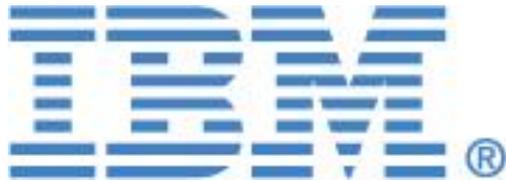


# Tutorial for TestConductor for RiA



Rational. Rhapsody

## RiA Tutorial

for

IBM<sup>®</sup> Rational<sup>®</sup> Rhapsody<sup>®</sup>  
TestConductor Add On



## License Agreement

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

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

## Trademarks

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

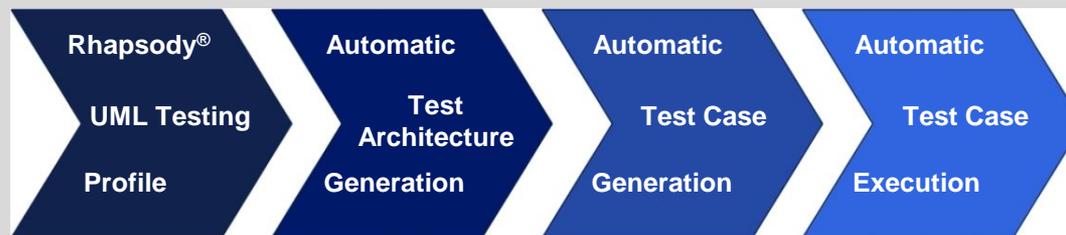
All other product or company names mentioned herein may be trademarks or registered trademarks of their respective owners.

© Copyright 2000-2012 BTC Embedded Systems AG. All rights reserved.

# TestConductor for Rhapsody for Ada

**In this tutorial** we would like to give you an impression of the Rhapsody Testing Environment, which goes beyond current embedded software testing technologies; it ensures that the system can be continuously tested throughout the design process. The Testing Environment and its parts seamlessly integrate in Rhapsody UML and guide the user through the complex process of test preparation, execution and result analysis.

**TestConductor** is the test execution and verification engine in the Rhapsody Testing Environment. It executes test cases defined by sequence diagrams, statecharts, and source code. During execution TestConductor verifies the results against the defined requirements.



*IBM® Rational® Rhapsody® Testing Environment*

# StopWatch Application

The screenshot displays the Rhapsody IDE interface for configuring an event. The top window, titled "Events", shows the configuration for the event "evKey" on the object "myStopWatch[0]". The "Arguments" section contains a table with the following data:

Name	Type	Value
KeyVal	int	

Below the configuration window, a console window shows the output of the application. The output consists of a sequence of time strings, each printed twice with a 0.5-second delay between the two prints of each string. The strings are: 0:0, 0:0, 0:1, 0:1, 0:2, 0:2, 0:3, 0:3, 0:4, 0:4, 0:5, 0:5.

The [StopWatch application](#), the example Ada application for this tutorial, models a simple stopwatch. Make yourself familiar with the use cases of the application. Open the project „AdaStopWatch“ from the folder „Samples/AdaSamples/TestConductor“ in your Rhapsody installation, run the component «StopWatchComp», and use the following input:

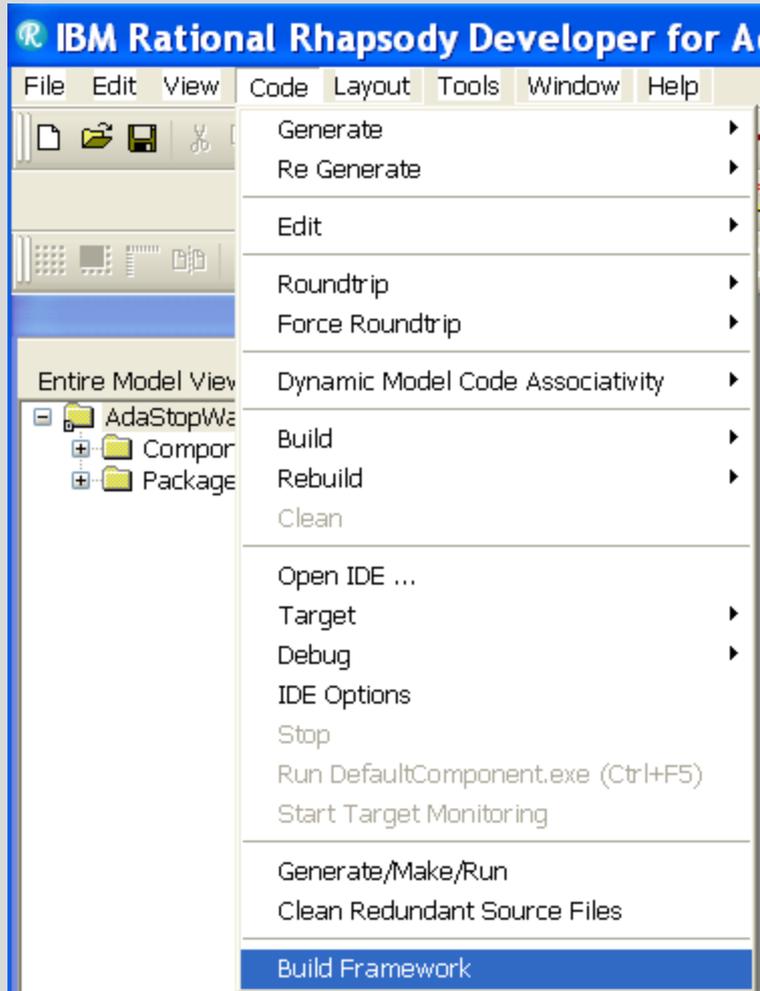
**To start** the application, press “Go” in Rhapsody’s animation toolbar.

**To start** the stopwatch, generate event “evKey(1)” using the animation toolbar.

**To stop** the stopwatch, generate event “evKey(1)” again using the animation toolbar.

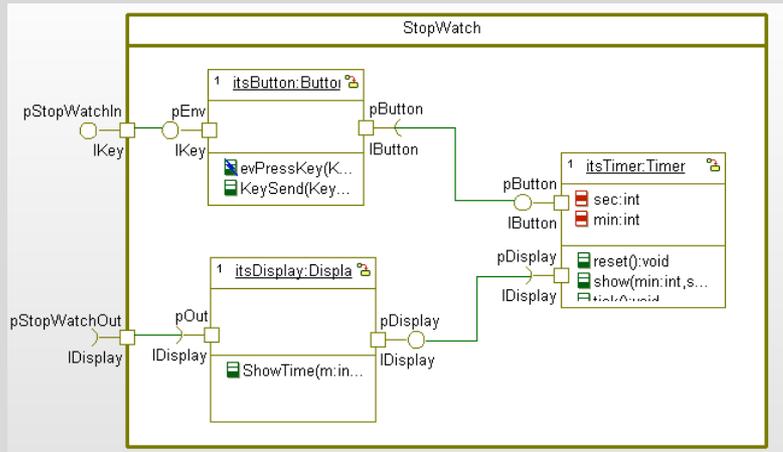
**When running**, the stopwatch outputs the elapsed time in minutes and seconds to the console. Each second is printed twice, one time with a colon and 0.5 seconds later without a colon, similar to a stopwatch with blinking colon.

