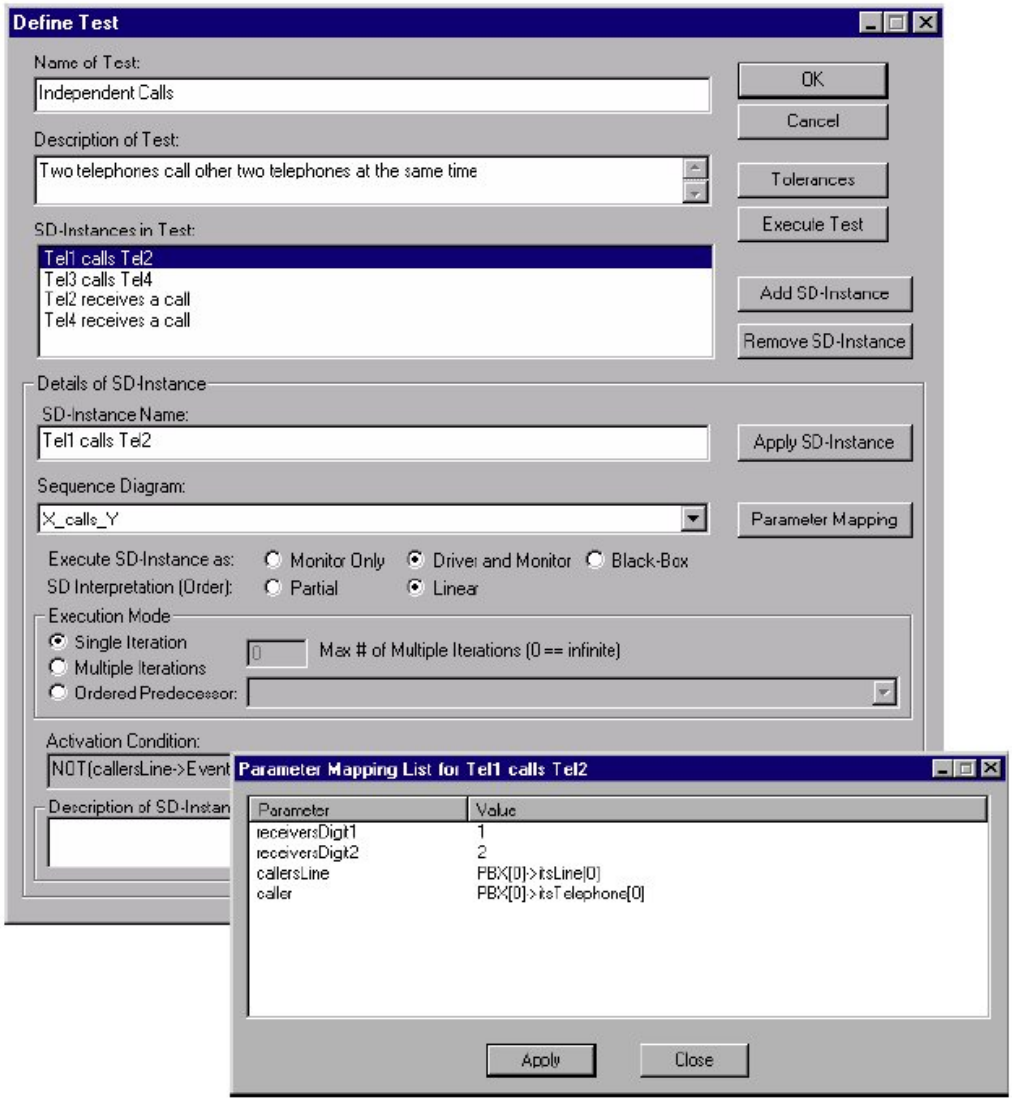
Multiple Iteration, to have TestConductor check this property several times during test execution

12. Click **Parameter Mapping** to display the list of parameters for the sequence diagram and double-click `caller`.
13. Insert the formal name of `Telephone 2`, "PBX[0]->itsTelephone[1]", then click **OK**.
14. In the **Parameter Mapping List**, click **Apply** to bind the parameter with the concrete name.
15. If desired, add a description of the sequence diagram instance in the field at the bottom of the dialog box. For example, you could describe the requirements specified in the corresponding sequence diagram.
16. Click **Apply SD-Instance**. TestConductor adds the specified sequence diagram instance to the **SD-Instances in Test** list.
17. Repeat Step 1 to Step 6 to create two other sequence diagram instances with similar settings and parameter mappings that correspond to `Telephone 3` and `Telephone 4`.

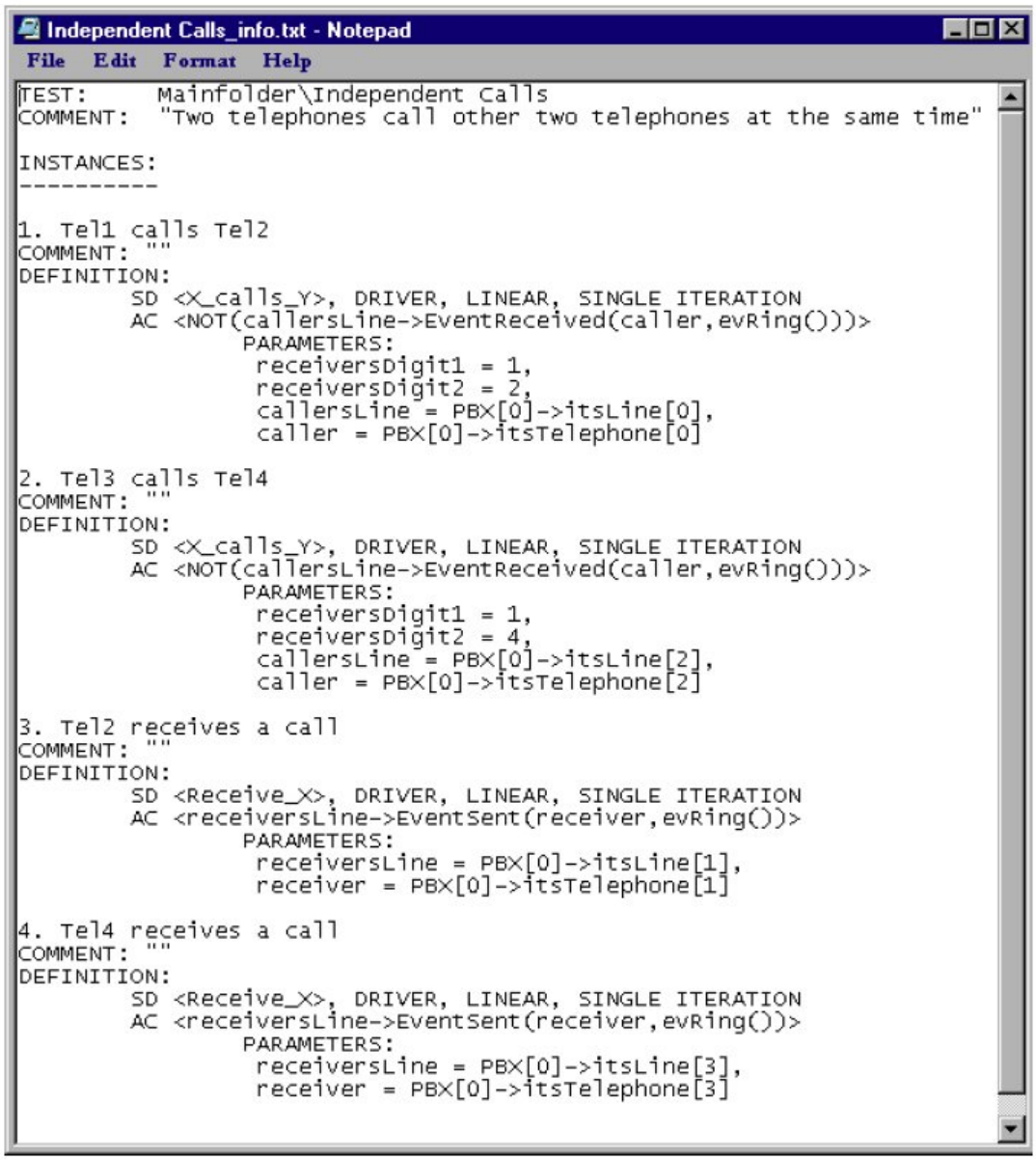
The completed test checks that `Telephones 2`, `3`, and `4` can call `Telephone 1` in any order. You can execute the test infinitely many times by injecting events manually, as specified in the "A telephone calls Telephone[0]" sequence diagram.

Automatic Driver

This example shows how to define an automatic driver with several independent sequence diagram instances. The following figure shows a test configuration with independent sequence diagram instances of the "X_calls_Y" sequence diagram (see page *172*) and the "Receive_X" sequence diagram (see page *171*). You specify the implicit order enforced between some of the sequence diagram instances using the activation conditions and parameter mappings. TestConductor drives events sent from the environment axis and monitors whether the order of "internal" messages corresponds to that specified in the sequence diagrams.



Mapping the parameters of the “X_calls_X” sequence diagram to different concrete names for different sequence diagram instances makes these sequence diagram instances completely independent. To define the automatically driven independent calls test, add four sequence diagram instances with the settings described in the following summary of the test.



This test checks that `Telephone 1` can call `Telephone 2`, and `Telephone 3` can call `Telephone 4` independently at the same time. In addition, it checks that `Telephones 2` and `4` can reply and complete calls independently. The test can be executed only one time due to the selected **Single Iteration for all SD instances** in the test configuration. Setting **Multiple Iteration** to `0`, with driver and monitor mode can lead to infinite test execution. In this case, you should specify adequate activation conditions for the corresponding sequence diagrams.

Ordered SD Instances

Using activation conditions, you can specify a predecessor order implicitly. This order might depend on the parameter mapping, and is an order of sequence diagram instance activations. For example, during execution of the test described in the previous section, the “Tel2 receives a call” sequence diagram instance is activated before the “Tel1 calls

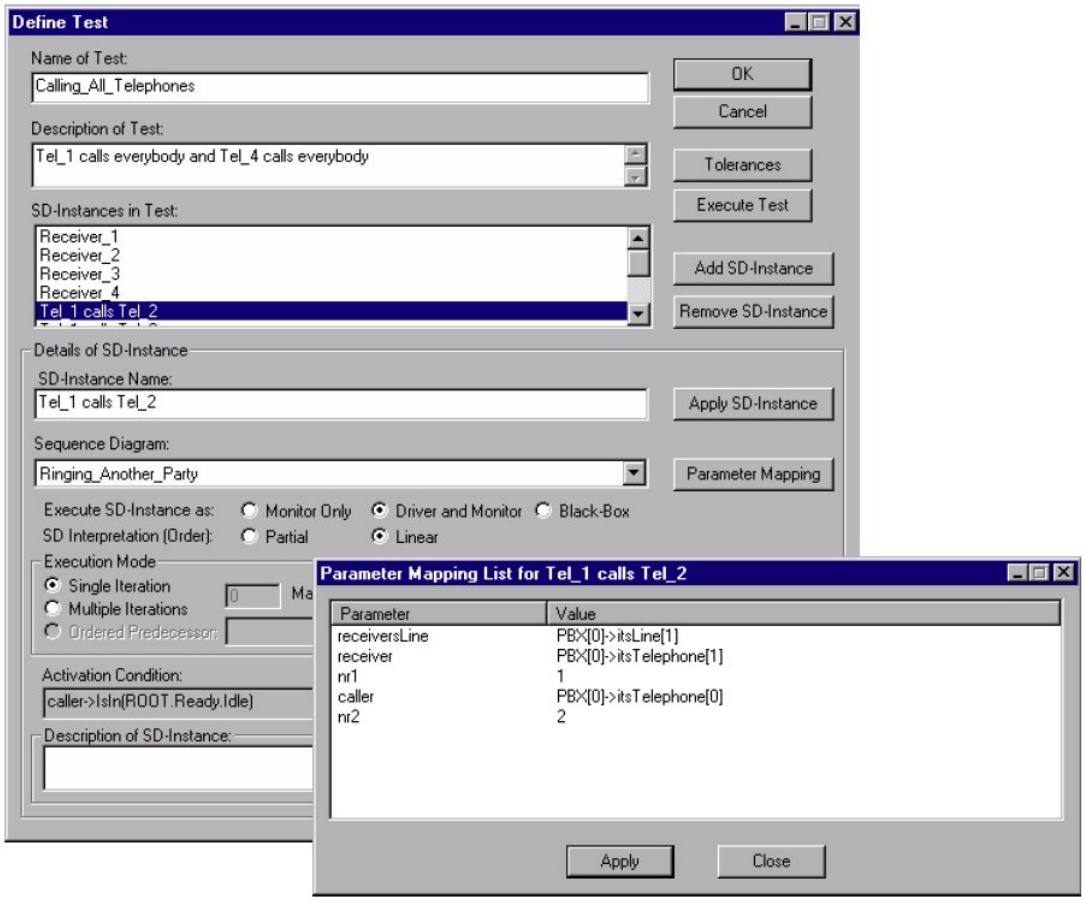
Tel2 SD" instance has been fully traversed. The following example shows the usage of explicit ordering of sequence diagram instances within a test configuration.

Note: Currently, TestConductor does not support ordered predecessors with multiple iterations.

The "Calling_All_Telephones" test configuration contains the following instances:

- Four instances (`Receiver_1`, `Receiver_2`, `Receiver_3`, and `Receiver_4`) of the "Answering_Call" sequence diagram. These sequence diagram instances are specified as driver and monitor with linear order and multiple iterations. They have disjointed parameter mappings (different concrete names bound to their parameters).
- Six instances of the "Ringing_Another_Party" sequence diagram (see the section "Condition Marks"). They are set as driver and monitor with linear order. They specify calls from `Telephone 1` to `Telephones 2`, `3`, and `4`, and from `Telephone 4` to `Telephones 1`, `2`, and `3` with predecessor order as follows:
  - The "Tel_1 calls Tel_2" sequence diagram instance has single iteration.
  - The "Tel_1 calls Tel_3" sequence diagram instance has "Tel_1 calls Tel_2" as its Ordered Predecessor.
  - The "Tel_1 calls Tel_4" sequence diagram instance has "Tel_1 calls Tel_3" as its Ordered Predecessor.
  - The "Tel_4 calls Tel_1" sequence diagram instance has "Tel_1 calls Tel_4" as its Ordered Predecessor.
  - The "Tel_4 calls Tel_2" sequence diagram instance has "Tel_4 calls Tel_1" as its Ordered Predecessor.
  - The "Tel_4 calls Tel_3" sequence diagram instance has "Tel_4 calls Tel_2" as its Ordered Predecessor.

The following figure shows the corresponding settings in the **Define Test** dialog.



During test execution, each of the last five sequence diagram instances can be activated only when the following two conditions are fulfilled:

- The sequence diagram instance specified in the test configuration as its predecessor has been fully traversed (passed or failed).
- Its activation condition becomes `TRUE`.

The specified test checks the following:

- `Telephone 1` can call all other telephones consecutively.
- `Telephone 4` can call all other telephones consecutively.
- `Telephones 1`, `2`, `3`, and `4` can answer calls as many times as they get the event `evRing` (as specified in the activation condition of the “Answering_Call” sequence diagram).

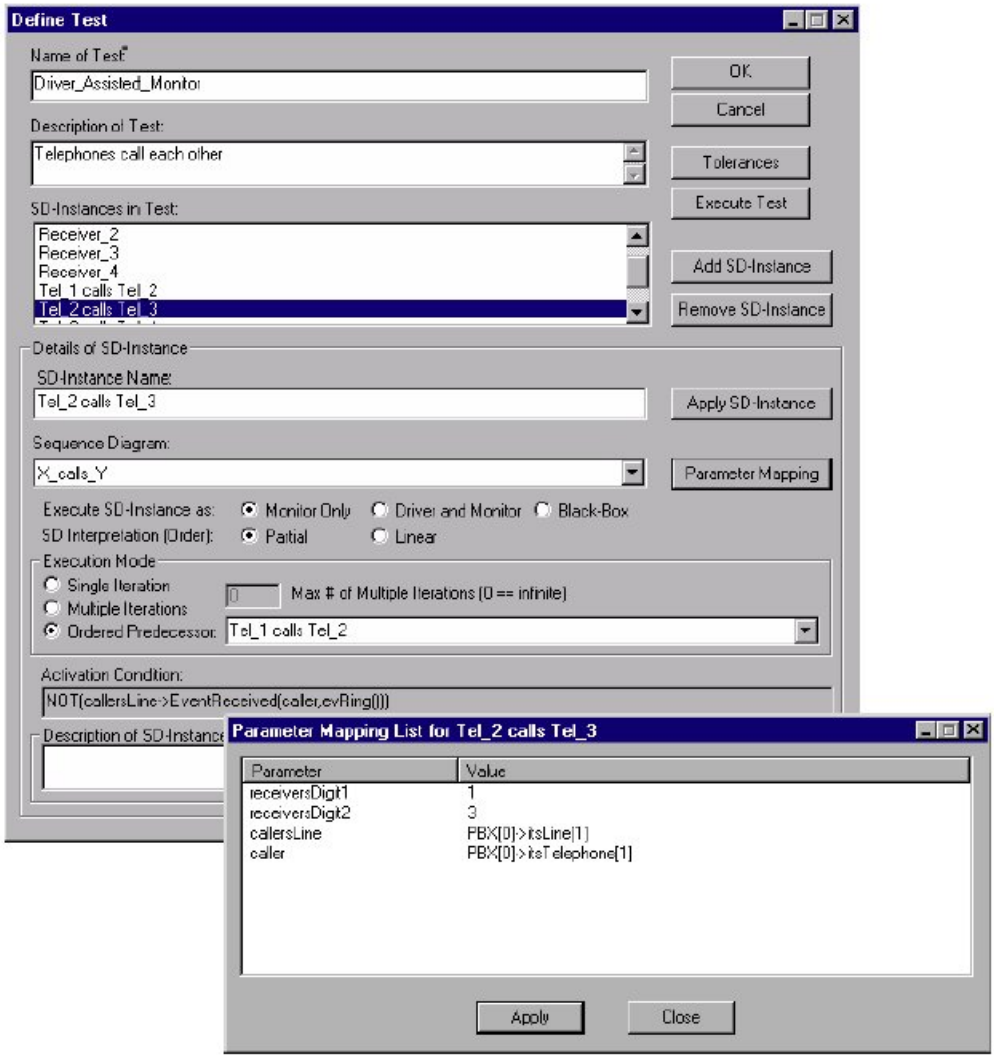
Driver-Assisted Monitor

The following examples show how to use driver-assisted monitors.

Example 1: Monitors and Drivers Specified as Sequence Diagram

This example shows how to define a combination of drivers and monitors. The “Driver_Assisted_Monitor” test configuration contains instances of the “Receive_X” sequence diagram (see page *171*) and the “X_calls_Y” sequence diagram (see page *172*). The sequence diagram instances have the following settings:

- Four instances (`Receive_1`, `Receive_2`, `Receive_3`, and `Receive_4`) of the “Receive_X” sequence diagram are specified as driver and monitor with linear order and multiple iteration. Their parameter mappings correspond to `Telephones 1`, `2`, `3`, and `4` and `Lines 1`, `2`, `3`, and `4`, respectively.
- Four instances (“`Tel_1 calls Tel_2`”, “`Tel_2 calls Tel_3`”, “`Tel_3 calls Tel_4`”, and “`Tel_4 calls Tel_1`”) of the “X_calls_Y” sequence diagram are specified as monitor only with partial order, single iteration, and the corresponding parameter mappings. The following figure shows the example of the parameter mapping for the “`Tel_2 calls Tel_3`” sequence diagram instance.

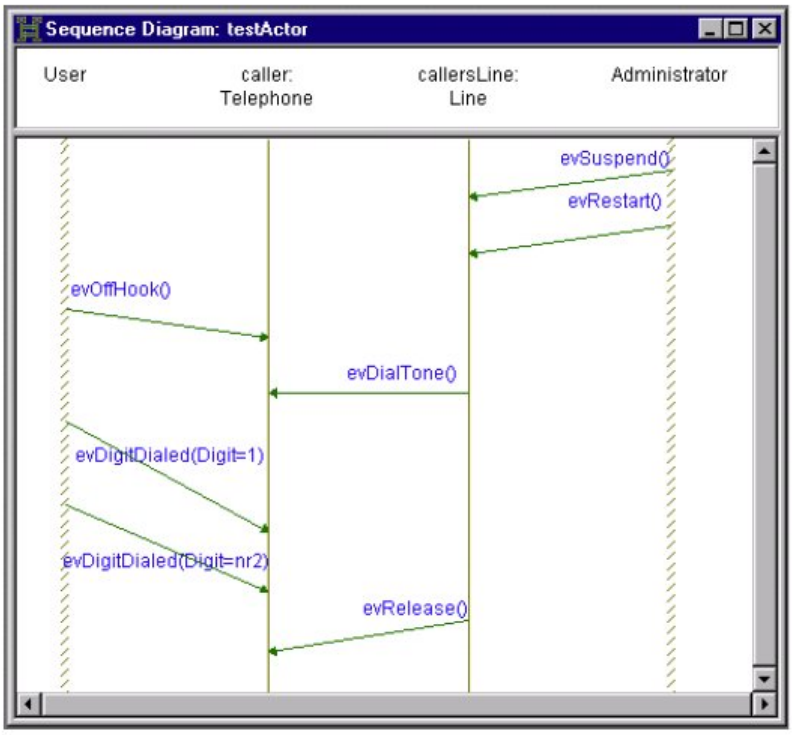


The test checks that every telephone can call the next telephone, and the telephone can reply and finish the communication. This test can be done for every specified pair of the telephones, independent of the order of the pairs. During test execution, you must drive

the model manually, as specified in the instances of the “X_calls_Y” sequence diagram. TestConductor completes the execution of the instances of the “Receive_X” sequence diagram whenever they have been activated.

Example 2: Unspecified Manual Driving

You can drive your model manually in an order not specified in any sequence diagram. This means that you do not check this part of a behaviour. For example, you can specify only communications between actor instances and internal objects when the actors have behaviour (code has been generated for them). The following sequence diagram shows such a specification for a new model. In this case, the new events `evSuspend` and `evRestart` are sent to the `Line` class from the `Administrator` actor.



The following “Check Administrator” test configuration defines a driver with an instance of the “testActor” sequence diagram.

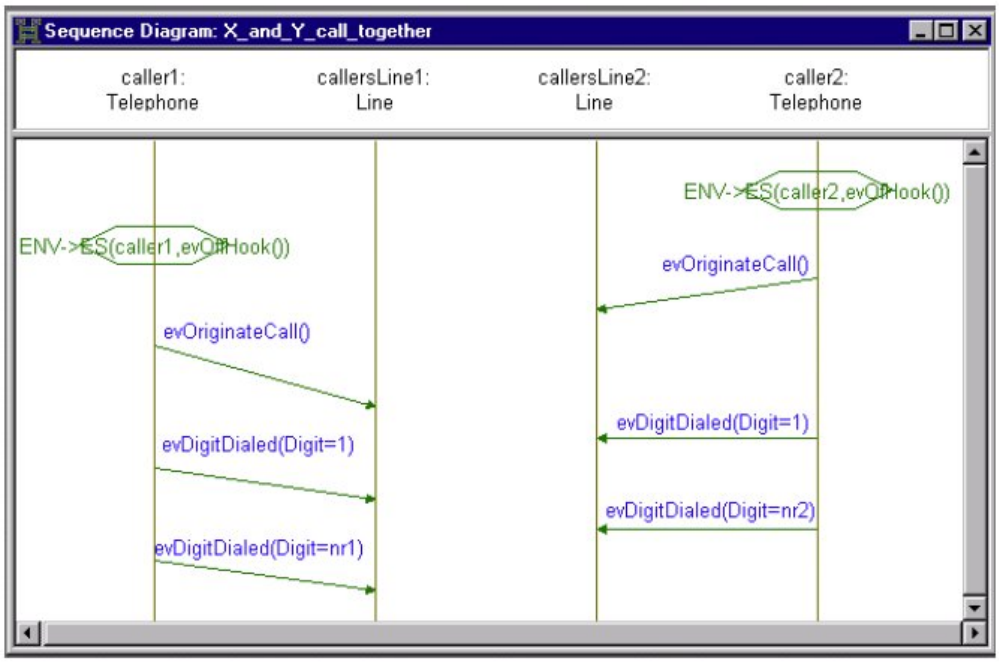
```
TEST:     Check Administrator
COMMENT:  "First Administrator sends events to Line 1.
After that User can make a call from Telephone 1."

INSTANCES:
----------

1. Tel1 calls Tel2
COMMENT: ""
DEFINITION:
        SD <testActor>, DRIVER, LINEAR, SINGLE ITERATION
        AC <TRUE>
                PARAMETERS:
                 callersLine = PBX[0]->itsLine[0],
                 caller = PBX[0]->itsTelephone[0],
                 nr2 = 2
```

This test checks that a new feature added to the system as the `Administrator` behaviour does not change the main behaviour of the model (in other words, `User` can make a call as previously specified). During test execution, you must inject input events for `Administrator` and `User` to stimulate them to send events specified in the "testActor" sequence diagram. TestConductor monitors all messages between the actors and internal objects specified in the sequence diagram under test.

Choosing Between Alternatives in a Cycle

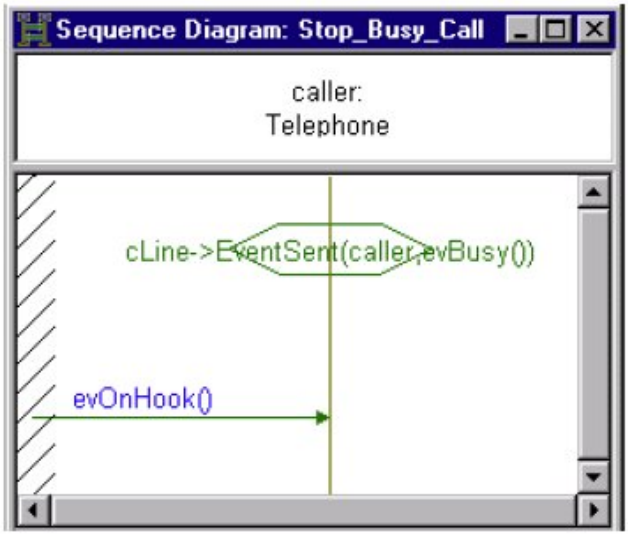
The predecessor ordering of sequence diagram instances provides a means to construct a tree or a forest (set of trees) of the related sequence diagram instances, but does not allow any cycle or choice between alternatives. Activation conditions/condition marks serve as another way to set causal dependencies between sequence diagram instances. The following test configuration explains how to combine predecessor ordering with multiple iteration to specify cycles with choice.

Consider the "X_and_Y_call_together" sequence diagram, with partial order interpretation.

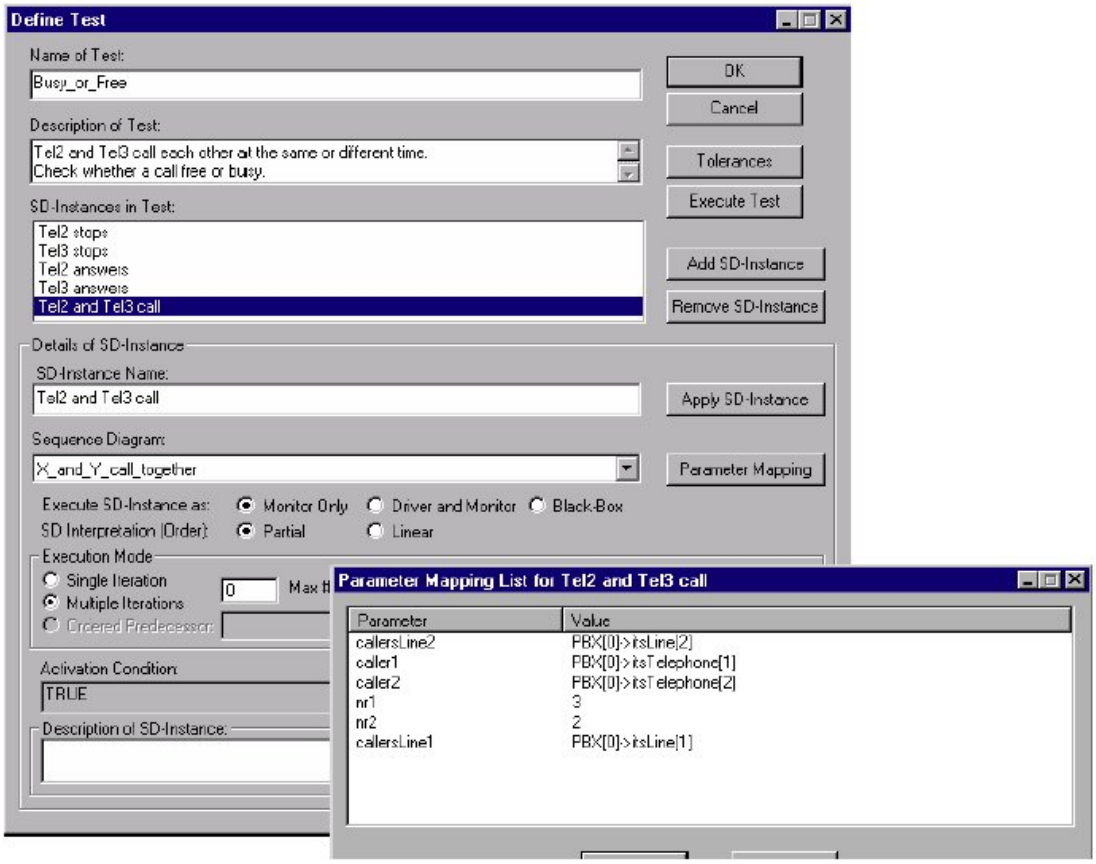


The specification says that two telephones can dial any numbers independently of each other whenever the environment sends them the `evOffHook` event. If these telephones call each other (specified by the corresponding mapping of the parameters `nr1` and `nr2`), the continuation depends on the order in which you have injected events from the environment to the telephones. A callee can be busy or answer the call.

The “Stop_Busy_Call” sequence diagram, shown in the following figure, specifies that a caller put the telephone on the hook if it gets the `evBusy` event. The “Busy_or_Free” test includes instances of the “X_and_Y_call_together” sequence diagram, the “Stop_Busy_Call” sequence diagram, and the “Answering_Call” sequence diagram.



The following figure shows the corresponding settings in the **Define Test** dialog.



The following information file of the test case definition summarizes the complete test description.

```
TEST:     MainFolder\Busy_or_Free\
COMMENT:  "Tel2 and Tel3 call each other at the same or different
time.
Check whether a call free or busy."

INSTANCES:
----------

1. Tel2 stops
COMMENT: ""
DEFINITION:
        SD <Stop_Busy_Call>, DRIVER, LINEAR, MULTIPLE ITERATIONS
        AC <TRUE>
                PARAMETERS:
                 cLine = PBX[0]->itsLine[1],
                 caller = PBX[0]->itsTelephone[1]

2. Tel3 stops
COMMENT: ""
DEFINITION:
        SD <Stop_Busy_Call>, DRIVER, LINEAR, MULTIPLE ITERATIONS
        AC <TRUE>
                 PARAMETERS:
                 cLine = PBX[0]->itsLine[2],
                 caller = PBX[0]->itsTelephone[2]

3. Tel2 answers
COMMENT: ""
DEFINITION:
        SD <Answering_Call>, DRIVER, LINEAR, MULTIPLE ITERATIONS
        AC <receiversLine->ES(receiver,evRing())>
                PARAMETERS:
                 receiversLine = PBX[0]->itsLine[1],
                 receiver = PBX[0]->itsTelephone[1]

4. Tel3 answers
COMMENT: ""
DEFINITION:
        SD <Answering_Call>, DRIVER, LINEAR, MULTIPLE ITERATIONS
        AC <receiversLine->ES(receiver,evRing())>
                PARAMETERS:
                 receiversLine = PBX[0]->itsLine[2],
                 receiver = PBX[0]->itsTelephone[2]

5. Tel2 and Tel3 call
COMMENT: ""
DEFINITION:
        SD <X_and_Y_call_together>, MONITOR, PARTIAL, MULTIPLE
ITERATIONS
        AC <TRUE>
                 PARAMETERS:
                 callersLine2 = PBX[0]->itsLine[2],
                 caller1 = PBX[0]->itsTelephone[1],
                 caller2 = PBX[0]->itsTelephone[2],
                 nr1 = 3,
                 nr2 = 2,
                 callersLine1 = PBX[0]->itsLine[1]
```

The test checks the following:

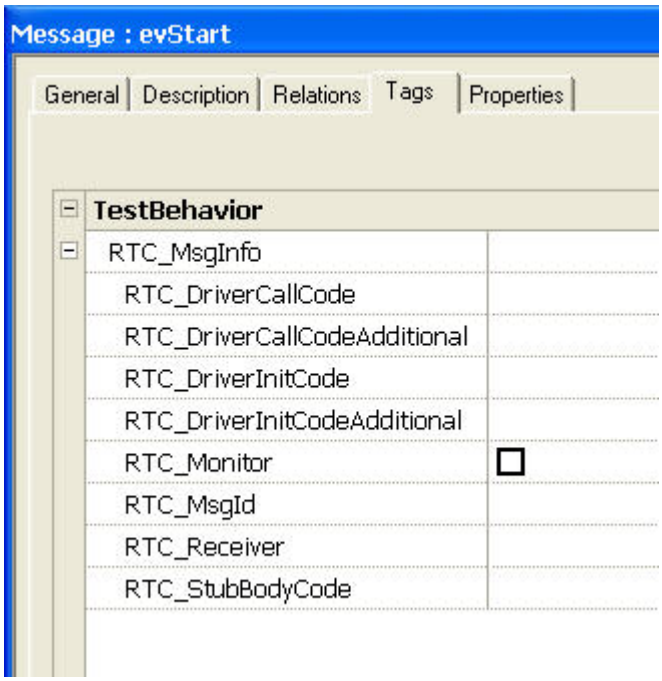
- `Telephone 2` and `Telephone 3` call each other independently.
- If a callee (`Telephone 2` or `Telephone 3`) is free, it answers the call.
- If a callee is busy, the caller hangs up.

You can execute the test continuously, injecting events to `Telephone 2` and `Telephone 3`. TestConductor monitors the `"Tel2 and Tel3 call"` sequence diagram instance and drives the remaining ones, selecting those relevant to the current situation. Note that the instance of the "X_and_Y_call_together" sequence diagram is the predecessor for the remaining four instances in the test configuration. This means that the sequence diagram instances "`Tel2 stops`", "`Tel3 stops`", "`Tel2 answers`", and "`Tel3 answers`" can be activated only after the `Tel2` and `Tel3` call instance has been activated and partially traversed. This order (and the choice between alternatives) is specified with the activation conditions and Condition Marks, but become valid only after the parameters have been bound to the corresponding names.

User Defined Driving Operation Calls

The default implementation of a driver operation generated by TestConductor may be overwritten and customized by the user, by stereotyping the message with stereotype `<<RTC_MsgInfo>>` in the sequence diagram and setting the corresponding values for the tags

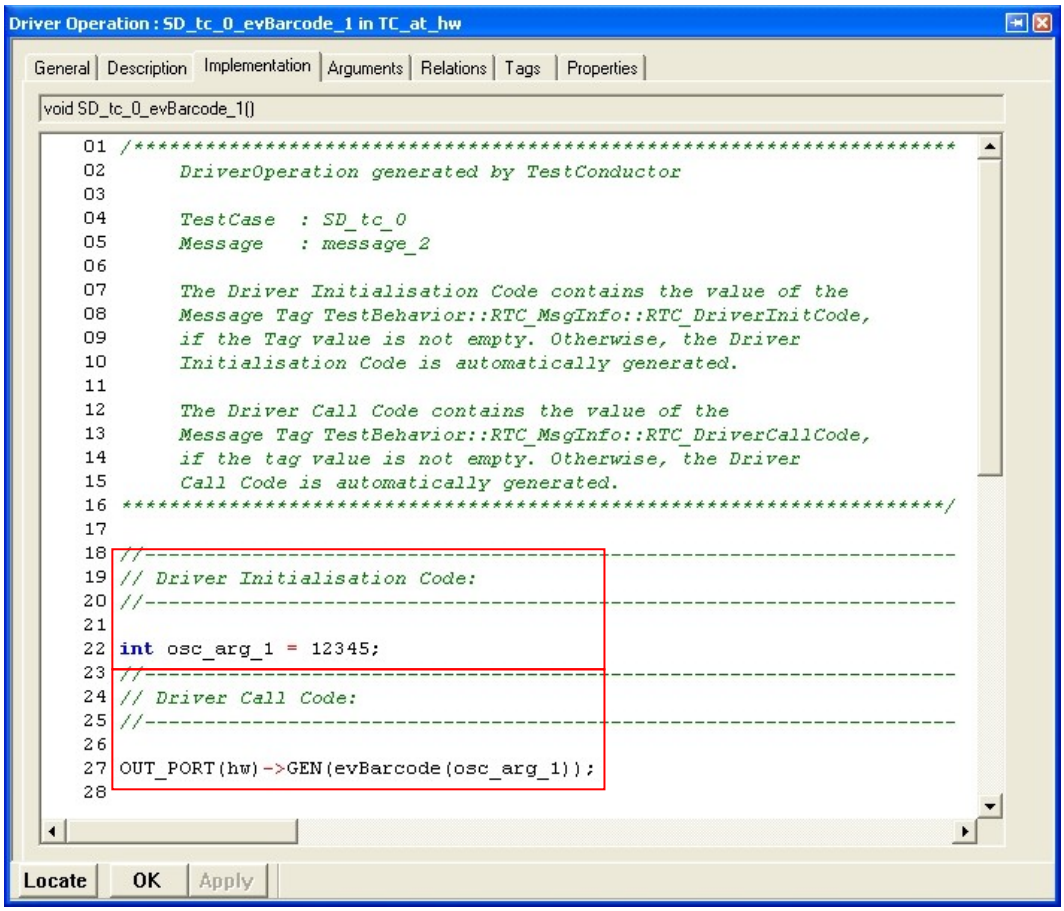
```
TestBehavior::RTC_MsgInfo::RTC_DriverCallCode,
TestBehavior::RTC_MsgInfo::RTC_DriverCallCodeAdditional,
TestBehavior::RTC_MsgInfo::RTC_DriverInitCode,
TestBehavior::RTC_MsgInfo::RTC_DriverInitCodeAdditional,
```



Usually, if the user modifies driver operations in the model, then this information is lost if the user updates a test case. The user can influence the generated code for driver operations and stub operations. Using the tags
`TestBehavior::RTC_MsgInfo::RTC_DriverCallCode`,
`TestBehavior::RTC_MsgInfo::RTC_DriverCallCodeAdditional`,
`TestBehavior::RTC_MsgInfo::RTC_DriverInitCode`,
`TestBehavior::RTC_MsgInfo::RTC_DriverInitCodeAdditional`,

the content of these tags is not lost during update of a test case.

The value for `RTC_DriverInitCode` is taken as the beginning of the driver operation body containing the initialization of necessary variables, whereas the value for `RTC_DriverCallCode` is taken as the trailing part of the driver operation body containing the call of the function to be driven.



Note that both properties can be overwritten separately by the user. In case the user wants to customize the initialization section only, only the property `RTC_DriverInitCode` has to be overwritten; TestConductor will continue to automatically generate the code for the driver call section (and vice versa).

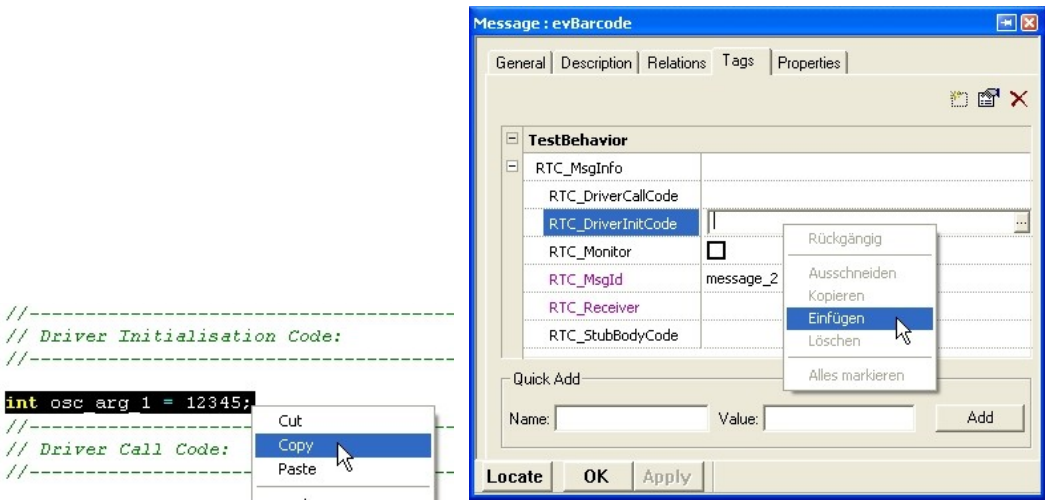
The value for `RTC_DriverInitCodeAdditional` is taken as additional initialization code that is generated in addition to the initialization code generated by TestConductor. The content of this tag is generated directly after the auto generated initialization code. Similarly, the value for `RTC_DriverCallCodeAdditional` is taken as additional call

code that is generated in addition to the auto generated call code. The content of this tag is generated directly after the auto generated call code.