# Before testing the model

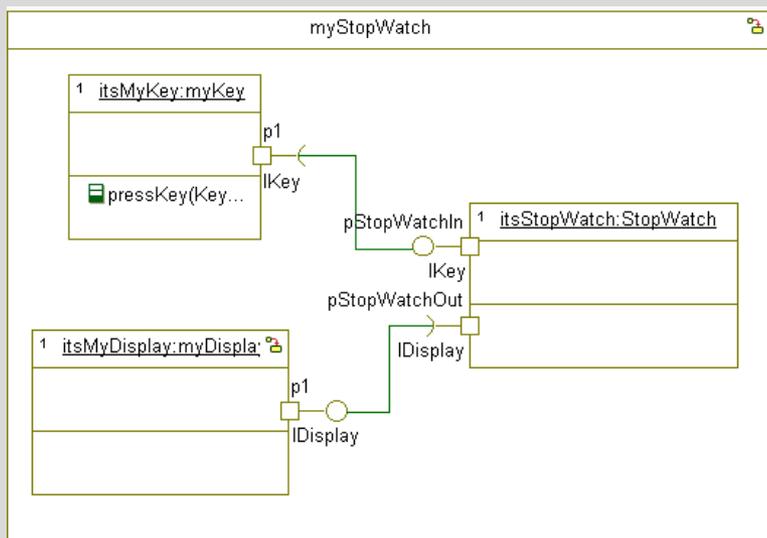


Before testing the tutorial model, one has to rebuild the Rhapsody framework. In order to do this, go to Rhapsody's Code menu and select "Build framework". This rebuilds Rhapsody's framework which is needed before we can test the tutorial model. Please note that this is only needed once, you don't have to rebuild the framework again if you want to test other models.

# StopWatch Model



The StopWatch model contains the StopWatch class and its three parts. The first part is a button that can be used to start and stop the stopwatch. The second part is the timer that is used in order to count the elapsed time. The third part is the display that displays the elapsed time. Within the stopwatch the different components are connected via ports and links. Additionally, the stopwatch class itself relays both the start/stop button and the display to its boundaries in order to be able to connect an external start/stop button and an external display .

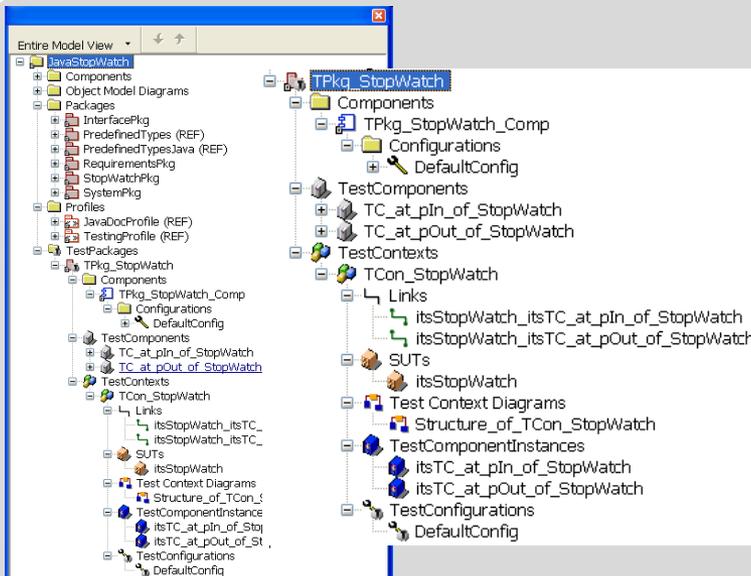


The myStopWatch class represents a sample instantiation of the StopWatch class. It connects a stopwatch to an external key “myKey” that can be used as a start/stop button. Additionally, it connects the stopwatch to an external display “myDisplay” that displays the elapsed time.

# System Under Test

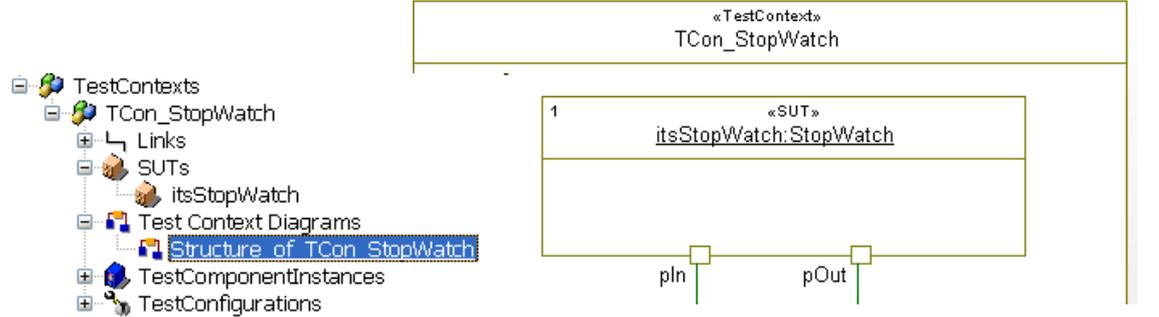
Defining the System Under Test (SUT) is the first step<sup>7</sup> in the test workflow. This tutorial will focus on the StopWatch class. To define StopWatch to be the SUT, we have to create a test architecture. The needed administrative framework will be placed in the folder „TestPackages“.

The System Under Test (SUT) is a part and 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.



1

Select the class „StopWatch“ in the browser and choose from context menu „Create TestArchitecture“.

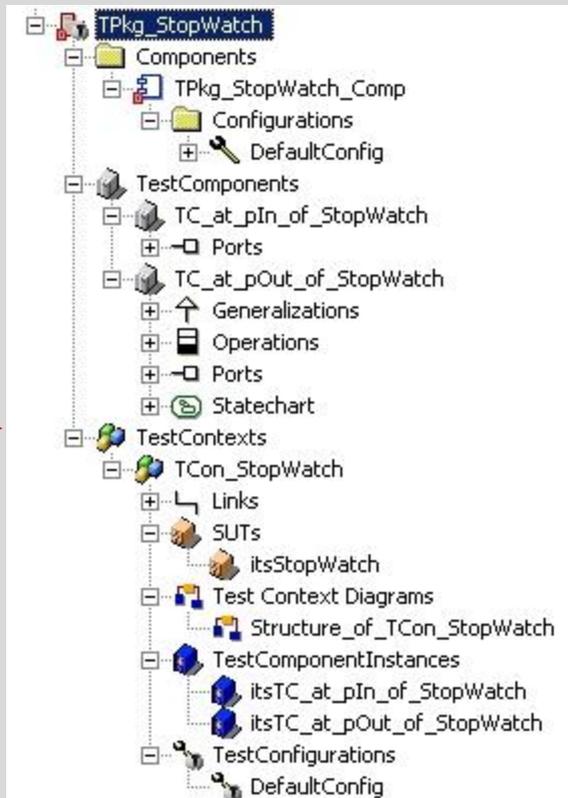


2

Have a look on the newly created Test Context Diagram „Structure\_of\_TCon\_StopWatch“, and view the resulting parts in the composite class „TCon\_StopWatch“ of our test context.

# Test Architecture

The automatically created test architecture is completely represented in the browser and seamlessly integrates into Rhapsody; think of it as an independent test model besides the design model. After creation the following elements are visible:

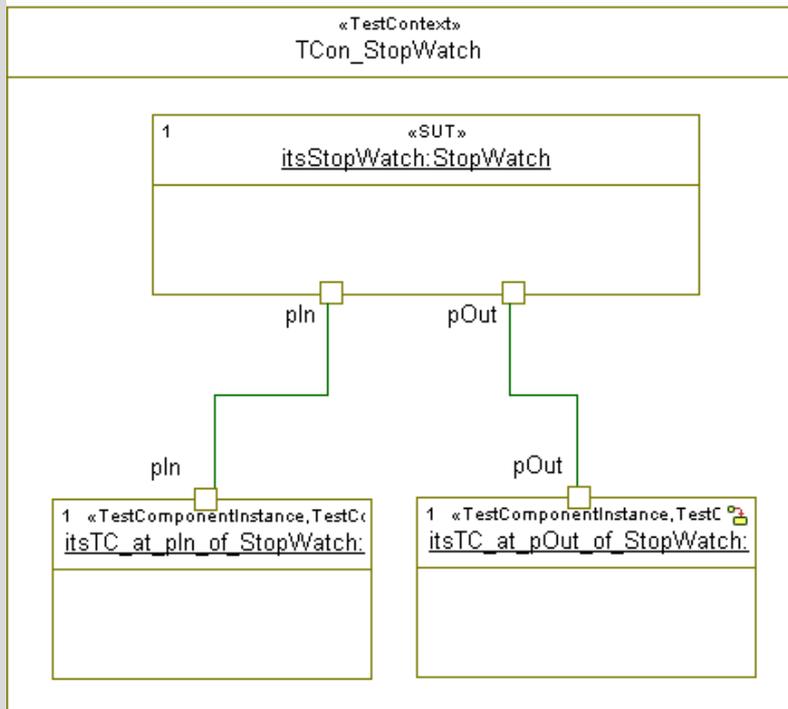


The new configuration under the component „TCon\_StopWatch\_Comp“ describes the collection of test components and SUT objects and their interconnections when a test case is started.