RTC_DriverInitCode and RTC_DriverInitCodeAdditional

The user can influence the initialization of arguments before the message is driven using the tags `RTC_DriverInitCode` and `RTC_DriverInitCodeAdditional`. To do this uses have to add the stereotype `RTC_MsgInfo` to the SD message. This adds automatically the tags `RTC_DriverInitCode` and `RTC_DriverInitCodeAdditional` to the message. The user can fill these tags with code which will be used as initialization code of the driver operation when the test case 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.



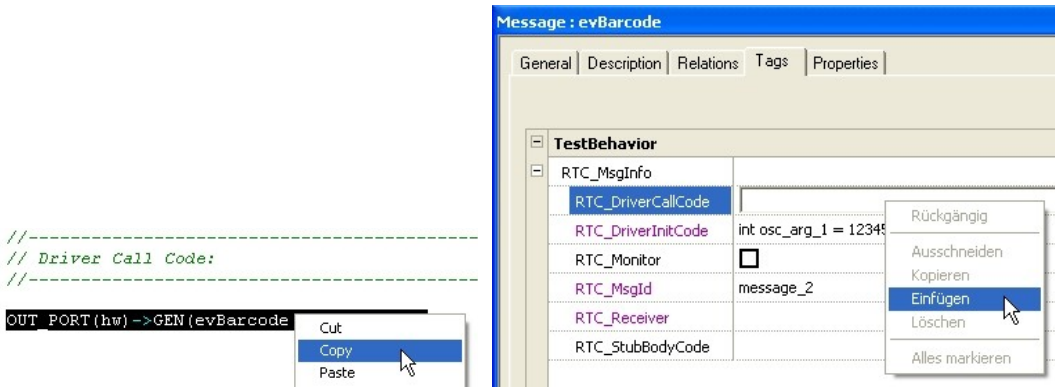
RTC_DriverCallCode and RTC_DriverCallCodeAdditional

The user can also influence the call of the driven operation using the tags `RTC_DriverCallCode` and `RTC_DriverCallCode`Additional. 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 responsiblitythat the sequence diagram test case is indeed executable after customization. Basically, the specified message of the sequence

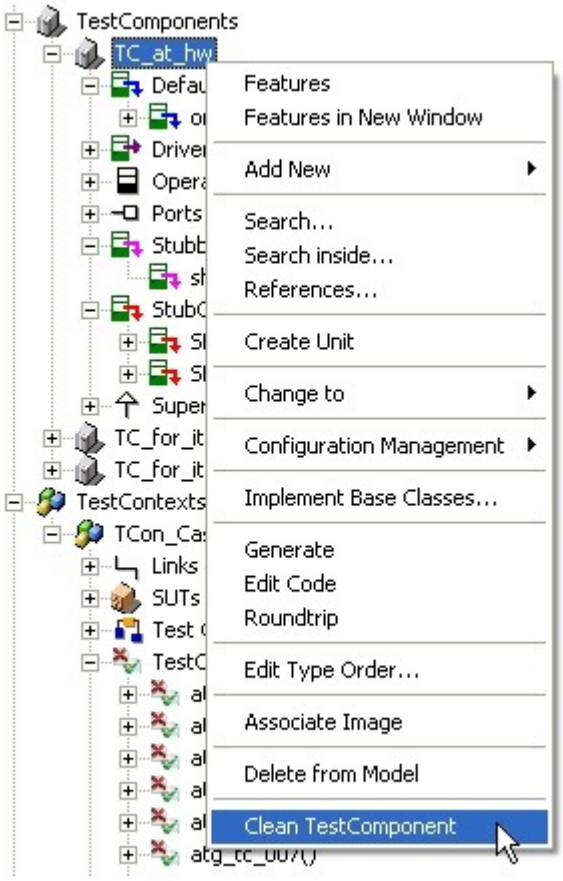
diagram test case, which now is present as source code, has to be represendted in the user defined code.

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_DriverDriverCode` before starting to implement his own changes.



Clean TestComponent

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



Clean TestPackage

Driver and stub operations can be deleted manually, but TestConductor provides the functionality to delete the automatically generated operations of all test components of a TestPackage at once. Furthermore, **Clean TestPackage** also deletes all results and coverage results from the TestPackage.

To clean a test package select the test package und choose from the context menu the item **Clean TestPackage.**

To regenerate the driver an stub operations select the test case or the test context or the test package and choose from the context menu the item **Update TestCase/TestContext/TestPackage**.

Deleting User Defined Driver Operation Calls

TestConductor uses user defined operation calls if the tags `TestBehavior::RTC_MsgInfo::RTC_DriverInitCode` and `TestBehavior::RTC_MsgInfo::RTC_DriverCallCode` are not empty, even if the tags are overwritten. To delete the user defined operation call and use the auto generated driver operations from TestConductor, reset the tags to delete the content of the tag.

User Defined Stub Operation Calls

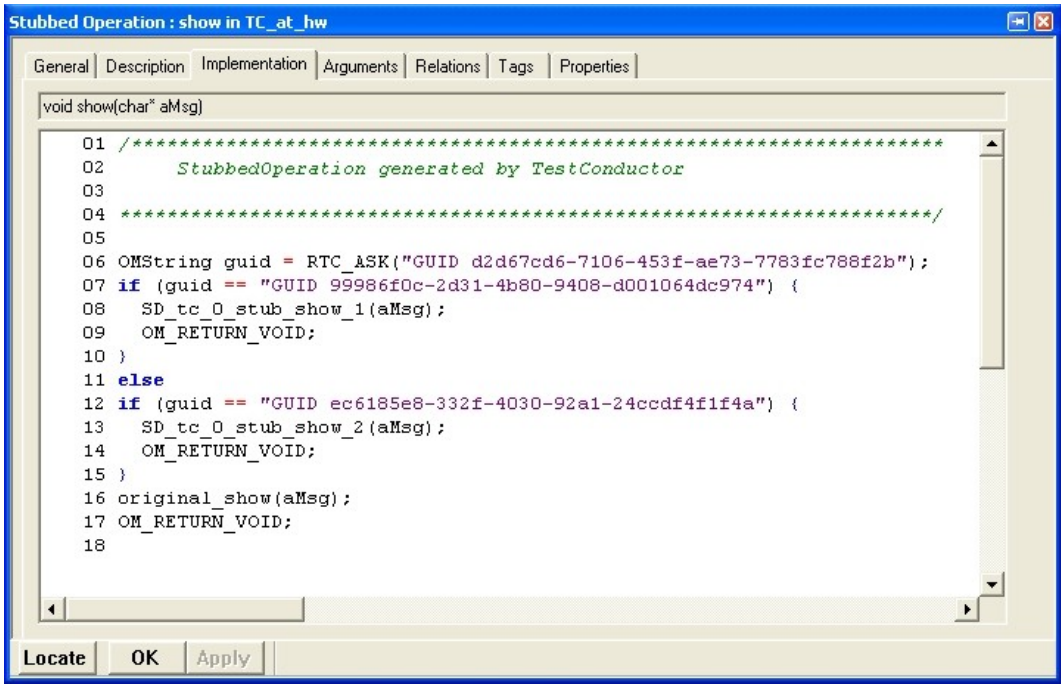
Stub operations are created for any operation call in the sequence diagram going from the SUT to a test component if the following items are all true:

- a return value (or a returned value for an out or in/out argument) is specified for this operation
- the tag `TestConductor::RTC_MsgInfo::RTCMonitor` for the sequence diagram message is set to false
- the tag `TestConductor::RTCInstInfo::RTCMonitor` for the To-sequence diagram instance line is false

TestCondoctor needs the ability to determine and control the value returned by the operation. On the other hand there might be calls to the same operation without a specified return value or the operation is called by a test component on a test component: because of this TestCondoctor has to generate a different body for the operation, but it must still be possible to call the original operation.

To ensure this, TestCondoctor creates a copy of the original operation with the name `orginal_` followed by the operations name, having the same signature. In the implementation body of this so called *DefaultOperation* the original function is called non-virtually. For every occurrence of the operation where it should be stubbed, a new operation is added to the test component with the same signature of the original operation. This so called *StubOperation* returns the specified return value, out and in/out arguments. The name of the stub operation is the concatenation of the name of the test case, the string "_stub_", the name of the original operation followed by a number to make it unique.

The body of the original operation is deleted completely and a new implementation is generated this way: The operation does a call to a special TestCondoctor operation and uses the `OMString` value returned by TestCondoctor in a switch statement to select which operation should be called. If a stub operation has to be invoked TestCondoctor returns its `GUID`, if the original operation has to be called TestCondoctor returns an empty string.



The actual values of formal parameters defined for the sequence diagram or sequence diagram instance are propagated to the stub operation this way: If any parameter is used in the return value or out or in/out arguments of the operation that has to be stubbed, then in the body of the stub operation this parameter is exchanged with the value of the parameter.

RTC_StubBodyCode

Normally, if the user modifies stub operations in the model, then this information is lost if the user updates a test case. The user can influence the code of the stub using the tag `RTC_StubBodyCode`. To do this he has to add the stereotype `<<RTC_MsgInfo>>` to the sequence diagram message, this adds automatically the tag `RTC_StubBodyCode` to the message. The user can fill this tag with code which will be used as body of the stub operation when the test case is updated. Important is that this code completely replaces the body that would be generated by TestConductor automatically.

An important limitation is: only virtual operations can be stubbed. Since the SUT is implemented, in the SUT code operations of other design classes are called. For instance, a class `A` which is the SUT class may call a operation "`f`" of a class `B`. Now, in a given test architecture, a new test component class BT is introduced that inherits from B in order to be able to use an instance of class BT instead of an instance of class B directly. However, the SUT code still calls the operation "`f`" of `B`, since the SUT code remains untouched. But when "`f`" is a virtual operation, the virtual dispatching mechanism of *UML* ensures that the most specialized variant of the operation is called, i.e., if class `BT` implements a new version of the called operation "`f`", then this function is called. This function can be stubbed, since it is defined in the testing component `BT`. However, if the SUT calls a non-virtual function, it cannot be stubbed since this operation is in general not defined in a testing component.

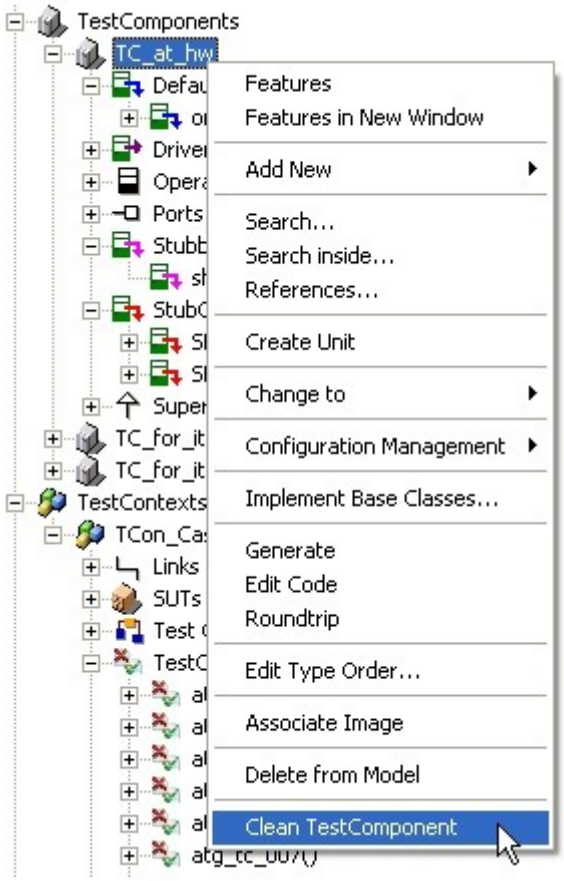
If an operation is stubbed multiple times in the same test component in the same sequence diagram instance, then for each occurrence an individual stub operation is generated.

If an operation is stubbed multiple times in the same test component in the same SUT in different test cases respectively sequence diagram instances, then for each occurrence an individual stub operation is generated.

Tip: In case TestConductor has not created stub operations for a sequence diagram message, the at the beginning mentioned conditions are not fulfilled. To “inspire” TestCondutor to create such stubbing functionality anyhow, the user can define “*” as expected return value for the sequence diagram message followed by an update on the test case. In some cases TestConductor will then create the customizable stubbing functionality as shown in the above picture.

Clean TestComponent

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



Clean TestPackage

Driver and stub operations can be deleted manually, but TestConductor provides the functionality to delete the automatically generated operations of all test components of a TestPackage at once. To clean a test package select the test package und choose from the context menu the item **Clean TestPackage.**

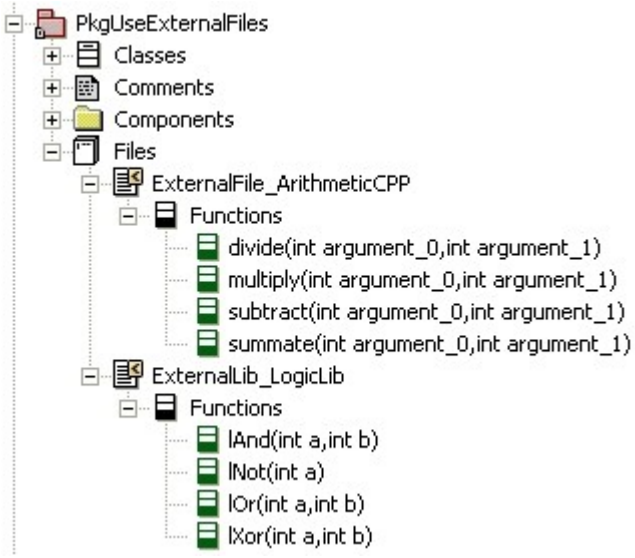
To regenerate the driver an stub operations select the test case or the test context or the test package and choose from the context menu the item **Update TestCase/TestContext/TestPackage**.

Deleting User Defined Stub Operation Calls

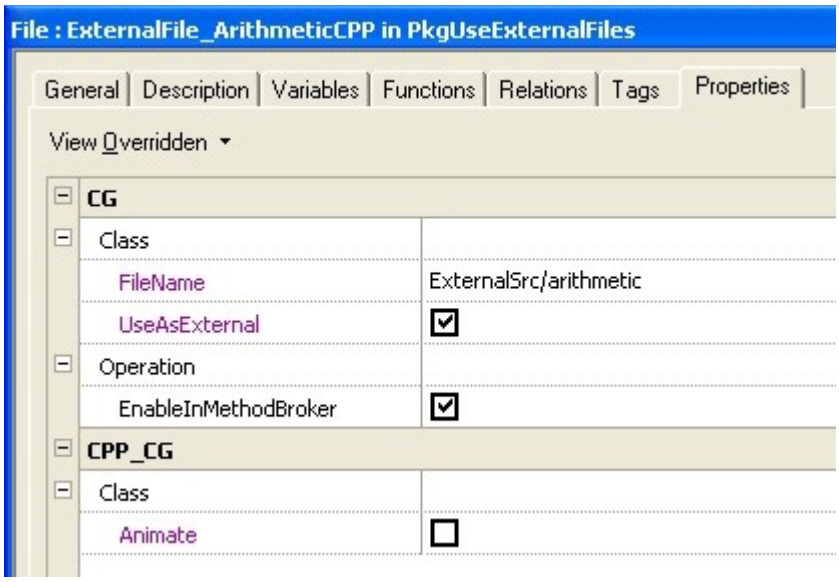
TestConductor uses user defined operation calls if the tags `TestBehavior::RTC_MsgInfo::RTC_StubBodyCode` are not empty, even if the tags are overwritten. To delete the user defined operation call and use the auto generated stub operations from TestConductor, reset the tags to delete the content of the tag.

Black-Box Testing of External Files and Libraries

TestConductor comes with the C++ sample `CppTestingExternalFiles`. This project contains the package `PkgUseExternalFiles`, where two files are defined. The declared external file `ExternalFile_ArithmeticCPP` consists of a source file `arithmetic.cpp` and the corresponding header file `arithmetic.h`. The file `ExternalLib_LogicLib` consist of the library `LogicLib.lib` and a corresponding header file `LogicLib.h`. Further information on how to define files can be found in the *Rhapsody User Guide*.



Open the feature dialog of a file, select the Properties tab and browse the overwritten properties of `ExternalFile_ArithmeticCPP`.



`CG.Class.UseAsExternal` is set to `TRUE`.

`CG.Class.FileName` determines the basename of the referenced external file. This property defines `ExternalFile_ArithmeticCPP` to refer to `arithmetic.h` in the project's `ExternalSrc`-directory.

`CPP_CG.Class.Animate` is set to `FALSE`. Whatever the library or the external source file contains Rhapsody animated code, the property has to be set to `FALSE`. Setting this property to `FALSE` means, that the file, which will become in this example the SUT, will not be animated. Furthermore, disabling the animation of the SUT means to perform a black-box test.

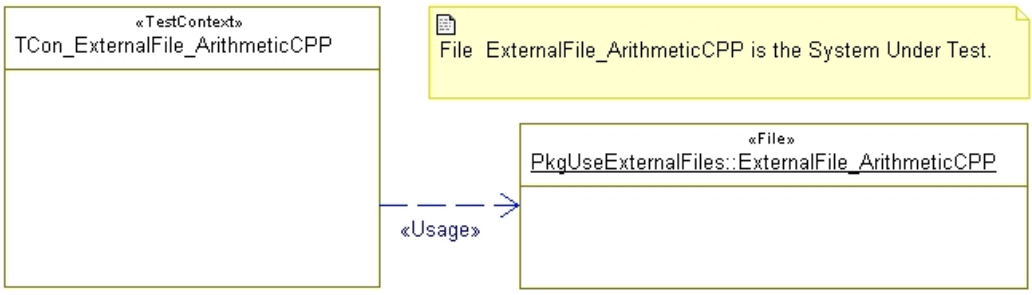
In order to use external header and implementation in code-generation, component `UseExternalFiles` defines the additional include-path "`../..`", which refers to the project's root-directory. The implementation of the external functions is made available to code-generation by defining additional source "`../../ExternalSrc/arithmetic.cpp`". In order to link the library the configuration `UseExternalFiles::Default` defines under Libraries "`../../LogicLib/NotInstrumented/LogicLib.lib`".

To use this example and the provided test cases in the test packages `TPkg_ExternalFile_ArithmeticCPP` and `TPkg_ExternalLib_LogicLib` the user has first to generate/build the `LogicLib.Lib` and the header file `LogicLib.h`. Browse the package `PkgLogicLib`, set the containing configuration `LogicLib::NotInstrumented` active and build the configuration by using the **Generate/Make/Run** button.