A test component is a class of a test system. Test component objects (test component instances) realize partially the behavior of a test case. A test component might have a set of interfaces via which it might communicate via connections with other test components or with SUT objects.

A test context describes the context in which test cases are executed. It is responsible for defining the structure of the test system. The test components and SUT objects are normally parts of a test context.

The automatically created test context represents the formal structure of the test system. TestConductor analyzed the model structure in consideration of the selected SUT and proposed a test structure, which is visualized in the test context diagram inside the test context. TestConductor generated corresponding test components for ports and associations of the SUT.

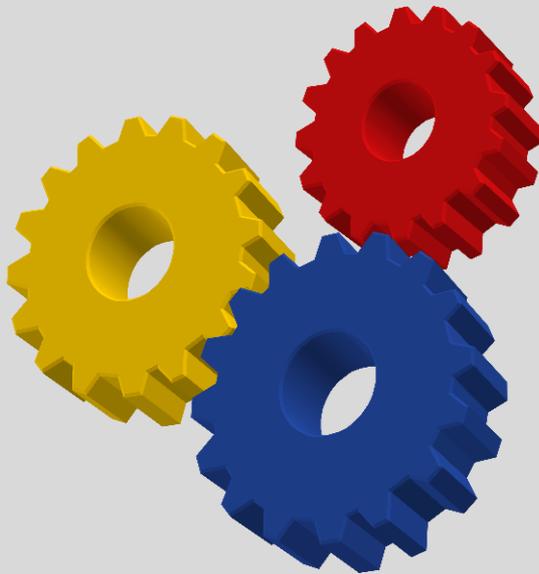


The composite class „TCon\_StopWatch“ is the part container for the SUT object and the created test component objects.

The class „TC\_at\_pln\_of\_StopWatch“ realizes the required interface „IKey“ and thus can be connected to the “pln” port of the stopwatch class that provides this interface.

The class „TC\_at\_pOut\_of\_StopWatch“ provides realizations for the interface “IDisplay” and thus can be connected to the “pOut” port of the stopwatch class.

**Test cases are the soul** of a test system. Until now we created a complete test architecture around the SUT with a few mouse clicks in less than a minute. The established and reviewed test system is linkable and runnable. Well, the body works, let's have a look at the test cases. A test case ...



**is a specification** of one case to test the system including what to test, with which inputs, and what the expected outcomes are. It is defined in terms of stimuli injected to SUT objects and observations coming from SUT objects.

**is an operation** of a test context that specifies how a set of cooperating test components interact with the SUT.

**can be specified** as sequence diagrams, statecharts, and source code.

**can be generated** automatically by using TestConductor's test case wizard.

**can be recorded** as animated sequence diagrams.

**can be created** by hand.

# Test Case Specification

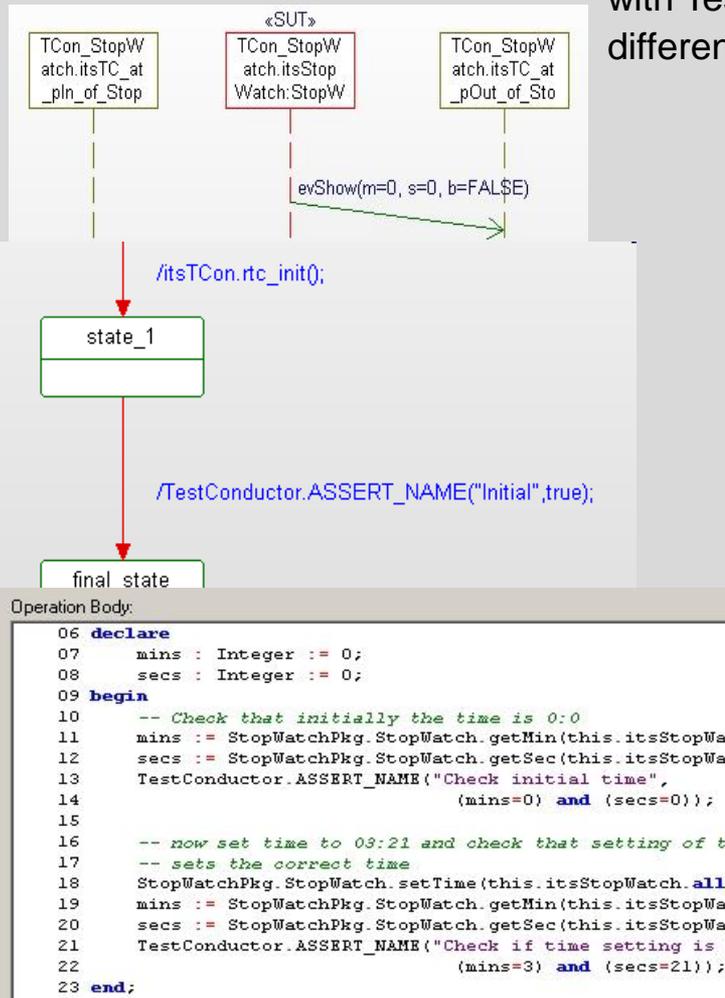
How to manually create test cases and how to execute them with TestConductor will be discussed in the following sections. The different kinds of definitions have their own strengths:

**Sequence diagram** test cases can be recorded automatically or created by hand. In some cases they have already been specified during the analysis phase of the project, and define the actions and reactions of the SUT. The graphical formalism boosts the readability and understanding.

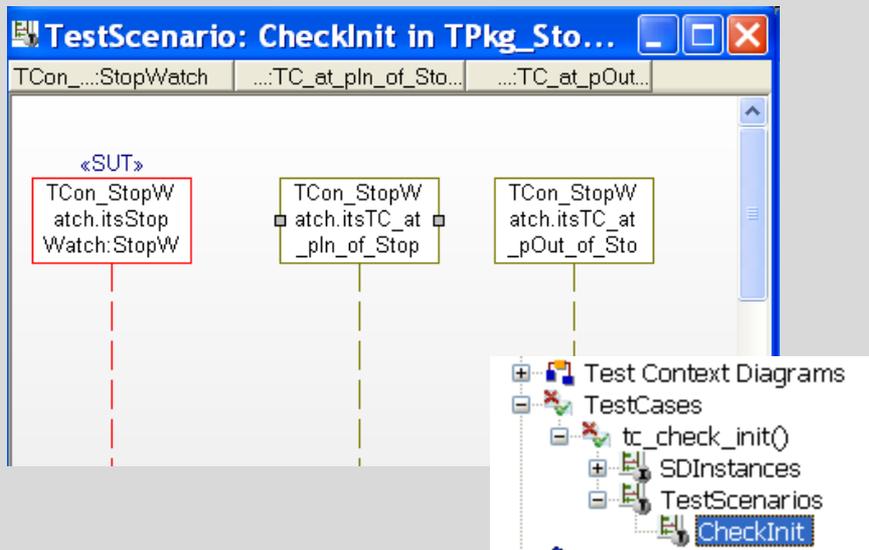
**Statechart** test cases are a well known and convenient means to specify behavior based on states and modes.