Test Packages

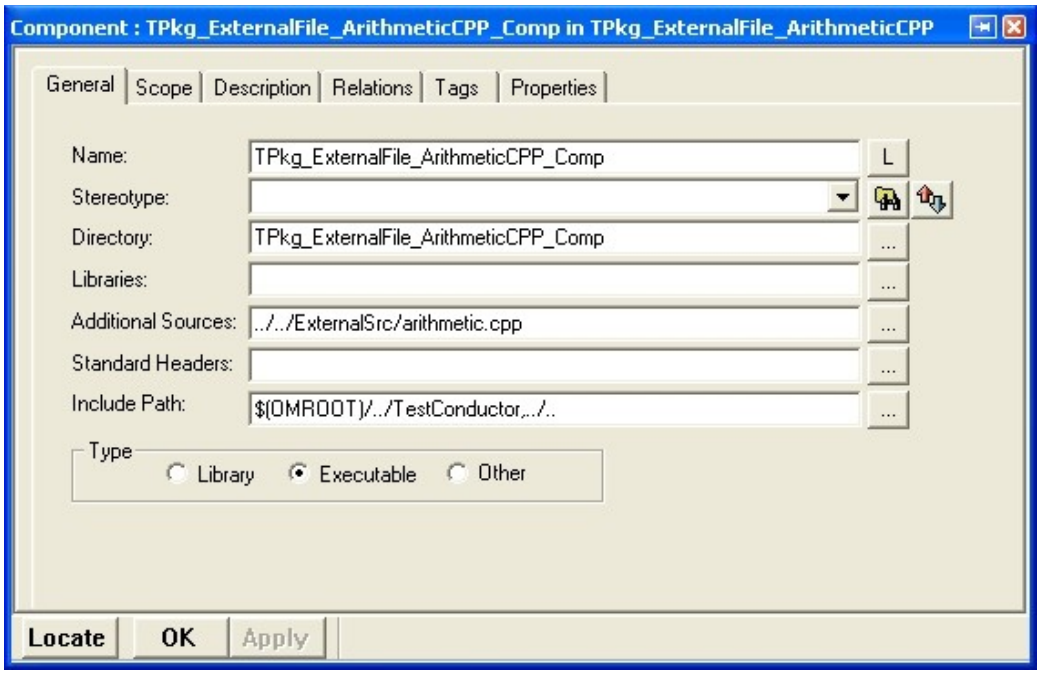
The example comes with pre-defined test architecture for the file `ExternalFile_ArithmeticCPP`. The test architecture was created as follows:

For testing external file `ExternalFile_ArithmeticCPP`, select `ExternalFile_ArithmeticCPP` and choose **Create TestArchitecture** in the context menu. A new test package `TPkg_ ExternalFile_ArithmeticCPP` will be created

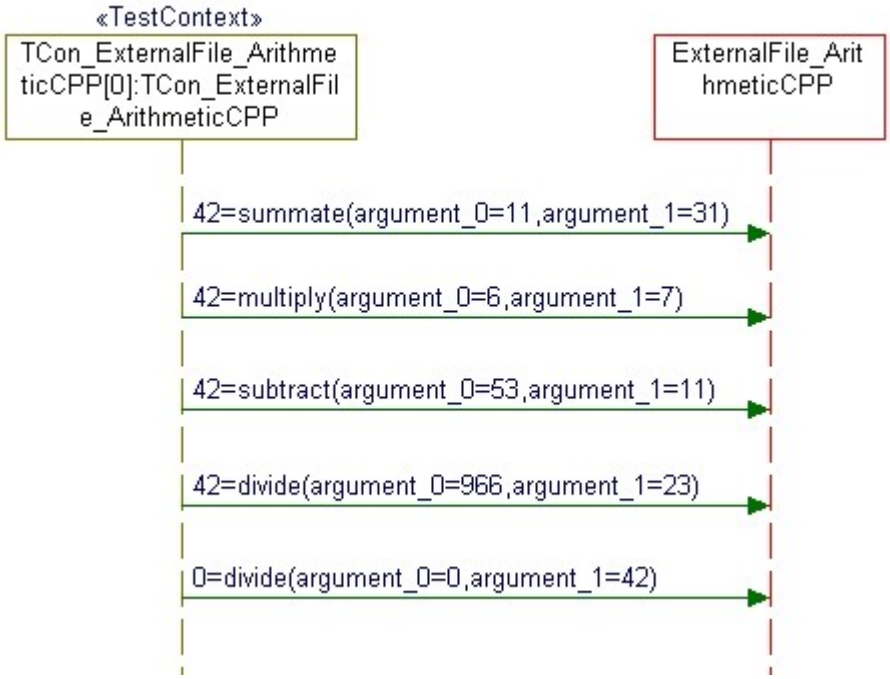


In order to make test context `TPkg_ExternalFile_ArithmeticCPP::TCon_ExternalFile_ArithmeticCPP` compilable and linkable, the user has to modify code generation component `TPkg_ExternalFile_ArithmeticCPP::TPkg_ExternalFile_ArithmeticCPP_Comp`:

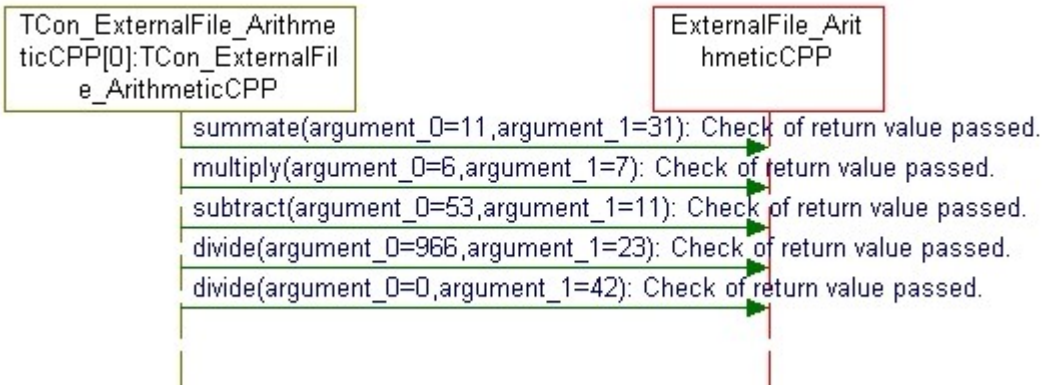
1. enter "`../../ExternalSrc/arithmetic.cpp`" into entry **Additional Sources** in the General tab.
2. extend the include path in entry **Include Path** to "`$(OMROOT)/../TestConductor,../..`"



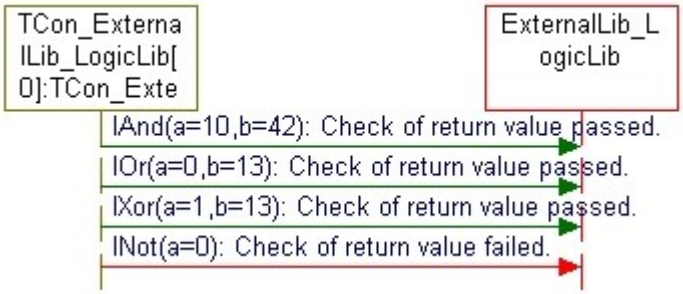
The example comes with a pre-defined test architecture for the file `ExternalFile_ArithmeticCPP` and the library `ExternalLib_LogicLib`. Also the following sequence diagram test cases have already been defined:



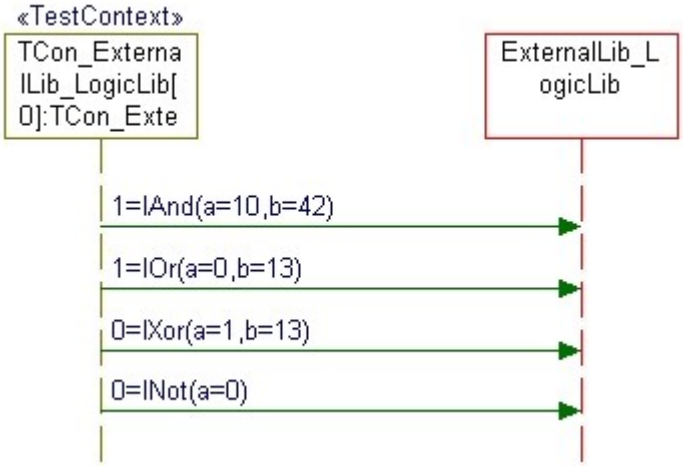
To execute the test case `SD_tc_0` select the test case in the Rhapsody browser and choose from the context menu **Update Test Case, Build Test Case, Execute Test Case**. In the TestConductor execution dialog click on the button **Activate Test**. TestConductor shows that the test case `SD_tc_0` passed. For further information select in the TestConductor execution dialog the entry `SD_tc_0` and click on the button **Show as SD**. The animated sequence diagram displays the text execution result and states, that all return values occurred as specified.



Now execute the test case `SD_tc_0` in the test context `TCon_ExternalLib_LogicLib`. The test will fail and the **Show As SD** sequence diagram will state, that the check of the return value failed.



Open the test scenario `SDTestScenario_0` of test case `SD_tc_0` in the test context `TCon_ExternalLib_LogicLib`.



The expected value in the expression "`0=lNot(a=0)`" is wrong. The correct return value has to be "`1=lNot(a=0)`". Correct the test scenario and re-run the test. It will pass.

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

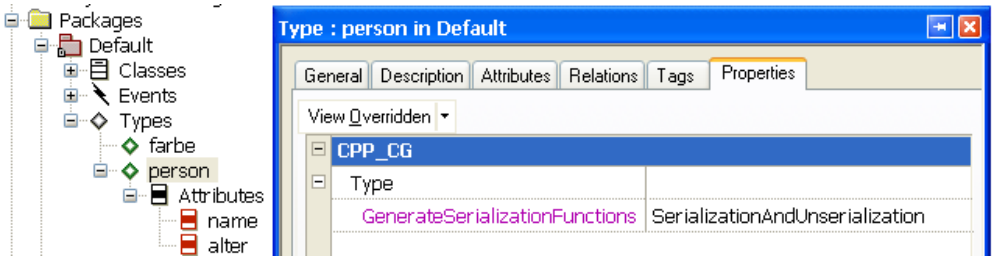
interfaces can be used with files in C, declaration and implementation of functionality can be handled separately in the model.
TestConductor offers specific support for <<CIntefaceFile>> interfaces, by stubbing the implementations if a file to be tested as SUT refers to <<CInterfaceFile>> interfaces.

Using Serialize/Unserialize Functions for User Defined Types

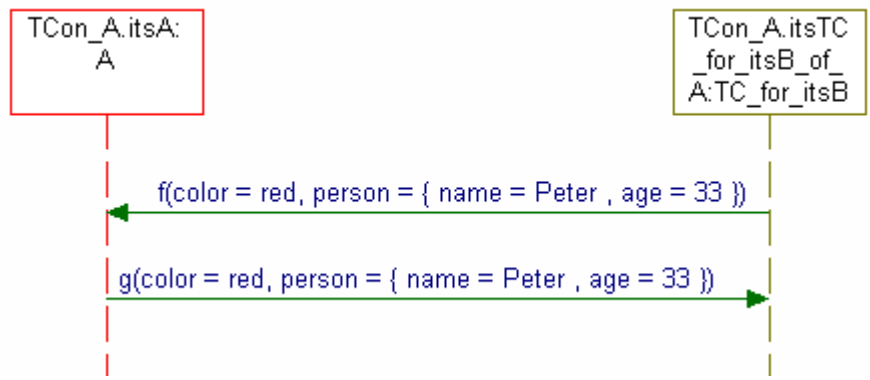
Rhapsody can animate (display) the values of simple types and one-dimensional arrays. However, if you want to animate a more complex type, the type must be converted to a string (char *) for Rhapsody to display it. This can be done generally in two different ways, either by using auto-generated serialization/unserialization 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, the property "<Lang>.Type.GenerateSerializationFunctions" must be set to "SerializationAndUnserialization":



If this property is set correctly, for arguments with enum type one can use the literals of the enum definition in the test scenarios, and for arguments with structure type one can specify each attribute defined in the structure type. The following test scenario shows two message "f" and "g" that both have two arguments, one of enum type and one of a structure type:

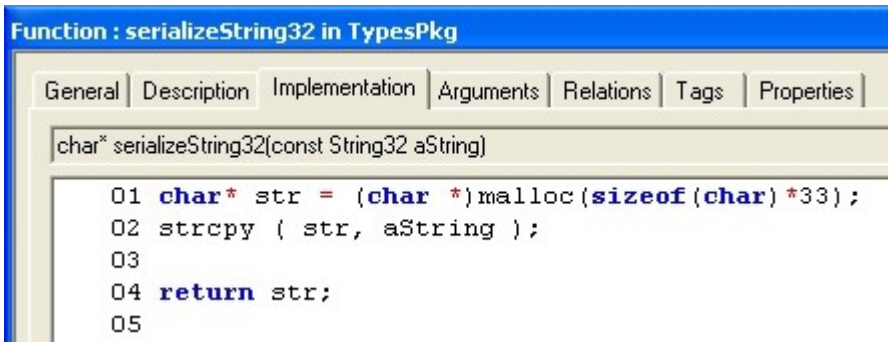


Using manually defined serialization /unserialization functions

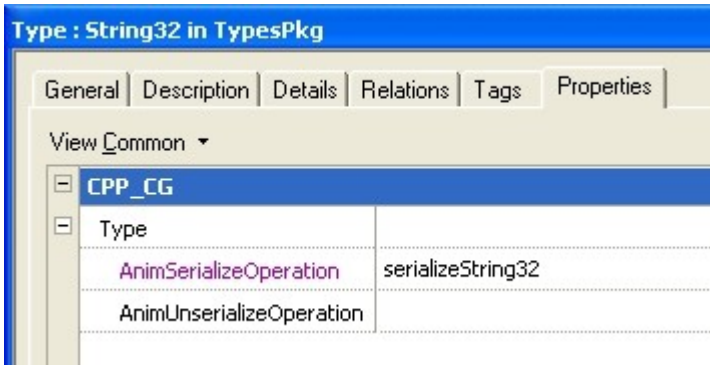
Besides using the auto generated serialization/unserialization functions of Rhapsody, one can also manually define serialization/unserialization functions. These functions are global instrumentation functions, that takes one argument of the type you want to display, and returns a char *. Further information 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 selected directly 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.

In case of non fault tolerant programming of these (un-)serialize function the application/model may probably work during normal operation, but can crash, if the user executes a test case on the same model. The following example shows a Sting32 type.

The user defined the following serialize function:



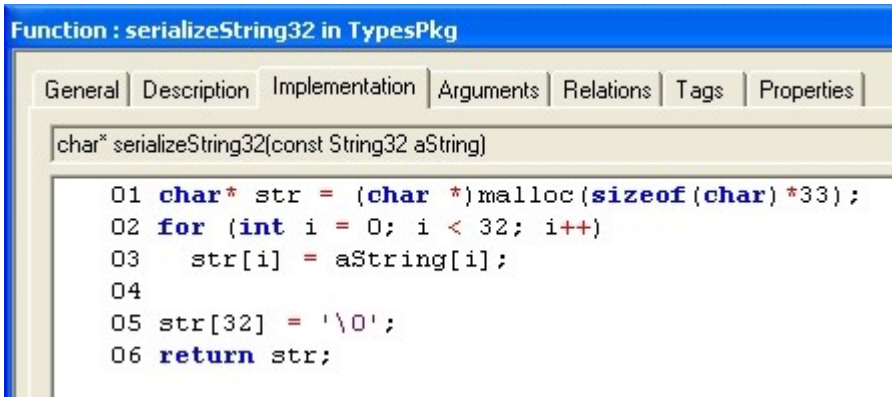
And connected it correctly to the corresponding property



During normal operation everything will work properly. But during execution of a test case on the unchanged model the execution will crash.

The reason for the crash is the serialization function for String32, it causes a crash if it is called with a not initialized string. If TestConductor registers as an observer the framework notifies TestConductor about operation calls. To do this the framework serializes the arguments of the constructor (== conversion to string).

If the serialize function for String32 is modified this way the application will not crash:



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:

1. By errors in the model − the computed model behavior does not meet requirements specified by a sequence diagram
2. 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, state chart 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. This implies that TestConductor meets failure in the following two cases:

1. The real order of message actions during model execution does not correspond to specifications in sequence diagram instances.
2. The real argument values of messages during model execution do not correspond to those specified in sequence diagram instances.

During test compilation, TestConductor translates every sequence diagram instance into internal sequence(s) of message actions specified in the sequence diagram instance. As you activate a test, TestConductor starts the model execution and creates the first iteration copies of sequence diagram instances without specified ordered predecessors as the original run-time instances. During test execution, TestConductor checks the activation condition of each created run-time instance until it gets value `TRUE` (that is a run-time instance becomes active). After that, TestConductor checks every messages appearing in the model execution. For every currently active run-time instances from the **Execute Test** dialog, it compares the following:

1. Whether the current message belongs to the message set of the corresponding sequence diagram.

2. Whether all message actions preceding the current message in the corresponding run-time instance have already occurred.

If the first condition does not hold, TestConductor ignores the current message. If both conditions hold, TestConductor marks the current message as green. If only the first condition is fulfilled − one or more actions preceding current one in the corresponding run-time instance have not yet appeared in the model execution − TestConductor creates a red message, reports failure and stops to traverse the run-time instance with erroneous message action. After that it continues to generate run-time instances with respect to the specified execution mode, check activation conditions and new message actions.

Failure Reporting

TestConductor draws a green horizontal message arrow for operation calls that have been monitored. Events that have been monitored in-order are drawn as slanted messages as in sequence diagram animation. The starting point of the slanted message is where the event has been sent. The end point refers to the point where this event must be consumed according to the original sequence diagram specification.

Note: In our green, blue, red approach one could consider the dashed line as half-green (event has been sent) and half-blue (consumption not yet monitored).

Following classes of errors can be detected by TestConductor:

1. Sending out of order
2. Event Sending - Parameter values do not match
3. Event Sending - Parameter values not in range
4. Consumption out of order
5. Event Consumption - Parameter values do not match
6. Event Consumption - Parameter values not in range
7. Operation Call out of order
8. Operation Call - In Parameter values do not match
9. Operation Call - In Parameter values not in range
10. Operation Call returned - Return value does not match
11. Operation Call returned - Out Parameter values do not match
12. Operation Call returned - Out Parameter values not in range
13. DataFlow Message - Value does not match
14. DataFlow Message - Value not in range
15. DataFlow Message out of order

16. Assertion failed

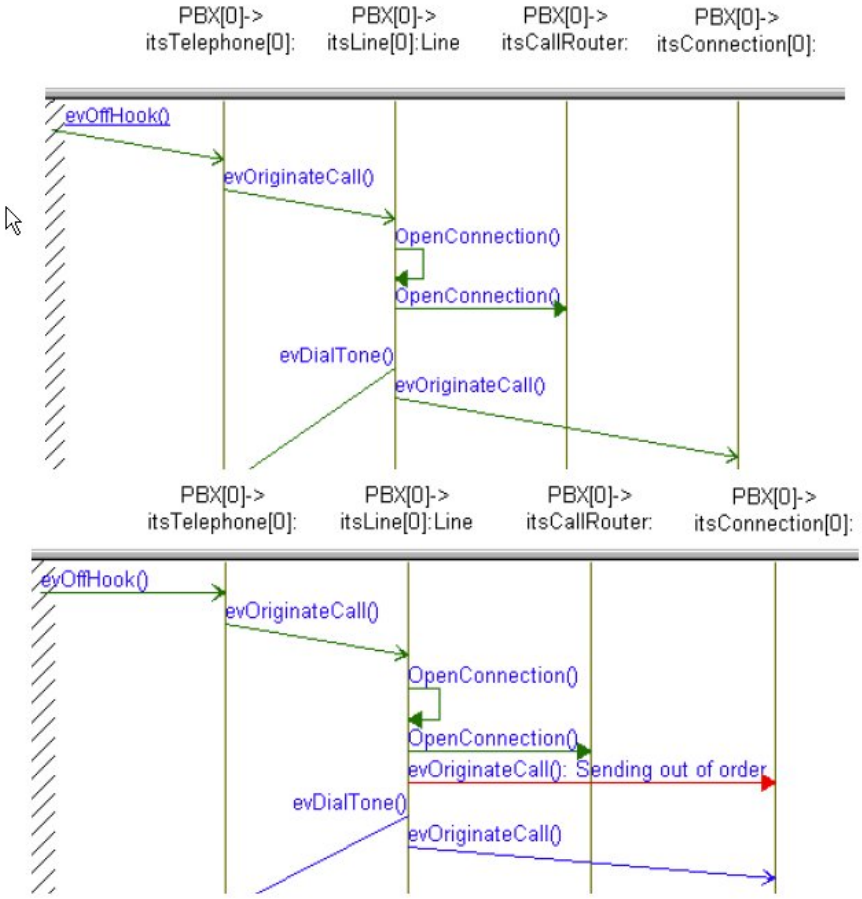
TestConductor draws a red horizontal message to visualize a failure. The red arrow refers to a point where a message was monitored out-of-order or where parameter values did not match. The red message is labeled with a text (M() represents the failed message):

- M():Sending out of order
- M():Event Sending - Parameter values do not match
- M():Event Sending - Parameter values not in range
- M():Consumption out of order
- M():Event Consumption - Parameter values do not match
- M():Event Consumption - Parameter values not in range
- M():Operation Call out of order
- M():Operation Call - In Parameter values do not match
- M():Operation Call - In Parameter values not in range
- M():Operation Call returned - Return value does not match
- M():Operation Call returned - Out Parameter values do not match
- M():Operation Call returned - Out Parameter values not in range
- M():DataFlow Message - Value does not match
- M():DataFlow Message - Value not in range
- M():DataFlow Message out of order
- M():Assertion <SD_instance_X: message Y> failed

TestConductor draws blue messages for messages that have not yet monitored, neither sending nor consumption of events. Such a drawn sequence diagram contains the original sequence diagram specification used for the test. All green and blue messages represent the messages of the original sequence diagram. Green and blue messages, together with the red arrow make failure analysis much easier. If the red message is erased, then the drawn sequence diagram can be used to reproduce the same failure.

Note: Red messages can not be erased automatically from a failure sequence diagram used in a new test. Workaround is to erase it manually if such a sequence diagram shall be used in a test. Following samples explain the failure cases.