**Source code** test cases are often preferred by experienced programmers.

**In summary** TestConductor, the Rhapsody test case execution engine, works with all kinds and combinations of test case definitions.



# Test Case: Sequence Diagram I



To manually create a sequence diagram test case we have to define a test scenario which is represented as a sequence diagram and link it to a test case. TestConductor simplifies this process with a single command.

**1**

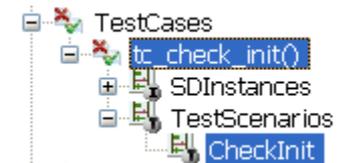
Select the test context „TCon\_StopWatch“ in the Rhapsody-Browser ...

## Create SD TestCase

- Create Flowchart TestCase
- Create Code TestCase
- Create Statechart TestCase

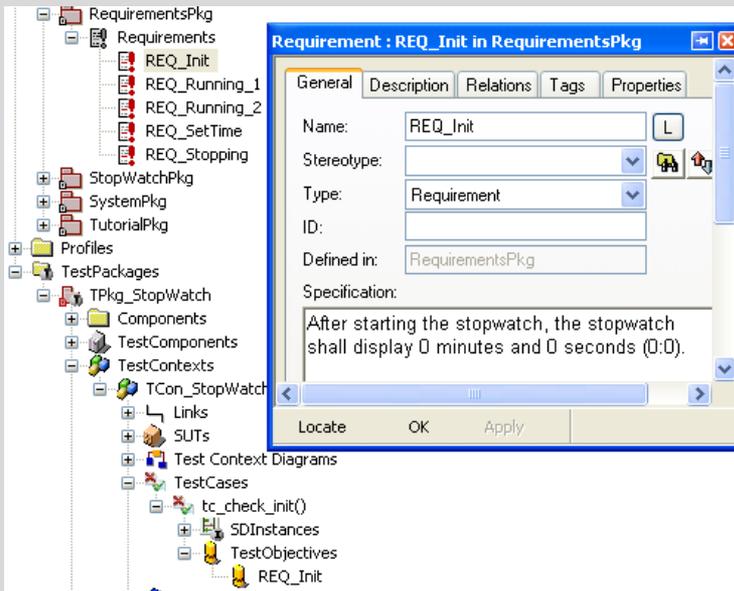
**2**

... and choose from the context menu „Create SD TestCase“..

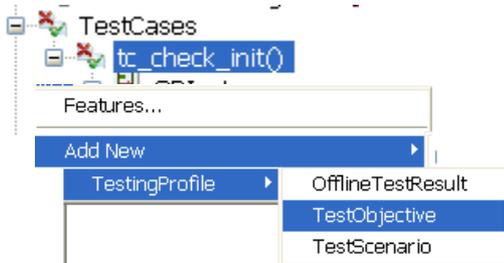
**3**

Rename the test case to „tc\_check\_init“. Rename the test scenario to „CheckInit“ and open it.

# Test Case: Sequence Diagram II

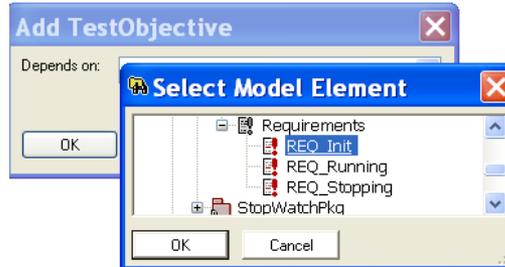


Determine the test objective of the test case: the SD test case should check that requirement “REQ\_Init” is indeed fulfilled by the stopwatch class. To make explicit that the SD test case shall verify this particular requirement, a test objective is added to the test case.



1

Select the test case and select “Add New -> TestingProfile -> TestObjective”



2

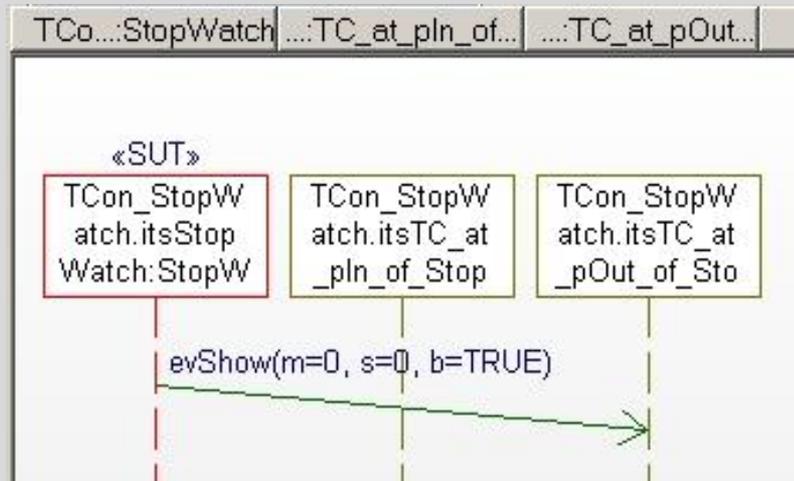
Select requirement “REQ\_Init” as target of the test objective



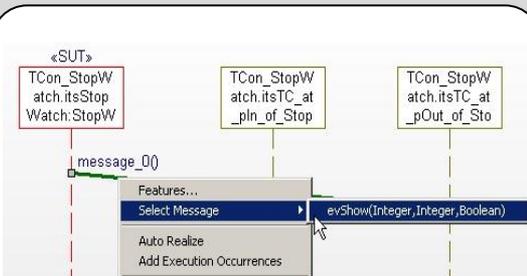
3

The test objective now links the test case to the requirement “REQ\_Init”.

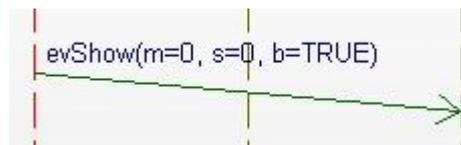
# Test Case: Sequence Diagram III



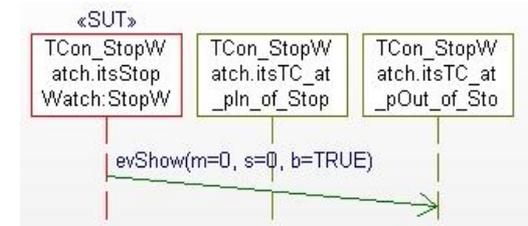
Define **action and reaction** of the system under test. We will specify the „CheckInit“ scenario, where the SUT shall emit event “evShow” with current time 0:0 after starting the SUT. This output shall be generated automatically by the SUT, since no further input is needed for that.

**1**

Draw the message “evShow” from the SUT to the test component “TCon\_StopWatch.itsTC\_at\_pOut\_of\_StopWatch”.

**2**

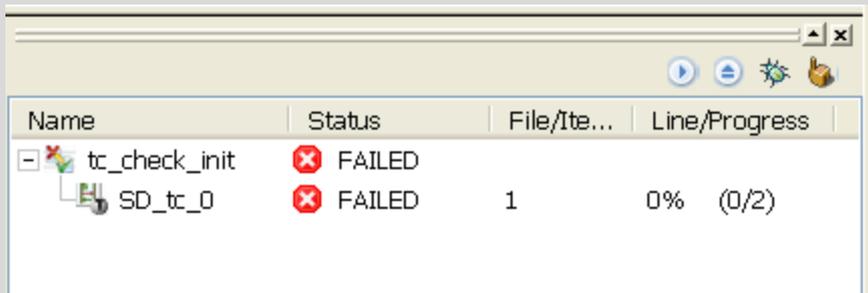
Specify argument values  $m = 0$ ,  $s = 0$ ,  $b = \text{TRUE}$  for the message.

**3**

That's it already. The test case specification is complete.

# Test Case Execution I

Execute the test case with Rhapsody TestConductor. The execute dialog lists all executed test scenarios, their progress and status.

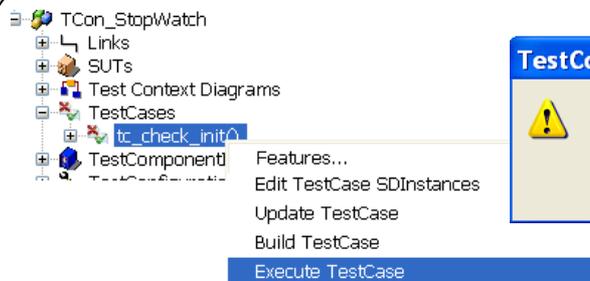


Name	Status	File/It...	Line/Progress
tc_check_init	❌ FAILED		
SD_tc_0	❌ FAILED	1	0% (0/2)

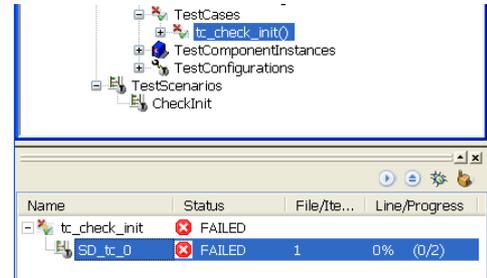
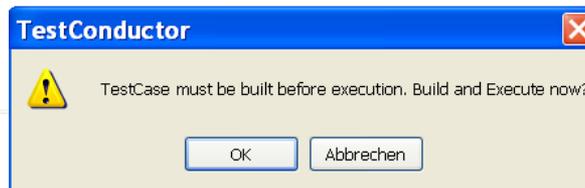
The status, the final result can be either „PASSED“ or „FAILED“.

The progress displays how many steps are finished yet. In case of a passed test 100% have to be achieved.

The buttons in the top right corner of the execution dialog can be used to control actual test case execution and will be explained later.

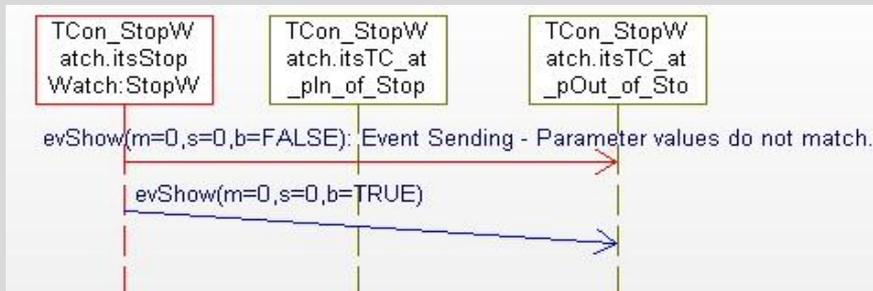
**1**

To execute a test case, simply right-click the test case and select “Execute TestCase” from the context menu. In case the test model needs to be updated and/or the tested executable needs to be compiled, a popup window appears in order to update the test case and/or build the executable.

**2**

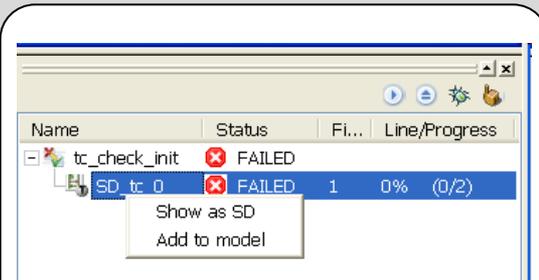
The test case execution dialog is a dockable dialog that can be placed e.g. underneath the main browser window

# Test Case Execution II



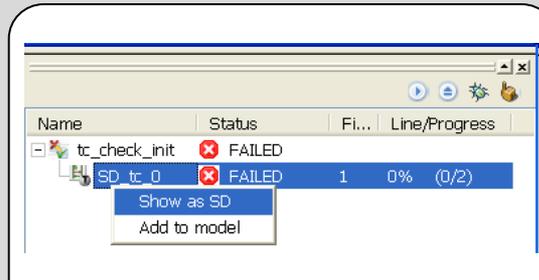
Summary Info	
Total number of SDs used:	1
Total number of SD instances in test:	1
Total number of executed SD instances:	0

The test case execution **FAILED** with Rhapsody TestConductor. To analyze the reason TestConductor offers two kind of views. The HTML-report displays a textual summary and can be found directly under the test case in the Rhapsody-Browser. TestConductor created a debug sequence diagram to display the error. The red arrow visualizes the faulty step and the reason. TestConductor expects the parameter value „TRUE“ for argument „b“, but observes the value „FALSE“ during actual test execution. The expected value was not specified correctly... by accident.



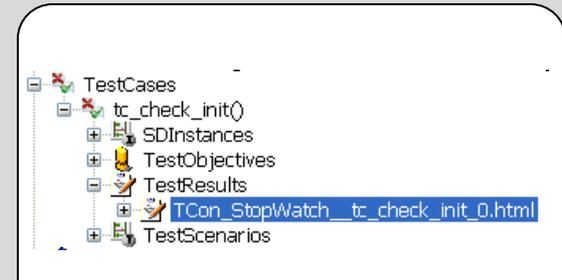
1

To open the debug sequence diagram right click the item SD\_tc\_0 in the TestConductor execution dialog...



2

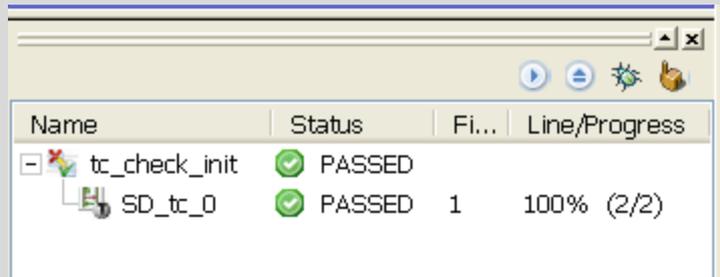
... and select “Show as SD”



3

In the browser, underneath the test case, you can find the generated html report. Double click the report to open it.

# Test Case Execution III



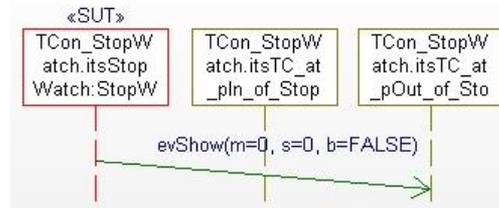
Name	Status	Fi...	Line/Progress
tc_check_init	✓ PASSED		
SD_tc_0	✓ PASSED	1	100% (2/2)

The test execution **PASSED** with Rhapsody TestConductor after we corrected the expected parameter value for argument “b” from “true” to “false” in the test scenario „CheckInit“. After changing the scenario and re-executing the test case, the test case is passed.

Refer to the [user guide](#) to get familiar with the extended functionality of TestConductor.

**1**

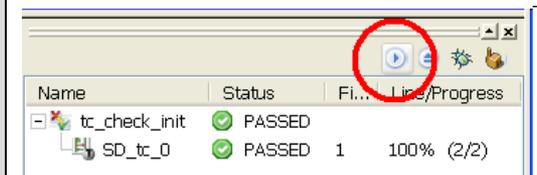
To correct the test case open the test scenario „CheckInit“.

**2**

For argument “b”, change the expected value from “TRUE” to “FALSE”.

**3**

Re-execute the test case by pressing the “Start” button in the top right corner of the execution dialog.