Event sending out-of-order

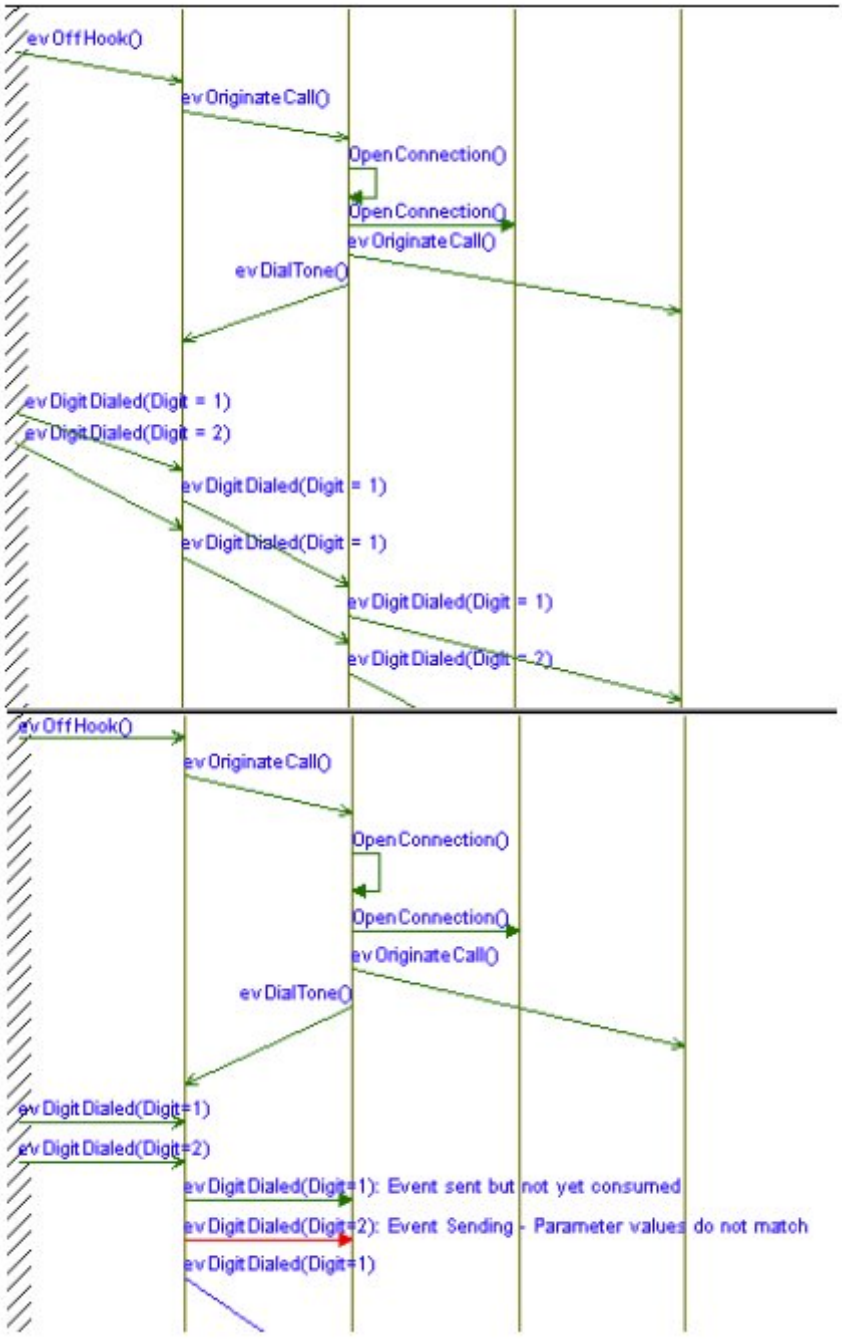


In this example, according to the specification: TestConductor must

1. Monitor the self message `OpenConnection()`
2. Monitor the operation call `OpenConnection()`
3. Monitor the sending of `evDialTone()`
4. Monitor the sending of `evOriginateCall()`

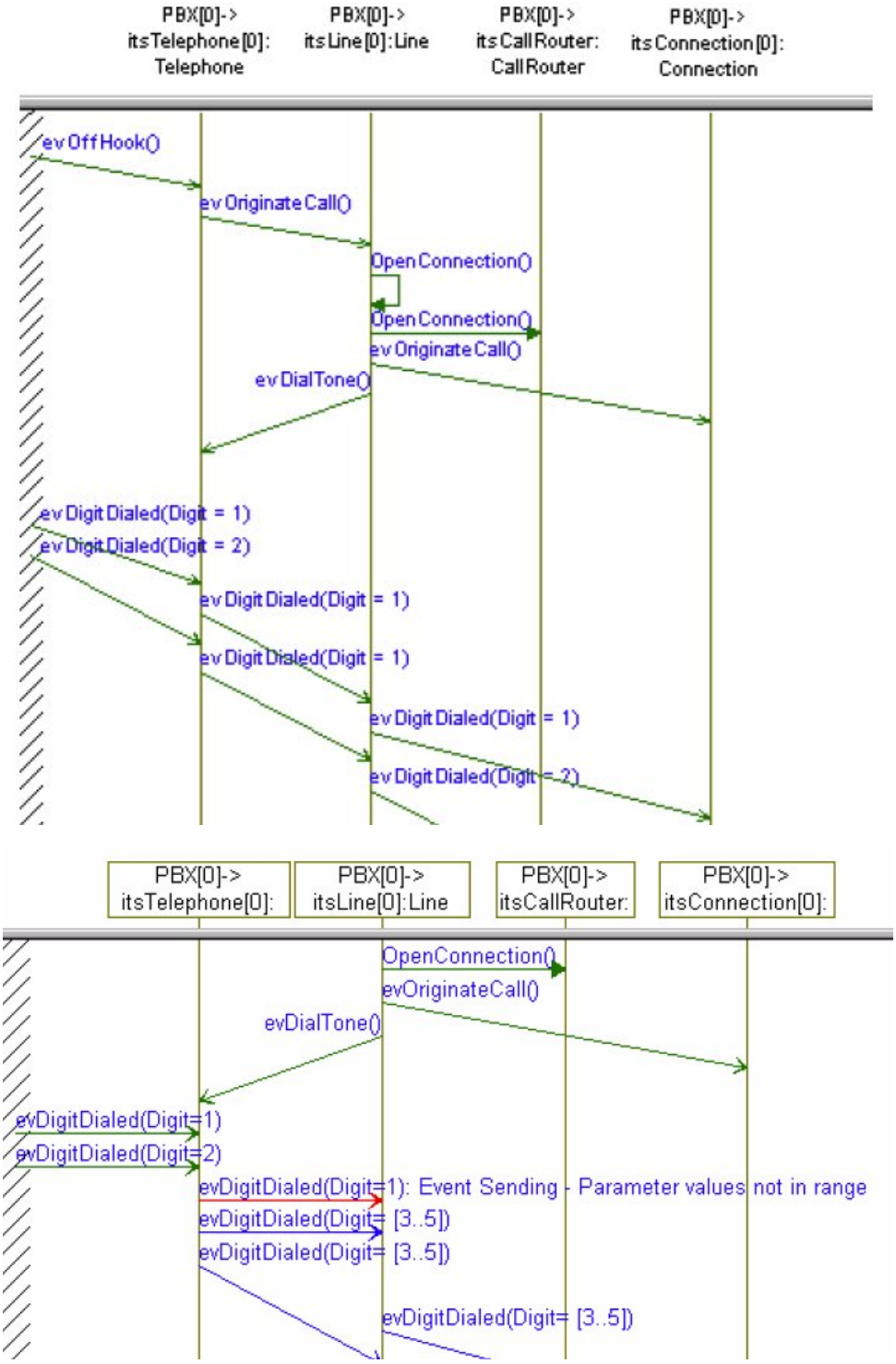
TestConductor sees, sending of event `evOriginateCall()` occurs before the sending of `evDialTone()`. Thus TestConductor gives the warning "`Sending out of order`".

Event sending in-order, but parameter values do not match



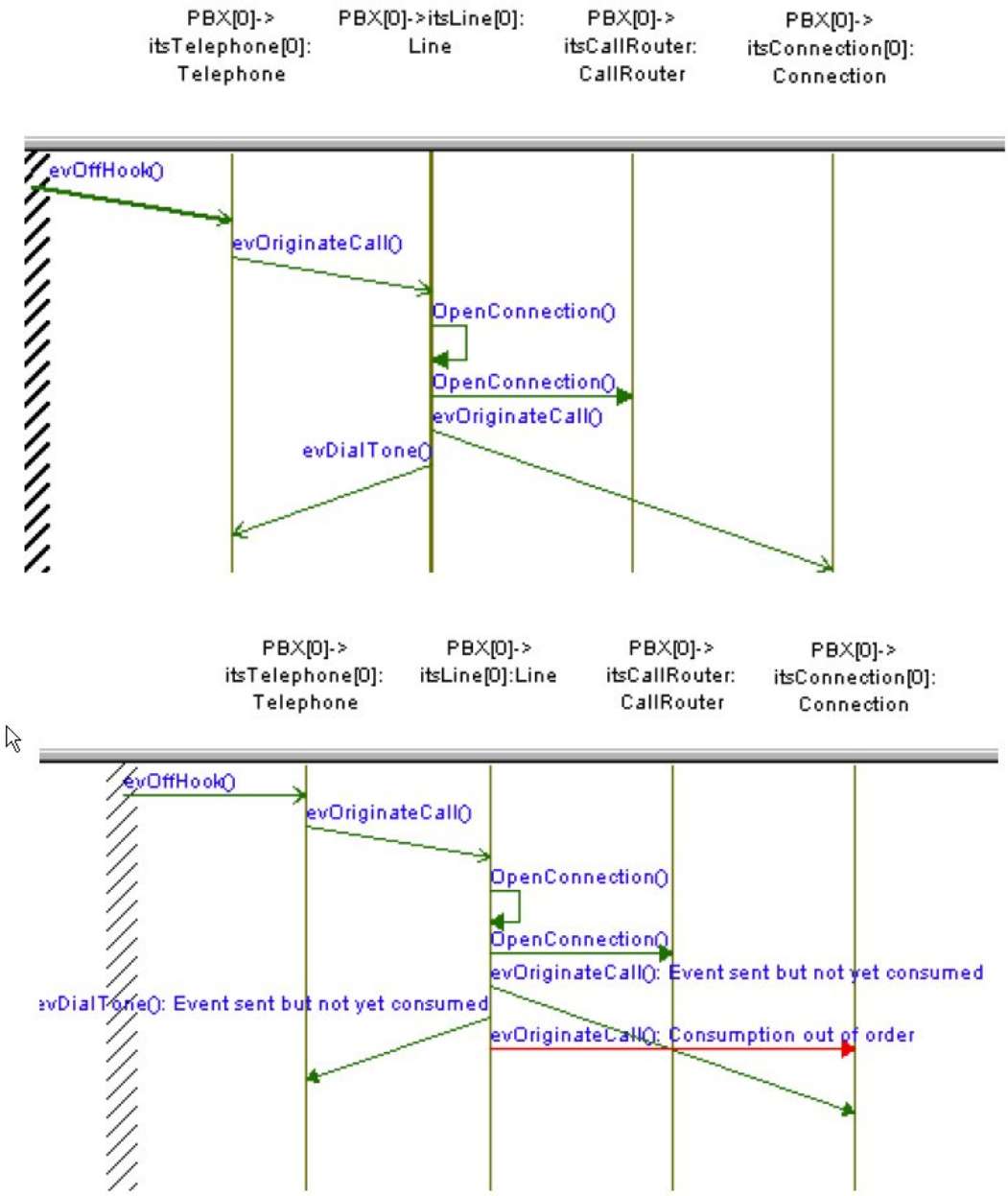
In this example, according to the specification, TestConductor must monitor the event `evDigitDialed(Digit = 1)`, but TestConductor is seeing `evDigitDialed(Digit = 2)`. Thus TestConductor reports a failure "`Event Sending -Parameters values do not match`"

Event sending in-order, but parameter values not in range



In this example, according to the specification, TestConductor must monitor the event `evDigitDialed(Digit = 1)`, but TestConductor is seeing `evDigitDialed(Digit = [3..5])`. Thus TestConductor reports a failure "`Event Sending - Parameters values not in range`".

Event consumption out-of-order



In this example, according to the specification, TestConductor must monitor

1. The operation call `OpenConnection()`
2. The sending of `evOriginateCall()`
3. The sending of `evDialTone()`
4. The consumption of `evDialTone()`
5. The consumption of `evOriginateCall()`

TestConductor sees consumption of `evOriginateCall()` before the consumption of `evDialTone()`. Thus TestConductor gives the warning "`Consumption out of order`".

Event consumption in-order, but parameter values do not match

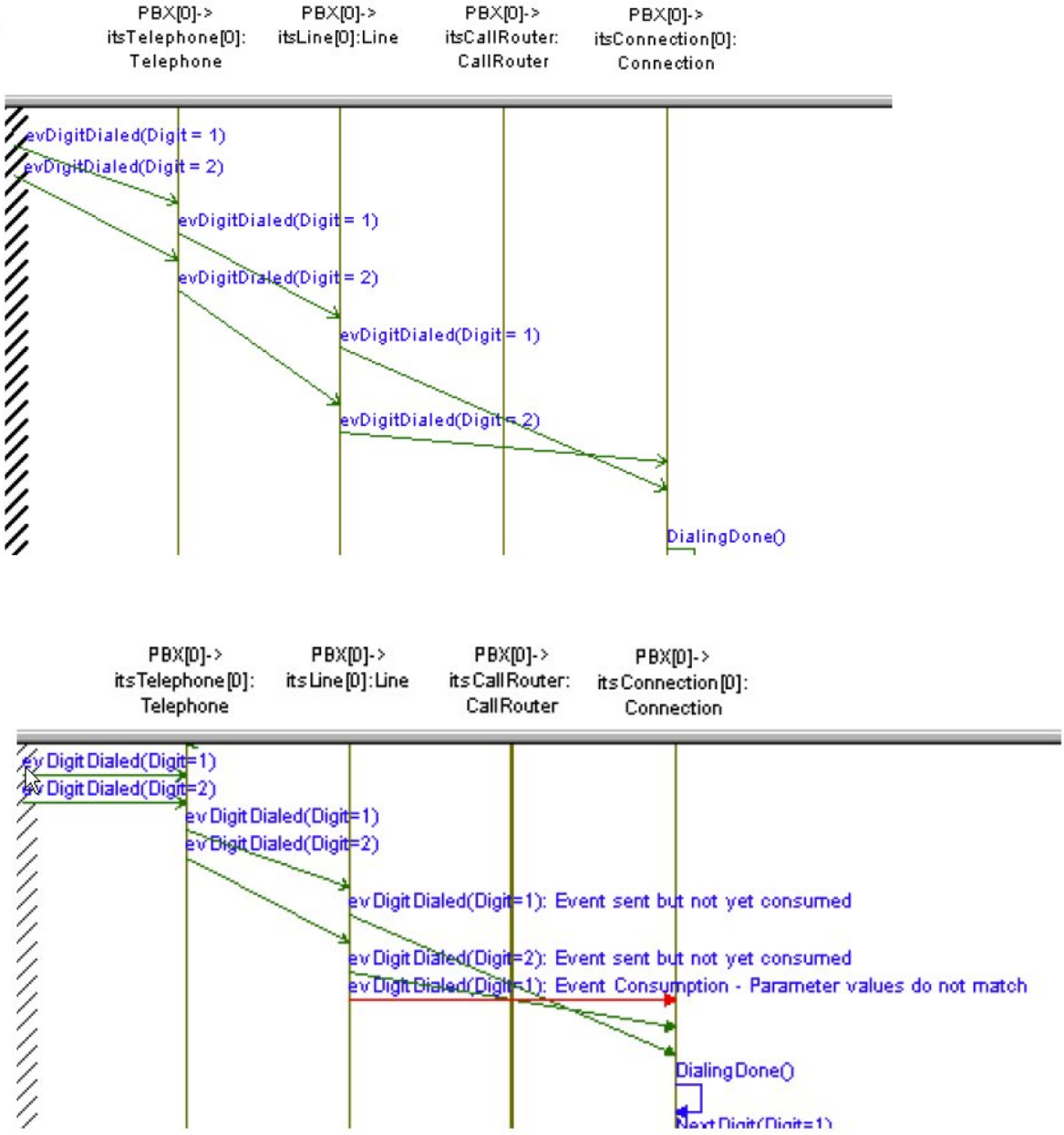


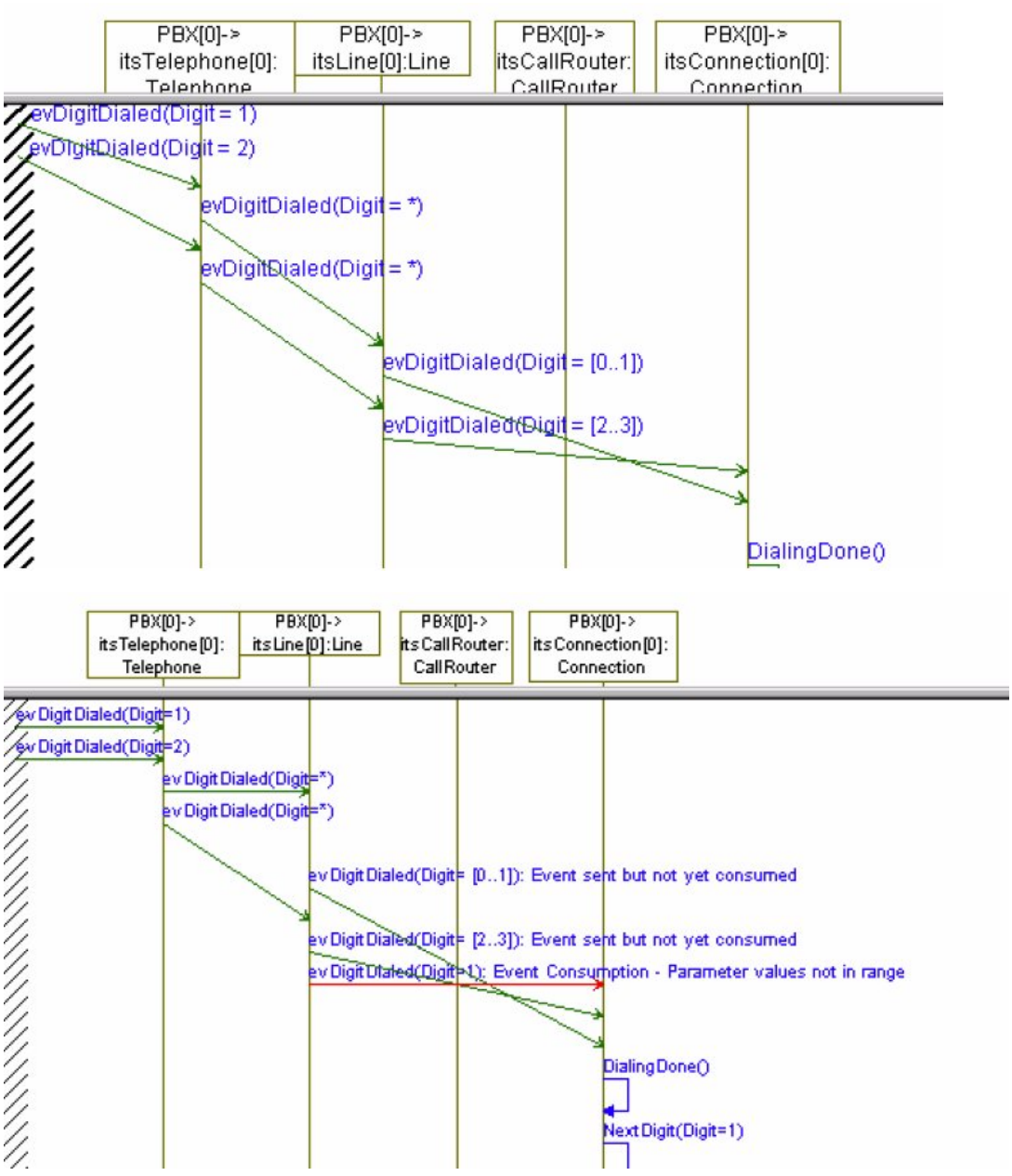
Figure 1: SD with message "Event Consumption – Parameter value do not match"

In this example, according to the specification, TestConductor must monitor

1. The sending of `evDigitDialed(Digit=1)`
2. The sending of `evDigitDialed(Digit=2)`
3. The consumption of `evDigitDialed(Digit=2)`
4. The consumption of `evDigitDialed(Digit=1)`

TestConductor sees, event consumption of `evDigitDialed()` came in-order, but the value of the parameter does not match. Thus TestConductor gives the warning "`Event Consumption - Parameter values do not match`".

Event consumption in-order, but parameter values not in range



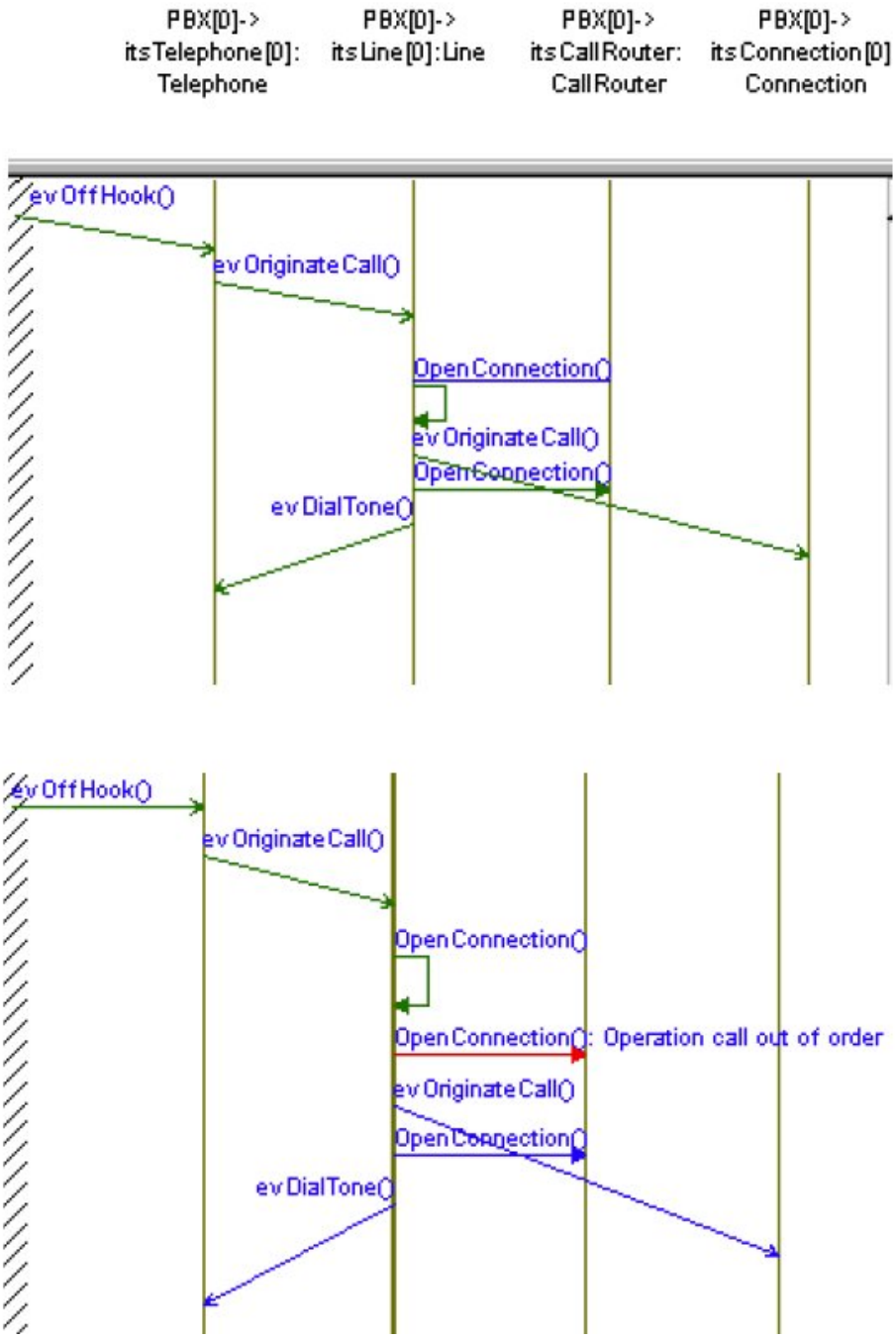
In this example, according to the specification, TestConductor must monitor

1. The sending of `evDigitDialed(Digit=[0..1])`

2. The sending of `evDigitDialed(Digit=[2..3])`
3. The consumption of `evDigitDialed(Digit=[2..3])`
4. The consumption of `evDigitDialed(Digit=[0..1])`

TestConductor sees, event consumption of `evDigitDialed()` came in-order, but the values in the event consumption does not fall in range specified. Thus TestConductor gives the warning "`Event Consumption - Parameter values not in range`".

Operation call out-of-order



In this example, according to the specification above, TestConductor must monitor

1. The self message `OpenConnection()`
2. The sending of `evOriginateCall()`
3. The operation call `openConnection()`

Operation call `OpenConnection()` from `Line` to `CallRouter` should occur after sending of the event `evOriginateCall()`. Thus TestConductor reports the failure "`Operation Call out of Order`".

Operation call in-order, but parameter values do not match

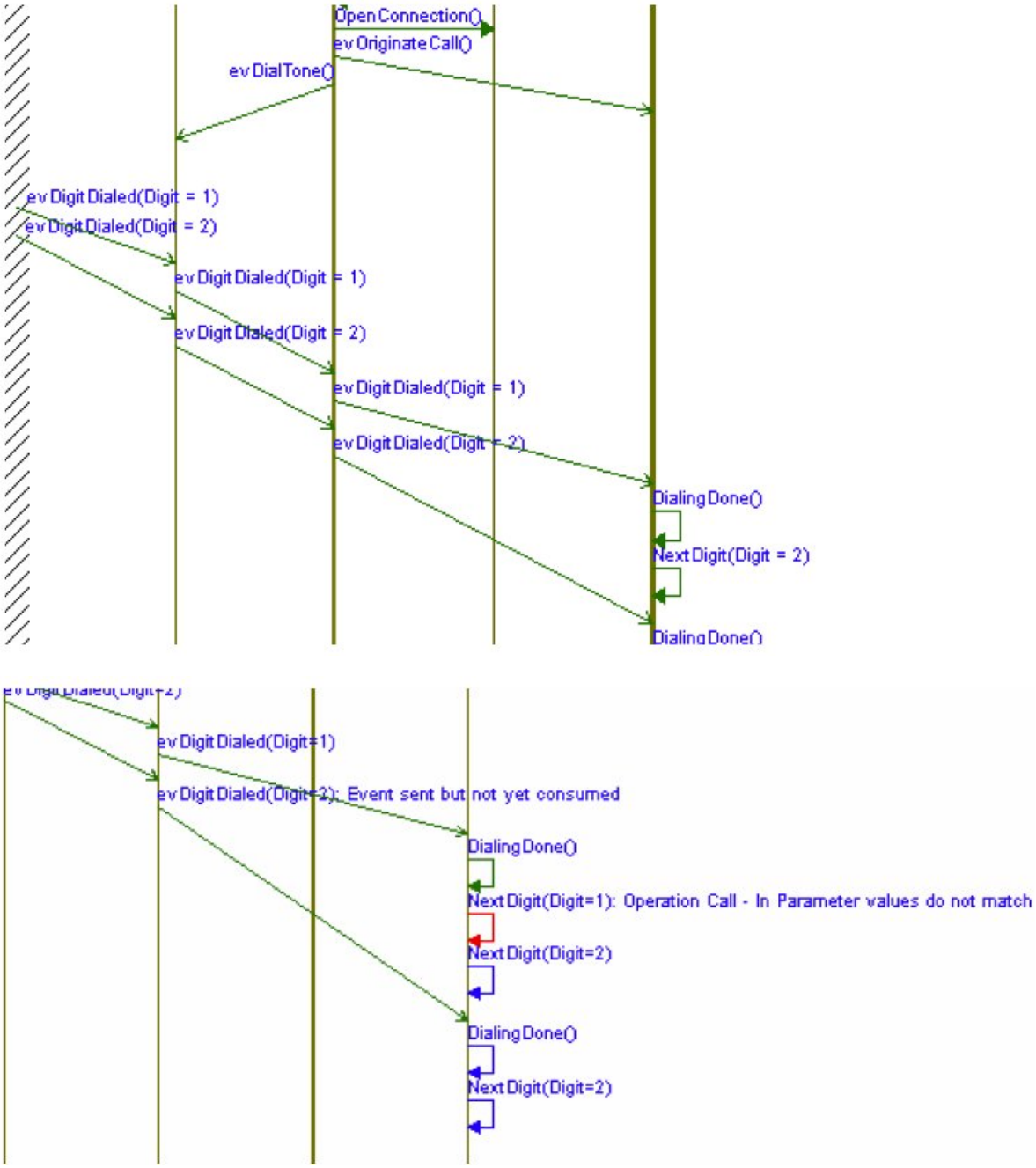


Figure 2: SD with message "Operation call – In Parameter value do not match"

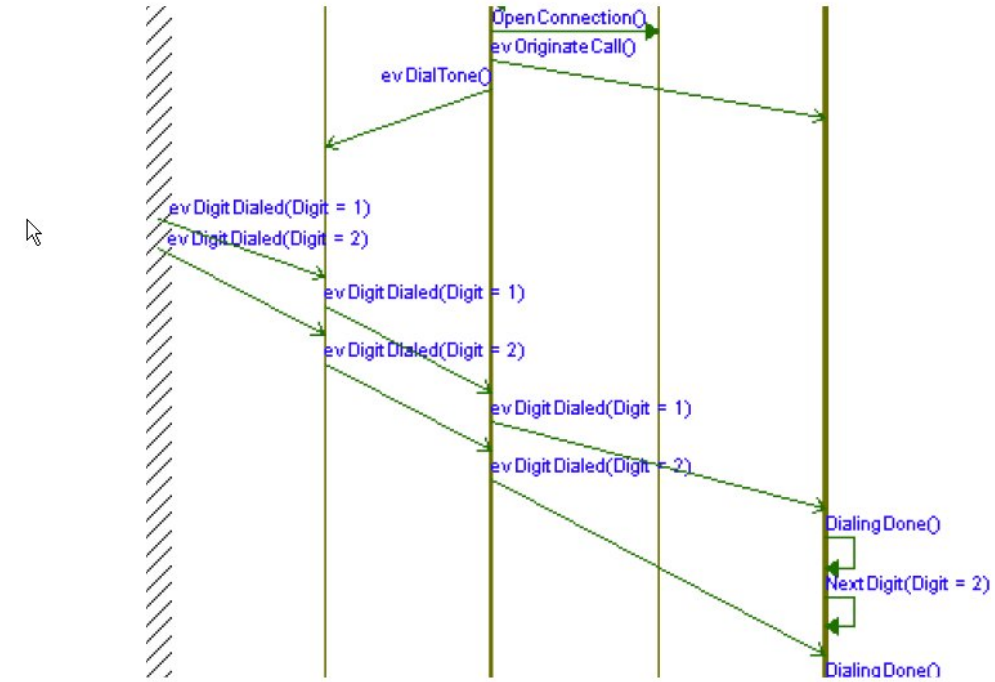
In this example, according to the specification, TestConductor

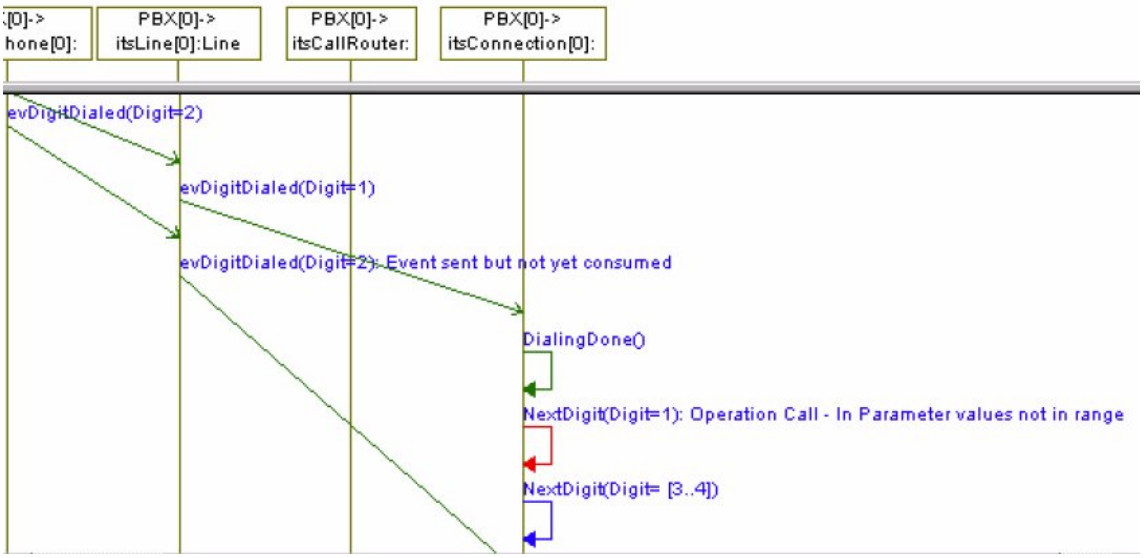
1. Should monitor the operation call `DialingDone()`

2. Must monitor the operation call `NextDigit(Digit=2)`

TestConductor sees that operation call `NextDigit(Digit=1)` instead of operation call `NextDigit(Digit=2)`. Here the operation call has come in order but the parameter value is incorrect. Thus TestConductor gives the warning "`Operation Call:In Parameter values do not match`".

Operation call in-order, but parameter values not in range





In this example, according to the specification, TestConductor

1. Should monitor the operation call `DialingDone()`
2. Must monitor the operation call `NextDigit(Digit=2)`

TestConductor expects operation call `NextDigit(Digit=[3..4])` as specified in the tolerance in the test definition, but sees operation call `NextDigit(Digit=2)` which is out of the range. Here the operation call has come in order but the parameter value is incorrect. Thus TestConductor gives the warning "`Operation Call:In Parameter values not in range`".

Operation call returned - Return value does not match

Here TestConductor expects a return value of `5` as of the specification but sees a `4`. Thus TestConductor gives the warning message "`4=op_int(a=1,b=2,c=3) Operation Call returned - Return value does not match. Expected values are: 5=op_int(a=1,b=2,c=3)`"

Operation call returned - Out Parameter values do not match

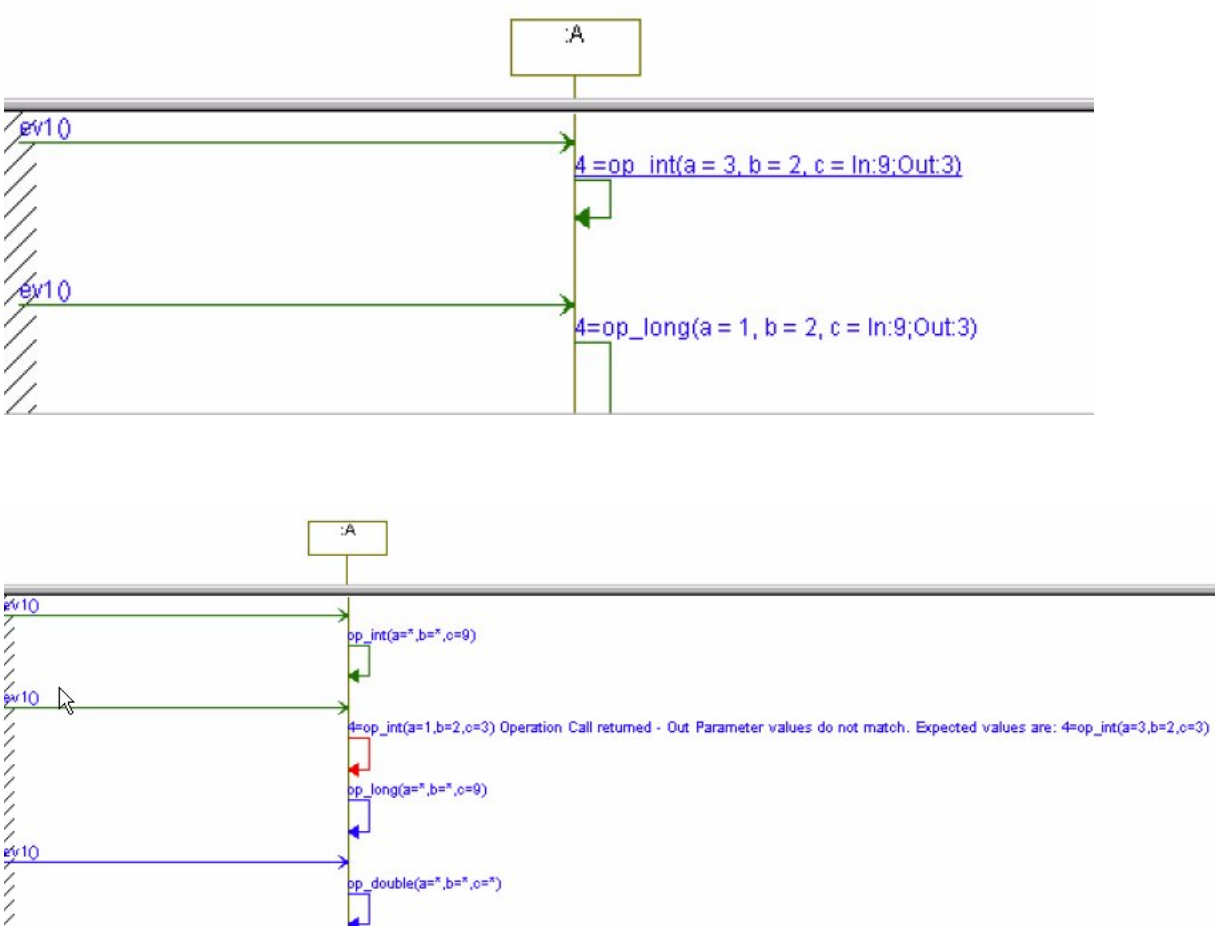
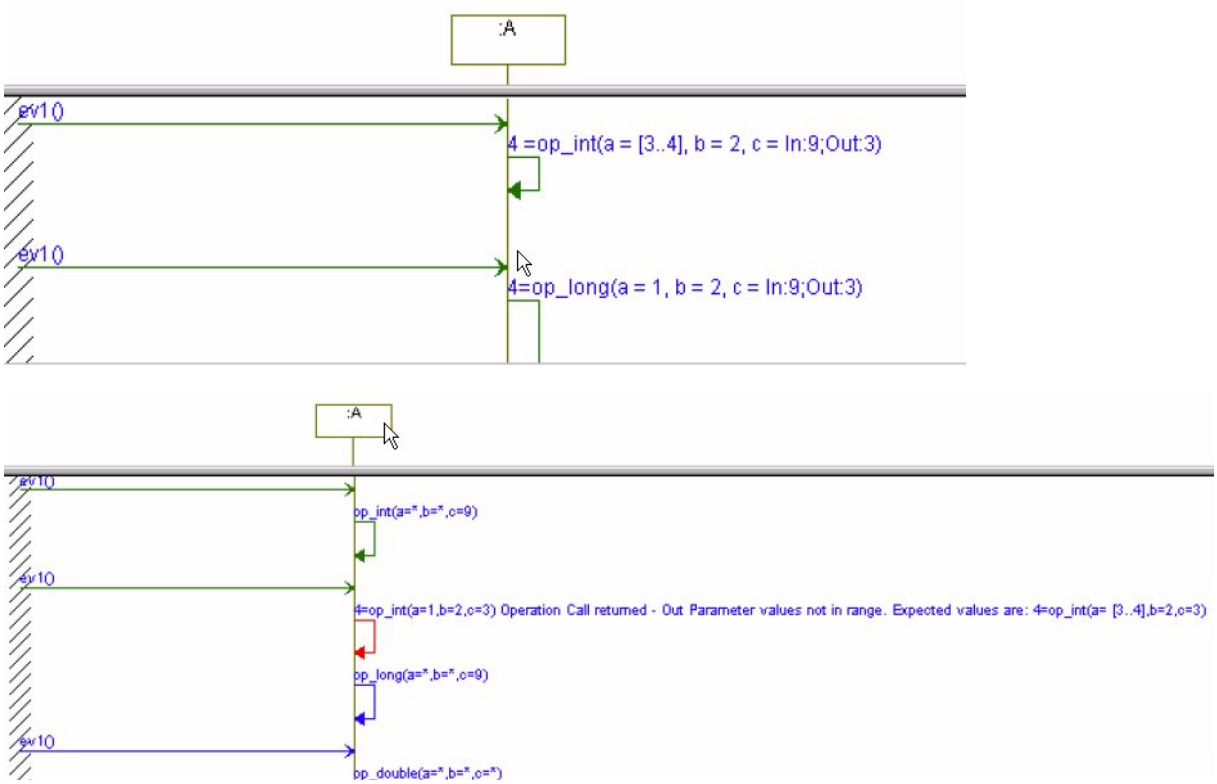


Figure 3: SD with message "Operation call returned – Out Parameter value do not match"