# Test Case: Source Code I

Operation Body:

```
06 declare
07   mins : Integer := 0;
08   secs : Integer := 0;
09 begin
10   -- Check that initially the time is 0:0
11   mins := StopwatchPkg.StopWatch.getMin(this.itsStopWatch.all);
12   secs := StopwatchPkg.StopWatch.getSec(this.itsStopWatch.all);
13   TestConductor.ASSERT_NAME("Check initial time",
14     (mins=0) and (secs=0));
15
16   -- now set time to 03:21 and check that setting of time indeed
17   -- sets the correct time
18   StopwatchPkg.StopWatch.setTime(this.itsStopWatch.all,3,21);
19   mins := StopwatchPkg.StopWatch.getMin(this.itsStopWatch.all);
20   secs := StopwatchPkg.StopWatch.getSec(this.itsStopWatch.all);
21   TestConductor.ASSERT_NAME("Check if time setting is correct",
22     (mins=3) and (secs=21));
23 end;
24
```

To manually create a source code test case create a code test case and write the test code into the edit field under the implementation tab of the test case. TestConductor provides a set of functions like e.g. „TestConductor.ASSERT\_NAME“ that can be used to execute checks during test case execution. If the function “setTime” (line 08) of the stopwatch works as expected, the test case passes.



- 1 Select the test context „TCon\_StopWatch“ and choose from the context menu „Create Code TestCase“.

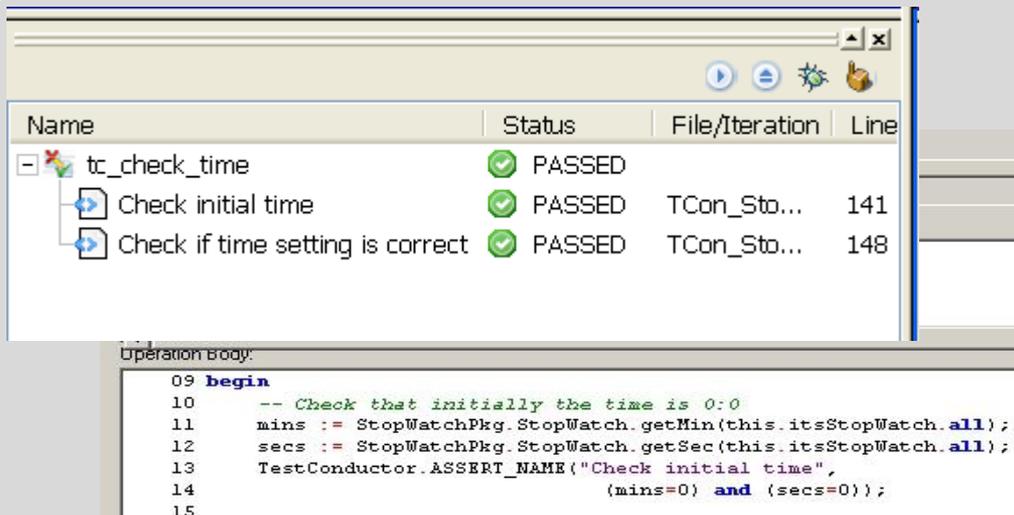


- 2 Rename the created test case to „tc\_check\_time“ and open the features dialog.



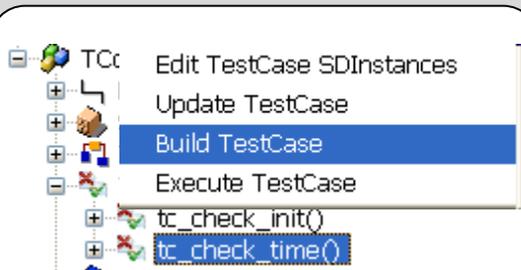
- 3 Replace the content of the edit field under the implementation tab of the test case with the content from the “tc\_check\_time” operation in the Tutorial package.

# Source Code Test Case: Execution

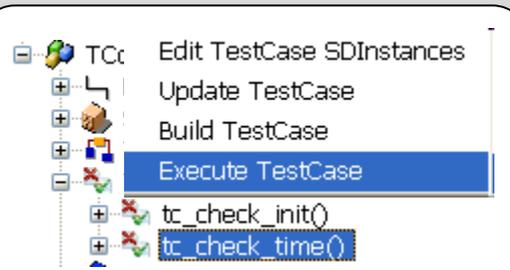


Execute the test case with Rhapsody TestConductor.

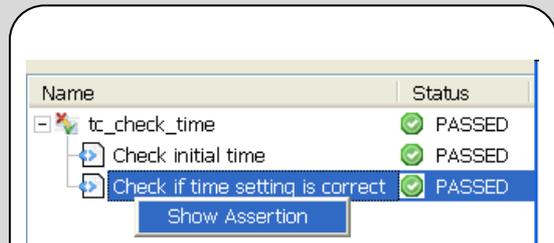
Both assertions evaluate to true and the test case passes. Double-clicking an evaluated assertion in the execution window highlights the assertion in the test model.

**1**

Select test case "tc\_check\_time" and then select "Build TestCase" from the context menu.

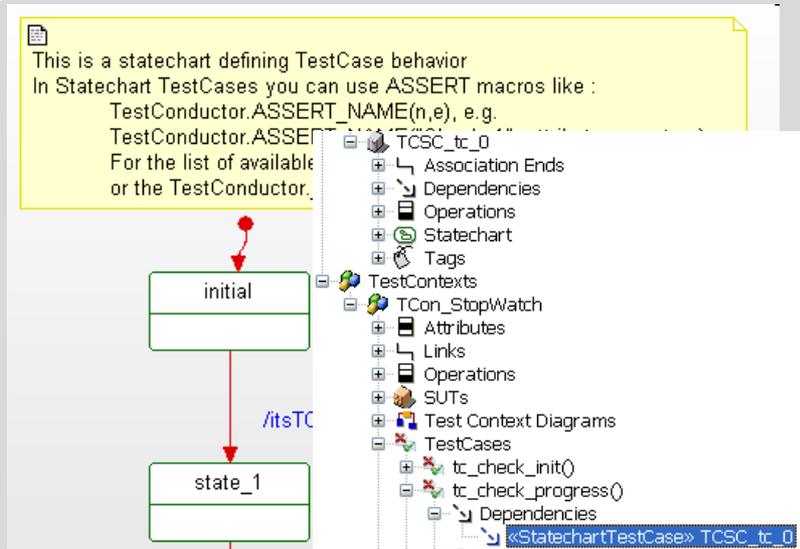
**2**

Select test case "tc\_check\_time" and select "Execute TestCase" from the context menu.

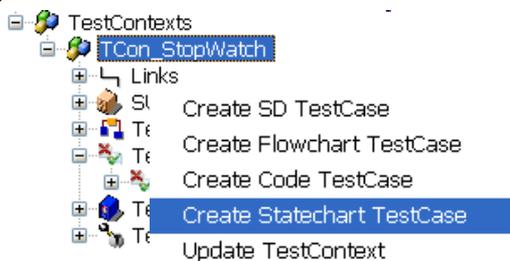
**3**

In the execution window, select the assertion and double-click "Show Assertion" in order to highlight the assertion in the model.

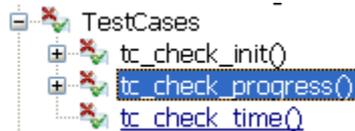
# Test Case: Statecharts I



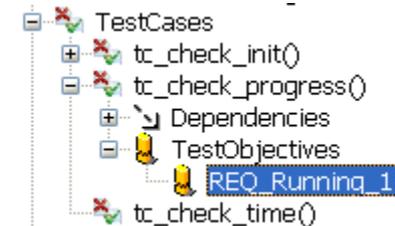
To manually create a statechart test case we have to define a test scenario which is represented as a statechart and link it to a test case. Technically, the test case has a dependency to a TestComponent that contains the statechart. TestConductor simplifies this process with a single command.