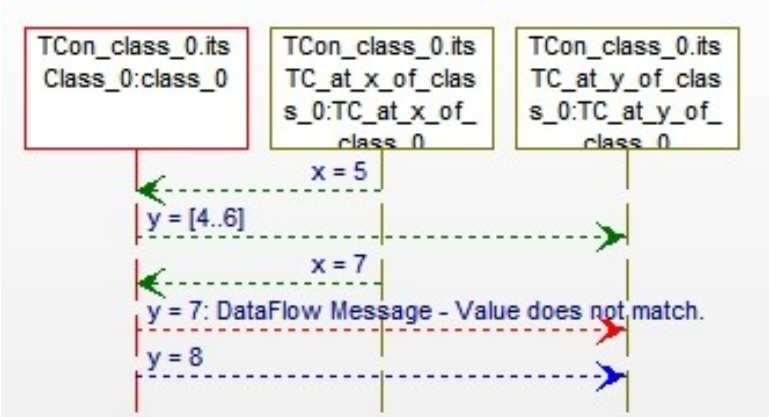
Here TestConductor expects a value of `3` as of the specification but sees `1`. Thus TestConductor gives the warning message "`4=op_int(a=1,b=2,c=3) Operation Call returned - Out Parameter values do not match. Expected values are: 4=op_int(a=3,b=2,c=3)`"

Operation call returned - Out Parameter values not in range



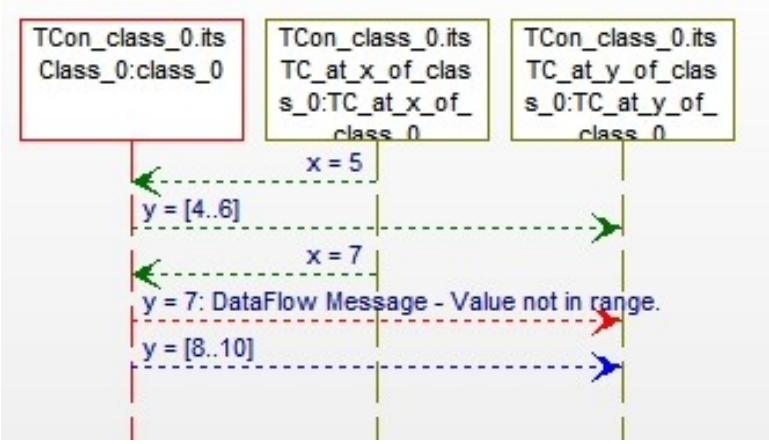
Here TestConductor expects the value in the range of `[3..4]` as of the specification but sees `1`. Thus TestConductor gives the warning message "`4=op_int(a=1,b=2,c=3) Operation Call returned - Out Parameter values not in range. Expected values are: 4=op_int(a= [3..4],b=2,c=3)`"

DataFlow Message - Value does not match



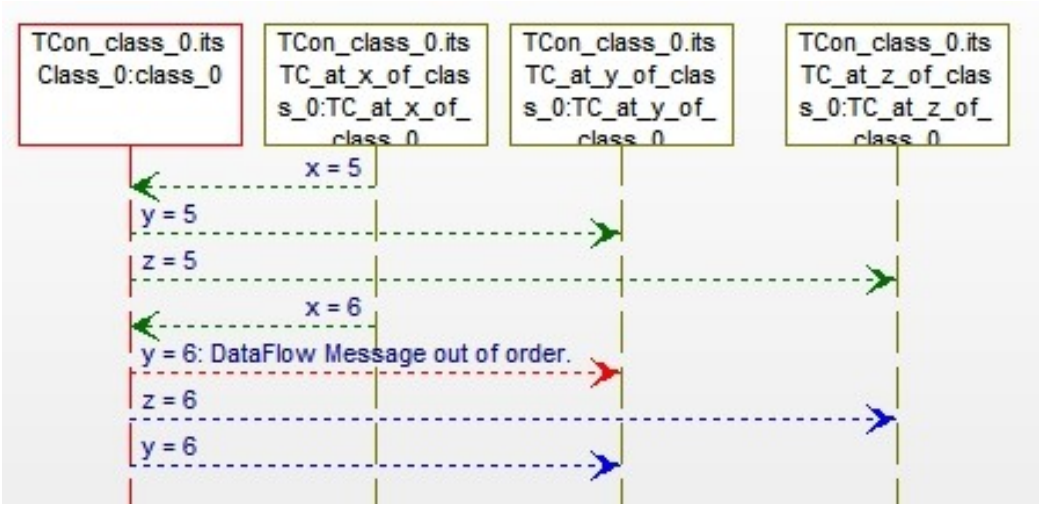
TestConductor expects dataflow 'y=8' but actually observed 'y=7'.

DataFlow Message - Value not in range



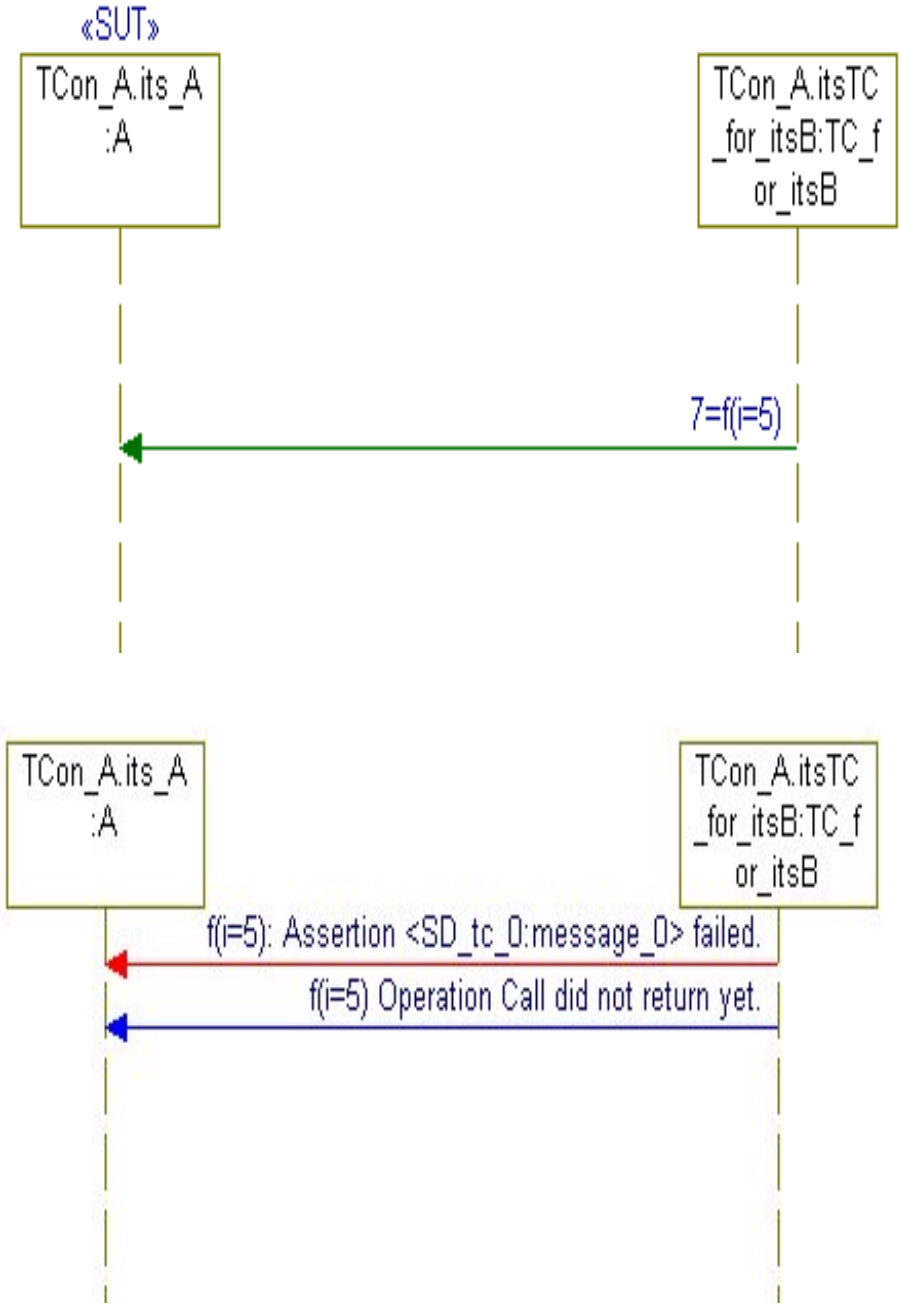
TestConductor expects y to be within range [8..10] but actually observed ‘y=7’, i.e. outside the expected range.

DataFlow Message out of order



TestConductor expects dataflow order ‘z=6’ before ‘y=6’ but avtually observed ‘y=6’ before ‘z=6’.

Assertion failed



When using test components to call operation from a SUT, TestConductor can observe return values from this operation via an assert marco. TestConductor automatically generates the *RTC_ASSERT_SD* macro in the driver operation of the test component:

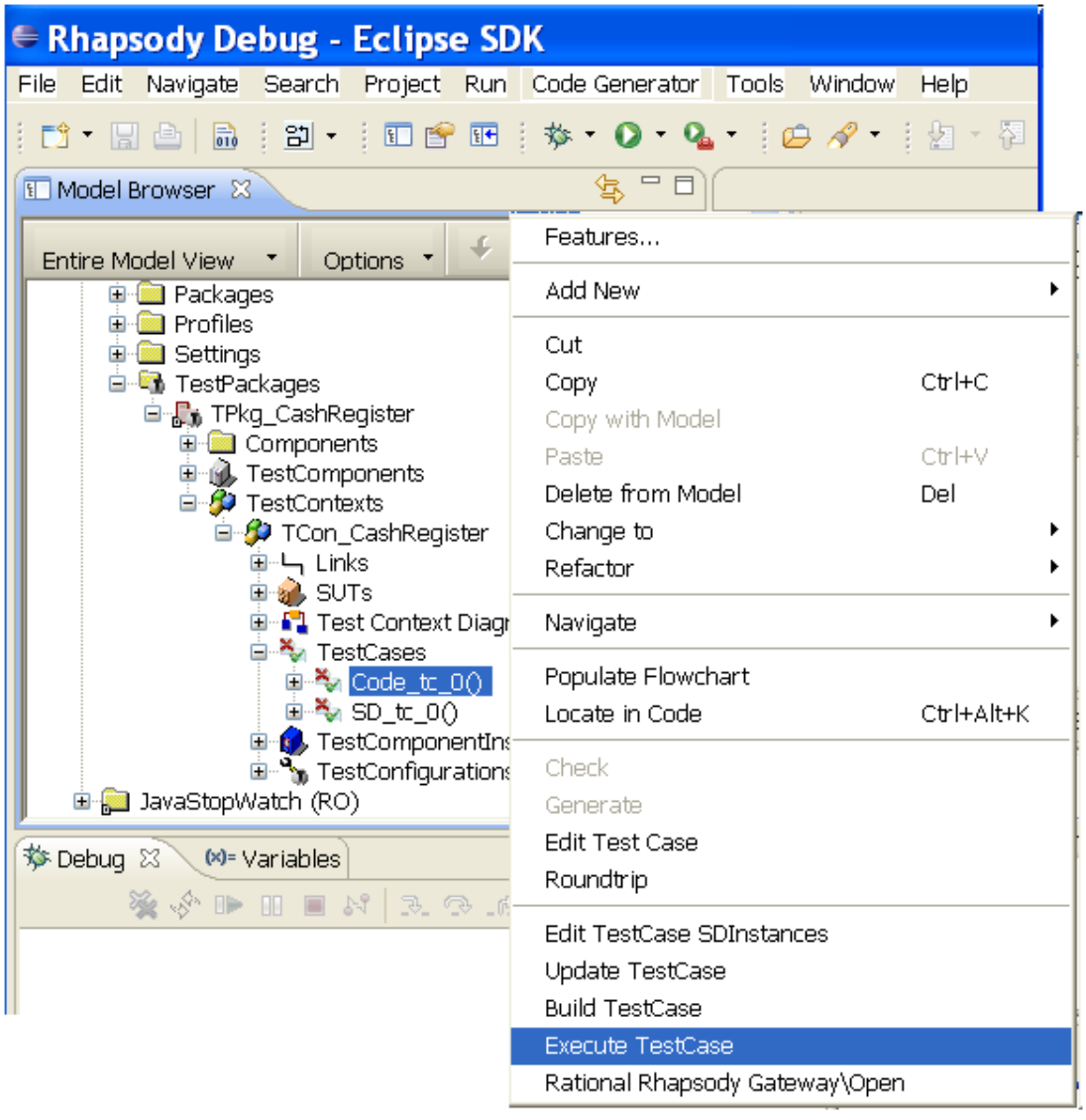
```
//---------------------------------------------------------------
// Driver Initialisation Code:
//---------------------------------------------------------------
```

```
int osc_ret;
int osc_arg_1 = 5;
//---------------------------------------------------------------
// Driver Call Code:
//---------------------------------------------------------------

osc_ret = itsA->f(osc_arg_1);
RTC_ASSERT_SD("SD_tc_0","message_0",osc_ret==7);
```

In this test scenario TestConductor expects a return value of `7 when calling f(I=5) on the SUT, but the actual returned value is different.` Thus, TestConductor gives the warning message "`Assertion <SD_tc_0:message_0>". The second message "f(i=5) Operation Call did not return yet.`" Occurs, because TestConductor interrupts the execution after detecting a failing assertion.

Using TestConductor from Eclipse

As an alternative to the standalone Rhapsody application, Rhapsody can also be used directly from Eclipse (Rhapsody platform integration with Eclipse, see “Integrating Rational Rhapsody and Eclipse” in the Rhapsody online documentation in the IBM knowledge center). Also TestConductor can be used directly from Eclipse when using Rhapsody platform integration with Eclipse; TestConductor does not support Rhapsody workflow integration with Eclipse. In general, all TestConductor functionality can be used when working with Eclipse. Similar to the standalone Rhapsody application, almost all TestConductor functionality is available in context menus of Rhapsody elements, and this holds also when working from Eclipse as can be seen in the following picture:



However, there are some differences that needs to be considered when using TestConductor from Eclipse:

- In contrast to executing TestConductor from the standalone Rhapsody application, the test execution windows of TestConductor are not always in front of the Eclipse main window. Selecting the Eclipse main window may hide the TestConductor test execution windows.
- In Eclipse, when creating a new test architecture, TestConductor automatically creates a new Eclipse configuration instead of a normal Rhapsody configuration. Additionally, TestConductor automatically launches the Eclipse New Project Wizard that can be used to create a new Eclipse project that is connected to the created Eclipse configuration.
- TestConductor does not support Rhapsody workflow integration with Eclipse.
- TestConductor does not support computation of code coverage when using Rhapsody platform integration with Eclipse.

Using TestConductor from Rational Quality Manager

TestConductor test cases can be referenced and executed from Rational Quality Manager. A detailed description how to integrate Rational Quality Manager and TestConductor can be found

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

TestConductor Rhapsody Plugins

TestConductor installs some Rhapsody plugins with additional functionality. The plugins are integrated in the TestConductor Testing Profile, this means the plugins are available for Rhapsody projects containing the Testing Profile.

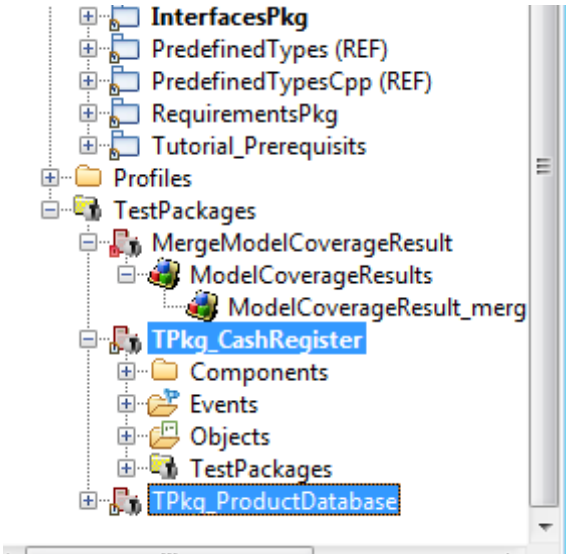
TestConductor Merge Coverage Reports Plugin

The plugin offers the functionality to merge several model coverage reports into one combined report and to merge several code coverage reports into one combined report.

Note: The plugin supports only merging of model or code coverage reports which have been created with Rhapsody 8.0.3 or higher. Merging of reports generated with previous releases of Rhapsody is not supported.

Merging model coverage reports

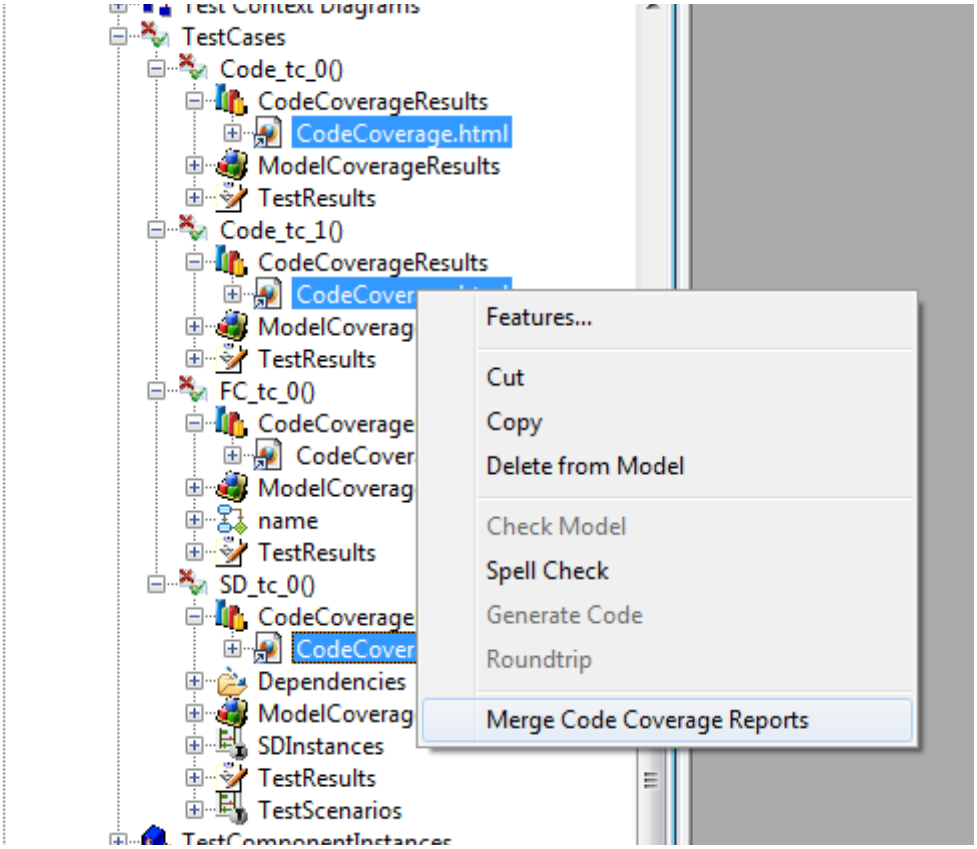
This function can be invoked using the menu helper 'Merge Model Coverage Reports'. The helper is available on TestPackages and supports multi selection. After invocation, the helper collects all model coverage reports inside the selected TestPackage(s) and merges them into one combined model coverage report which is added to the model. The combined report contains a list of the merged reports.
If one TestPackage is selected, the combined report is added to this TestPackage. If multiple TestPackages are selected the combined report is added to the joint parent TestPackage of the selected TestPackages (if exist) or to a TestPackage 'MergeModelCoverageResults if the joint parent of the selected TestPackages is the project itself.



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).