**1** Select the test context „TCon\_StopWatch“ and select “Create Statechart TestCase”.

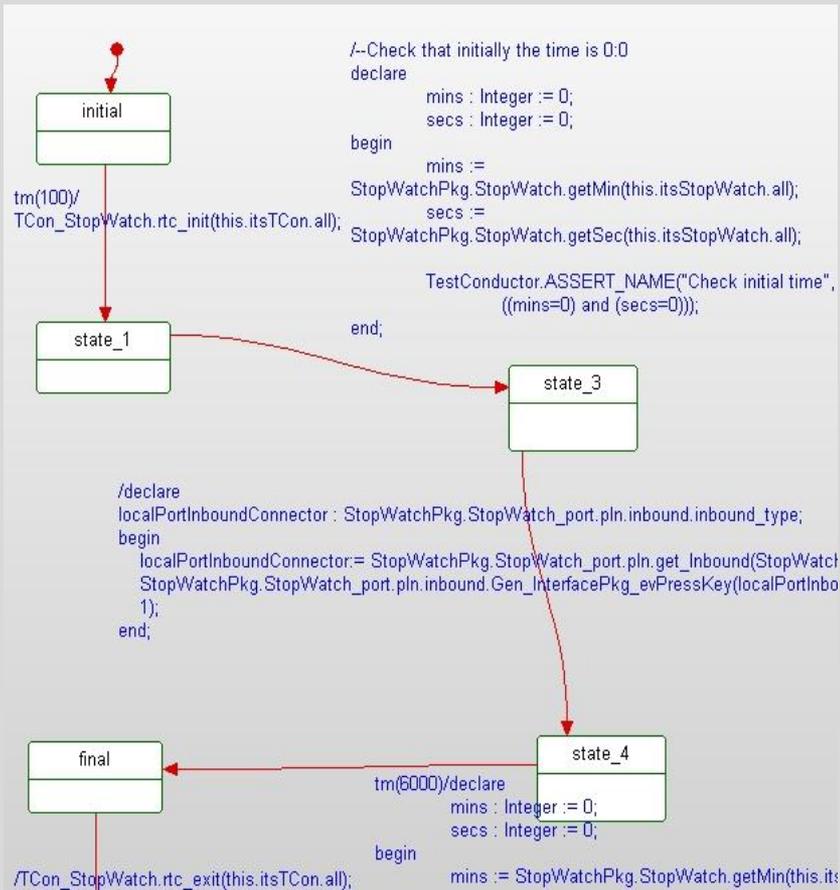


**2** Rename the test case to “tc\_check\_progress”

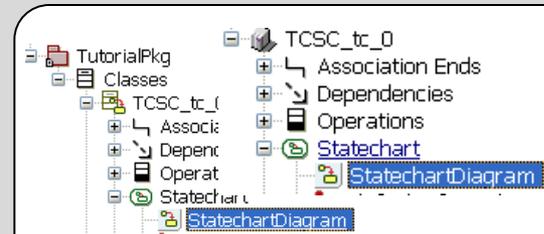


**3** Add a test objective (using “Add New -> TestingProfile ->TestObjective”) to requirement REQ\_Running\_1

# Test Case: Statecharts II



Vitalize the statechart in order to execute it with TestConductor. The statechart test case first checks that initially the stopwatch's time is indeed 0:0. After starting the stopwatch, the statechart test case waits 6 seconds, and then checks that indeed more than 3 seconds have been counted by the stopwatch during that period. To execute the checks the statechart test case uses the Rhapsody TestConductor function „TestConductor.ASSERT\_NAME()“. This function was already used for the code test case in order to perform code based checks. If both checks are passed, the complete test case is passed.



1

Replace the content of the test component statechart associated with this test case with the statechart of the Tutorial package.

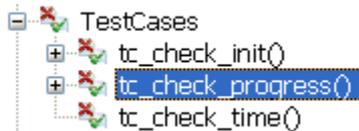
# Statechart Test Execution

The screenshot shows the TestConductor interface. At the top, a table lists test case results:

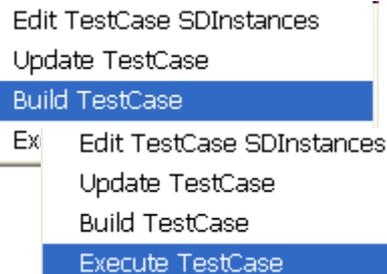
Name	Status	File/Iteration	Line
tc_check_progress	PASSED		
Check initial time	PASSED	TCSC_tc_...	408
Check elapsed time	PASSED	TCSC_tc_...	709

Below the table, a statechart diagram shows a transition from state\_1 to state\_3. A red arrow points from the 'Check elapsed time' entry in the table to the state\_1 box in the diagram. A blue arrow points from state\_1 to state\_3. A code snippet is visible: `TestConductor.ASSERT_NAME("Ch ((mins=0) and (secs=0));`

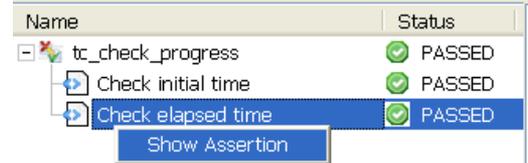
Execute the test case with Rhapsody TestConductor. Both assertions evaluate to true and the test case passes.

**1**

Select the test case „tc\_check\_progress“ ...

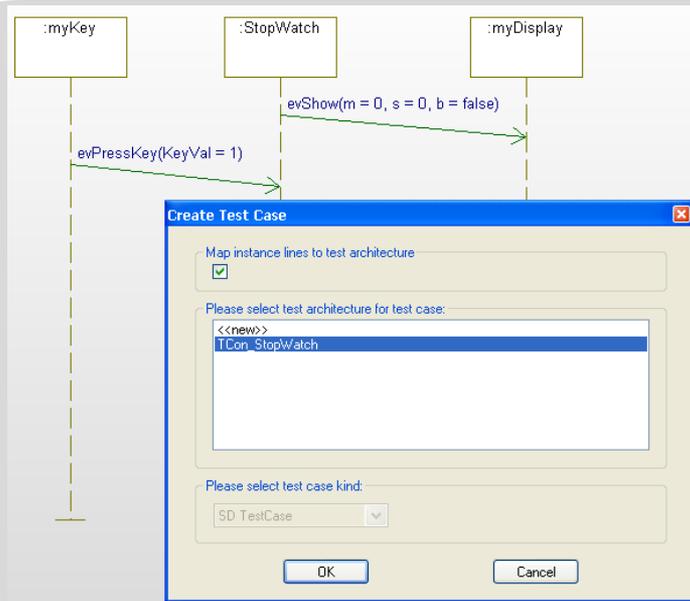
**2**

... and choose from context menu the items „Build TestCase“ and „Execute TestCase“.

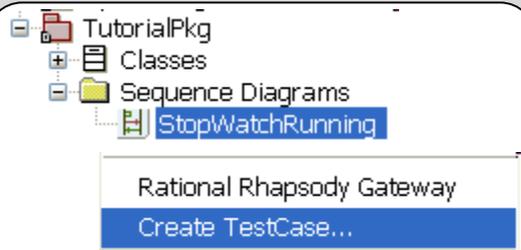
**3**

In the execution window, select the assertion and double-click “Show Assertion” in order to highlight the assertion in the model.

# Create Test Cases Using Test Case Wizard - SDs



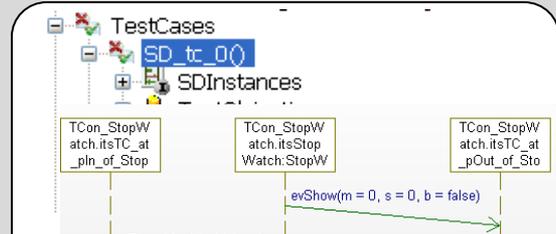
To create a test case based on existing sequence diagrams, operations or requirements, you can use the TestConductor test case wizard. For an existing sequence diagram, the test case wizard creates an analogue test case with the same message structure as the original sequence diagram. For an operation, the test case wizard creates a test case that tests the chosen operation, for a requirement the test case wizard creates a test case with the chosen requirement as the test objective.