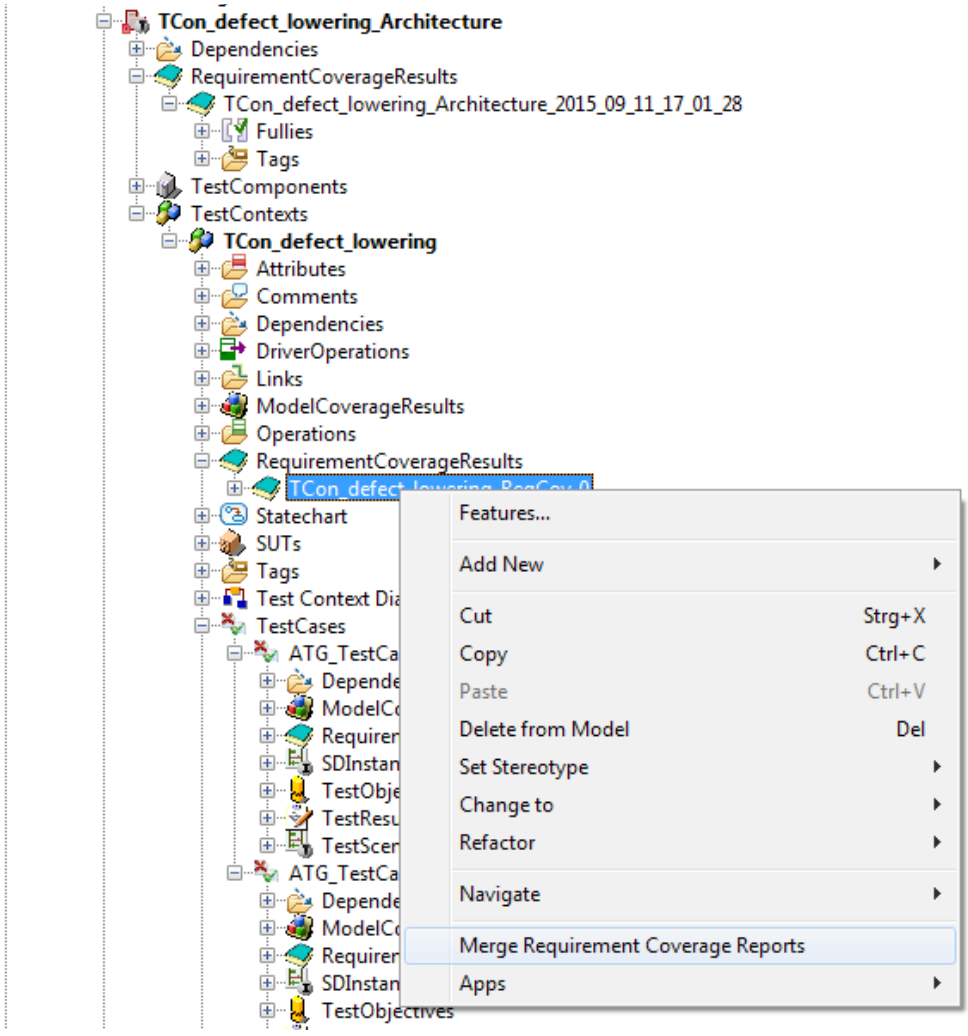


Merging requirement coverage reports

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.

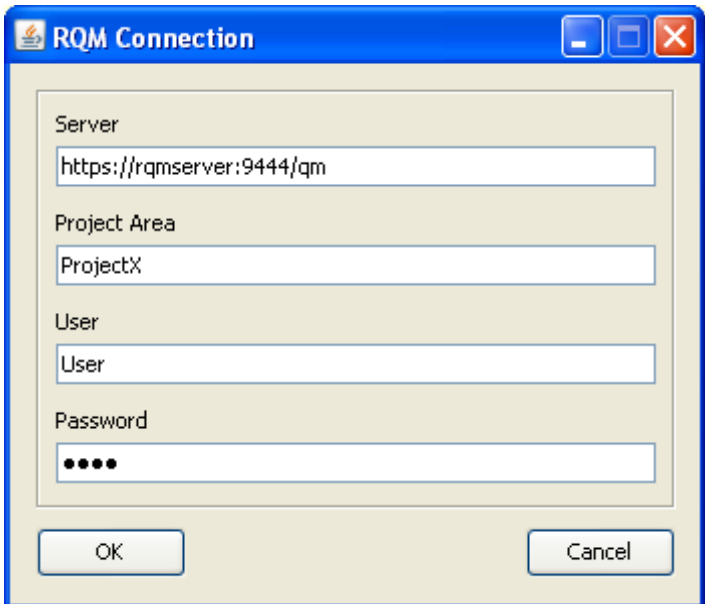


Note: Requirement coverage reports can only be merged if the settings the reports have been generated with (stored in their model based testing tags) are identical. If the settings of different requirement coverage reports are not compatible only a subset of the selected requirement coverage reports 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 RQM Plugin

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

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



After that, a RQM command line TestScript will be created in the specified ProjectArea. The required fields of the command line TestScript like the path to the used Rhapsody model or the full model path to the element which should be tested are set automatically. If additional options should be specified for the test, the necessary adaptions have to be done manually.

If the model is located on a RDM (Rational Design Manager) Server, the execution variables SERVER_URL, PROJECT_AREA_NAME, STREAM_NAME, USER_NAME and PASSWORD are automatically added to the TestScript.

In RQM, the TestScript can now be executed using the TestConductor RQM Adapter as described in the document “RQMTestConductorAdapter_HowTo.pdf”

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

Note: This functionality is not available when using Rhapsody in Eclipse platform integration.

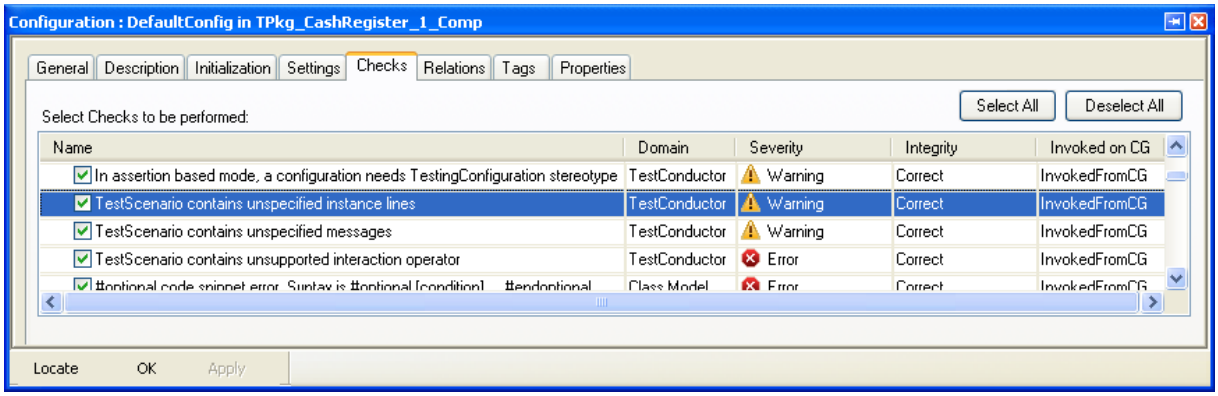
TestConductor Check Model Plugin

Rhapsody has a checker feature which provides the possibility to perform structural and behavioral checks of the model. In addition to the predefined internal checks which are included in Rhapsody, further external checks can be defined and added to the list of checks.

The model checks can either be performed for the active configuration or for selected classes (Tools -> Check Model). The TestConductor checks are also automatically invoked from the code generation.

More information about Rhapsody model checks in general can be found in the Rhapsody User Guide in the chapter 'Checks'.

If the TestingProfile is loaded, the external TestConductor model checks are available. For these checks TestConductor is set as its domain.



The following TestConductor checks are currently available:

- TestScenario contains unsupported SD operator (Warning)
- TestScenario contains unspecified messages (Warning)
- TestScenario contains unspecified instance lines (Warning)
- In assertion based mode, configuration needs <<TestingConfiguration>> stereotype (Warning)

Appendix

TestConductor Assert Macros (C/C++), TestConductor assert methods (Java), TestConductor assert functions (Ada)

As described in chapter Test Case Definition with Code on page *47* and in chapter *Test Case Definition with Flow Charts on page 51* and in chapter TestCase Definition with Statecharts on page 54, pre-defined assertion macros are used to get results from a test case 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 test case 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 test case is aborted immediately without executing further assertions.
- RTC_ASSERT_SD (`sd_instance_name, msgid, e`)
  Assertion that can be used within a sequence diagram. If such an assertion is used in e.g. a driver operation or a stub operation, 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 assertion is executed and attached to the specified 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, which

will be concatenated with the result of the assert macro (*PASSED*, *FAILED* etc.) and printed as result message, e.g. “Check of return value failed.”

- RTC_ASSERT_TRUE (`n, e`)
  This assertion is *PASSED*, if `e == TRUE`. Otherwise the result of the assertion is *FAILED*.

- RTC_ASSERT_FALSE (`n, e`)
  This assertion is *PASSED*, if `e == FALSE`. Otherwise the result of the assertion is *FAILED*.

- RTC_ASSERT_EQUAL (`n, e1, e2`)
  This assertion is *PASSED*, if `e1 == e2`. Otherwise the result of the assertion is *FAILED*.

- RTC_ASSERT_NOT_EQUAL (`n, e1, e2`)
  This assertion is *PASSED*, if `e1 != e2`. Otherwise the result of the assertion is *FAILED*.

- RTC_ASSERT_PTR_EQUAL (`n, e1, e2`)
  This assertion is *PASSED*, if pointer `e1` and pointer `e2` are equal (`e1 == e2`). Otherwise the result of the assertion is *FAILED*.

- RTC_ASSERT_PTR_NOT_EQUAL (`n, e1, e2`)
  This assertion 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 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 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 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 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 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 is *PASSED*, if the comparison of the strings `e1` and `e2` is not equal (`e1 != e2`). Otherwise the result of the assertion is *FAILED*.

For Java, TestConductor defines several assertion methods in the class TestConductor. The following methods are available for Java (the semantics is analogues to the C/C++ macros):

- public static void ASSERT_NAME(String n, boolean p)

- public static void ASSERT_SD(String s, String n, boolean p)
- public static void ASSERT_SD_NAME(String s, String n, String m, boolean p)
- public static void ASSERT(boolean e)
- public static void ASSERT_TRUE(String n, boolean e)
- public static void ASSERT_FALSE(String n, boolean e)
- public static void ASSERT_EQUAL(String n, boolean e1, boolean e2)
- public static void ASSERT_NOT_EQUAL(String n, boolean e1, boolean e2)
- public static void ASSERT_STRING_EQUAL(String n, String e1, String e2)
- public static void ASSERT_STRING_NOT_EQUAL(String n, String e1, String e2)

For Ada, TestConductor defines several assertion procedures in the package TestConductor. The following procedures are available for Ada (the semantics is analogues to the C/C++ macros):

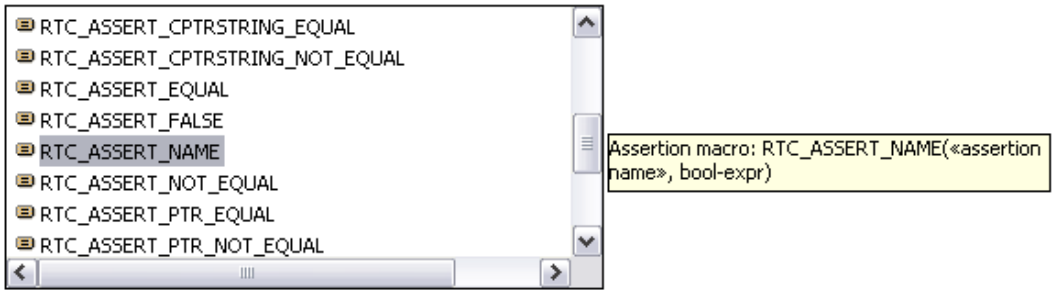
- procedure ASSERT_NAME(n : in String; p : in BOOLEAN; sfile : String := File; iline : integer := Line);
- procedure ASSERT_NAME_FATAL(n : in String; p : in BOOLEAN; sfile : String := File; iline : integer := Line);
- procedure ASSERT_SD(s : in String; n : in String; p : in BOOLEAN; sfile : String := File; iline : integer := Line);
- procedure ASSERT_SD_NAME(s : in String; n : in String; m : in String; p: in BOOLEAN; sfile : String := File; iline : integer := Line);
- procedure ASSERT(e : in BOOLEAN; sfile : String := File; iline : integer := Line);
- procedure ASSERT_TRUE(n : in String; e : in boolean; sfile : String := File; iline : integer := Line);
- procedure ASSERT_FALSE(n : in String; e : in boolean; sfile : String := File; iline : integer := Line);
- procedure ASSERT_EQUAL(n : in String; e1 : in boolean; e2 : in boolean; sfile : String := File; iline : integer := Line);
- procedure ASSERT_NOT_EQUAL(n : in String; e1 : in boolean; e2 : in boolean; sfile : String := File; iline : integer := Line);
- procedure ASSERT_STRING_EQUAL(n : in String; e1 : in String; e2 : in String; sfile : String := File; iline : integer := Line);
- procedure ASSERT_STRING_NOT_EQUAL(n : in String; e1 : in String; e2 : in String; sfile : String := File; iline : integer := Line);

Using IntelliVisor for TestConductor Assert Macros

TestConductor supports the usage of the IntelliVisor functionality of Rhapsody. To be able to use this for the defined TestConductor Assert Macros, you have to prepare Rhapsody's site.prp file. Please do the following steps:

- Close Rhapsody if it is open.
- Copy the file rtc.prp from the ..\TestConductor folder to the ..\Share\Properties folder of your Rhapsody installation.
- Open the site.prp file and add Include "rtc.prp".
- Save the site.prp file and open Rhapsody.

Using Ctrl+Space in a code based test case definition (Flowchart TestCase or Code TestCase) the known IntelliVisor list box opens. With the modifications above you are able to select one of the defined TestConductor Assert Macros. Selecting one of the macros also shows a hint that gives you information about the parameters of the macro.



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.

Syntax for Activation Conditions / Condition Marks

TestConductor uses the following scheme of event activation conditions:

```
ObjectName1->eventAction(ObjectName2,eventName)
```

The scheme of a state activation condition can be represented as follows:

```
ObjectName->stateAction(stateName)
```

The scheme of a method activation condition is as follows:

```
ObjectName1->methodAction(ObjectName2,methodName)
```

In this syntax:

- `eventAction` is `EventSent` or `EventReceived`
- `stateAction` is `StateEntered`, `StateExited` or `IsIn`
- `methodAction` is `MethodCalled` or `MethodReturned`

Note: The syntax of the activation condition is case sensitive. TestConductor checks only the syntax and not for static semantics.

For example:

- `PBX[0]->itsLine[0]->EventSent(PBX[0]->itsTelephone[0],evRing())`
  This activation condition is `TRUE` at the moment when object `PBX[0]->itsLine[0]` sends the event `evRing()` to object `PBX[0]->itsTelephone[0]`. In a sequence diagram, this corresponds to the origin of the message arrow.

- `PBX[0]->itsLine[0]->EventReceived(PBX[0]->itsTelephone[0],evDialTone())`
  This activation condition is `TRUE` at the moment when the object `PBX[0]->itsTelephone[0]` receives the event `evDialTone()` from object `PBX[0]->itsLine[0]`. In a sequence diagram, this corresponds to the end point of the message arrow.

- `line->MethodCalled(callRouter,OpenConnection())`
  The activation condition is `TRUE` at the moment when the `line` object calls the `OpenConnection()` method of the `callRouter` object.

- `line->MethodReturned(callRouter,OpenConnection())`
  The activation condition is `TRUE` at the moment when the `callRouter` object returns the `OpenConnection()` operation call to the `line` object.

- `telephone->StateEntered(ROOT.Ready.Calling)`
  The activation condition is `TRUE` at the moment when object `telephone` enters its "Calling" state chart state.

- `telephone->StateExited(ROOT.Ready.Calling)`
  The activation condition is `TRUE` at the moment when the `telephone` object exits its “Calling” state chart state.

- `telephone->IsIn(ROOT.Ready.Calling)`
  The activation condition is `TRUE` as long as the `telephone` object is in its “Calling” state chart state.

Note: You must specify the full state chart state name (the state path), e.g. “`ROOT.Ready.Calling`.” You can combine these expressions with `AND`, `OR`, and `NOT`.

For example:

```
(NOT (callersLine->EventReceived(caller,evRing()))) OR
(caller->StateEntered(ROOT.Ready.Idle))
```

Do not use two different event conditions with the conjunction `AND` as a combined activation condition. Such expressions can never have the value `TRUE`, because TestConductor and the Rhapsody animation tool work sequentially. At most, one event can be sent or received at every point in time. In addition, be careful when combining several state conditions by the conjunction `AND`: every object can stay in one “basic” state at every point in time, if its state chart does not contain a hierarchical state with orthogonal components. In addition, you can use the name `ENV` as an object name to specify event sending to and receiving from the system’s environment.

Activation conditions use the following shortcuts:

- `ES` for `EventSent`; `ER` for `EventReceived`
- `MC` for `MethodCalled`; `MR` for `MethodReturned`
- `SE` for `StateEntered`; `SX` for `StateExited`; `II` for `IsIn`

TestConductor Messages

Errors/Warnings regarding messages in Sequence Diagrams

Some sequence diagram features are not supported by TestConductor. They will be ignored and a warning comes up, but the test will be executed.

- Timeouts will be ignored.
- Cancelled timeouts will be ignored.
- Reply messages will be ignored.
- Execution occurrences will be ignored.
- Rhapsody in C initializers will be ignored.
- Rhapsody in C++/ Rhapsody in Java constructors will be ignored.
- Rhapsody in C cleanup operations will be ignored.
- Rhapsody in C++/ Rhapsody in Java destructors will be ignored.
- <name> : Unspecified messages will be ignored.
- <name> : Unrealized messages to an internal instance will be ignored.
- Messages with wrong syntax will also be ignored in test execution:
- Condition : <name> is not a valid expression.
- Time interval with a lower bound of 0 will be ignored.
- Time intervals are only supported on system border. Other time intervals will be ignored.
- <name> : Wrong syntax of time interval. Time interval will be ignored.
- Time intervals are only allowed for driver or black box tests. In monitor tests time intervals will be ignored.
- <name> : Method not supported by method broker. Remove message from sequence diagram. (only Rhapsody in Java)

Errors Regarding Complete Sequence Diagrams and Test (test will not be executed)

- In a black box test only messages from or to the system border are used for the test. If a sequence diagram only has internal or unsupported messages, a black box test will not be executed.
  ```
  SD has only internal Messages or unsupported elements.
  Black-Box test will not be executed.
  ```

- If a sequence diagram is empty or only has unsupported messages, the test will not be executed
  ```
  SD contains only unsupported elements. Compilation
  aborted. SD without any constructs is not supported.
  ```

- In some cases executing a test with a sequence diagram which hat more than 2000 messages leads to a crash due to a small stack size. In this case, please refer to the release notes how to increase the stack size of your system.
  ```
  Due to the actual size of this SD, test execution can
  ```

```
lead to a crash. In such a case, please contact support
to get a patch or refer to the release notes and use
the mentioned workaround.
```

- If two messages of a sequence diagram start/end at the same point TestConductor can not get correct information about the messages so the compilation fails. If this happens, make sure that there is only one message starting/ending on each message point.

```
TEST: <name>
Sequence Diagram: <name>
ERROR: Compilation error - Test will not be executed.
This error can have different reasons. Known reasons
are:
- Sequence Diagram contains a time interval beginning
or ending on other message points.
- Sequence Diagram contains unspecified messages.
```

- If the activation condition of a test has the wrong syntax the test will not be executed.

```
TEST: <name>
Sequence Diagram: <name>
ERROR: Syntax error in activation condition
<ActivationCondition>
```

- Another message arrow detected between start point and end point of operation.

```
TEST: <name>
Sequence Diagram: <name>
Another message arrow detected between start point and
end point of
Operation <name>.
This is not supported by TestConductor.
To execute the test, please move start/end points of
other messages above or below the message arrow of
<name>.
```

- If there is an unspecified message in the specification sequence diagram

```
<Message_name>: Unrealized Messages to an internal
instance will be ignored.
```

- If there is an unrealized message in the specification sequence diagram

```
<Message_name>: Messages with Stereotype <unrealized>
will be ignored.
```

- If the specification sequence diagram has an unspecified class

```
TEST: <test_name>
Sequence Diagram: <name>
Class of Instance <class_name> is unspecified. Test
will not be executed.
```

Restrictions

TestConductor supports Rhapsody in C/C++/Java/Ada with its existing and with its new features. The most important limitations are:

- Assertion based testing mode is only supported for RhapsodyC and RhapsodyC++.
- Code coverage computation with TestConductor is only supported with assertion based testing mode for RhapsodyC and RhapsodyC++.
- Code or flow chart test cases are only supported for Rhapsody in C/C++.
- Black box production code test case execution only for Rhapsody in C++ and C.
- TestConductor does not support C#.

Limitations of design elements (sequence diagrams)

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

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

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

Note: TestConductor will ignore condition marks during test execution when the syntax of the condition mark is not valid.

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

Functional Limitations

All TestConductor features are available for Rhapsody in C++, C, Java and Ada. Rhapsody Automatic Test Generation (ATG) is only available for Rhapsody in C++. For TestConductor, the most important limitations are:

- Flow chart test cases are only supported for Rhapsody in C/C++.
- Black box production code test case execution only for Rhapsody in C/C++.

Beside the listed important limitation there are some other know limitations:

- Obsolete profiles (ATGProfile, TestingProfile_CPP, TestingProfile_C, TestingProfile_Java, TestingProfile_Ada) must be deleted from models manually.
- Only virtual operations can be stubbed.

- When using animation based testing mode, TestConductor cannot generate stubs for triggered operations.
- 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 TestComponent instance 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.
- Building SUT for black-box testing requires an animation property change in the design model.
- Auto created operations are not animated and cannot be used in test cases: due to a limitation in the Rhapsody animation, auto generated operations like getter/setter for class attributes are not animated during execution, they do not appear in animated sequence diagrams and observers don't get notifications about these messages (even if the property `CG:CGGeneral:GeneratedCodeInBrowser` is set to `true`).