- 1 Select the sequence diagram "StopWatchRunning" in the tutorial package and select "Create TestCase..."

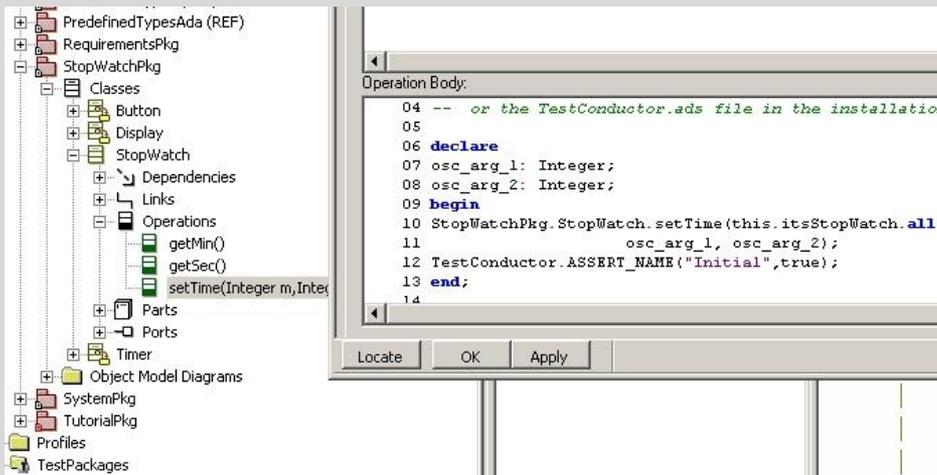


- 2 In the test case wizard dialog, the test context "TCon\_StopWatch" is already highlighted. Press OK to proceed.



- 3 As a result, a new test case "SD\_tc\_0" has been created which is based on a new test scenario containing the same messages as the original SD, but life lines adapted to the test context structure.

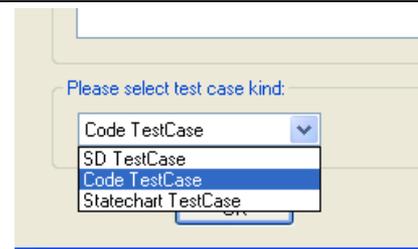
# Create Test Cases Using Test Case Wizard - Operations



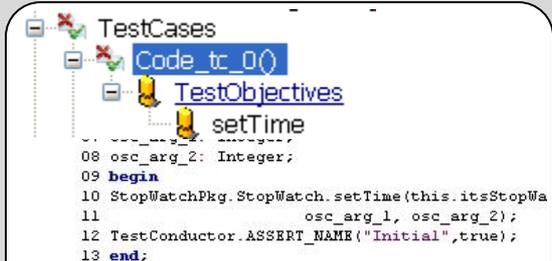
The test case wizard can also be used to **test operations** that are defined in the model. The wizard allows to create three different kinds of test cases: sequence diagram test cases, statechart test cases or code test cases. Independent of the chosen kind of test case, the created test case calls the selected operation. Additionally, the test case already contains a check that can be refined by the user in order to check the out values of the operation.



- 1 Select operation "setTime" of class StopWatch in the browser and select "Create TestCase..."



- 2 In the test case wizard dialog, select "Code TestCase" as test case kind and press OK.



- 3 As a result, a new code test case has been created that contains a call to operation "setTime" and also a dummy assertion that can be refined.

# Debugging Test Cases

The screenshot shows the TestConductor interface. At the top, a table lists test cases:

Name	Status	File	Line/Progress
SD_tc_0	EXECUTING		
SD_tc_0	ACTIVE	1	25% (2/8)

Below the table is a statechart diagram. It shows a sequence of events: `tm(500) at ROOT.Running.prestate`, `evShow(m = 0, s = 0, b = FALSE)`, `ShowTime(m = 0, s = 0, b = FALSE)`, and `evShow(m = 0, s = 0, b = FALSE)`. Three watch expressions are shown above the statechart: `TCon_StopW atch.itsTC_at _pln_of_Stop`, `TCon_StopW atch.itsTC_at Watch:StopW`, and `TCon_StopW atch.itsTC_at _pOut_of_Sto`. A toolbar with various animation controls is visible on the left.

Debugging failed test cases can also be done with TestConductor. When a test case fails, one can turn on debug execution mode in TestConductor's execution window. After switching on debug mode, when executing the test case one can step through it by using the "Go Step", "Go Idle", etc. buttons of Rhapsody's animation toolbar. Additionally, when stepping through the test case, one can use Rhapsody's animation features to inspect animated statecharts, animated SDs, etc. in order to find the reason why the test case fails.

The screenshot shows a tree view of test cases. The "SD\_tc\_0()" test case is selected, and a context menu is open with the following options:

- Edit TestCase SDInstances
- Update TestCase
- Build TestCase
- Execute TestCase

- 1 Select test case "SD\_tc\_0" and select "Execute TestCase".

The screenshot shows the TestConductor interface with a table of test cases:

Name	Status	File	Line/Progress
SD_tc_0	FAILED		
SD_tc_0	FAILED	1	50% (4/8)

The "Start" button in the toolbar is circled in red.

- 2 After the test case has failed, turn on debug execution mode by clicking the debug button in the execution dialog.

The screenshot shows the TestConductor interface with a table of test cases:

Name	Status	File	Line/Progress
SD_tc_0	EXECUTING		
SD_tc_0	ACTIVE	1	25% (2/8)

The "Start" button in the toolbar is highlighted.

- 3 Execute the test case again by pressing the "Start" button in the execution dialog. Now you can step through the test case by using Rhapsody's animation toolbar.

# Executing Multiple Test Cases

The screenshot shows the TestConductor execution window. The tree view on the left shows the following structure:

- TCon\_StopWatch (FAILED)
  - Code\_tc\_0 (PASSED)
  - Initial (PASSED) - TCon\_Stop... 132
  - SD\_tc\_0 (FAILED)
    - SD\_tc\_0 (FAILED) - 1, 50% (4/8)
  - tc\_check\_init (PASSED)
  - SD\_tc\_0 (PASSED) - 1, 100% (2/2)
  - tc\_check\_progress (PASSED)
  - Check initial time (PASSED)
  - Check elapsed time (PASSED)
  - tc\_check\_time (PASSED)
  - Check initial time (PASSED)
  - Check if time setti... (PASSED)

The summary table below the tree view is:

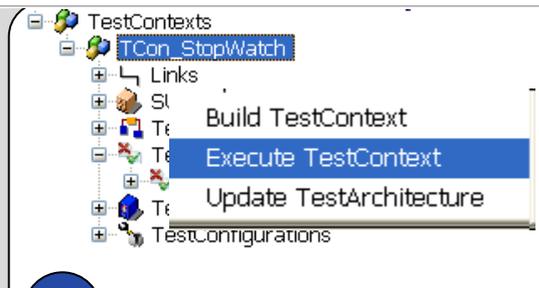
Test Context: TCon_StopWatch	Summary: FAILED
Code_tc_0	PASSED
SD_tc_0	FAILED
tc_check_init	PASSED
tc_check_progress	PASSED
tc_check_time	PASSED

Executing multiple test cases can be done by executing a complete test context or a complete test package. When a test context or a test package is executed, all test cases within the context or test package are executed. After all test cases have been executed, TestConductor computes an overall test result for the test context or the test package.



1

Select the test context "TCon\_StopWatch" and select "Update TestContext". After that, select "Build TestContext".



2

Select the test context again and press "Execute TestContext". All test cases will be executed one after the other.

The screenshot shows the TestConductor execution window. The tree view on the left shows the following structure:

- TCon\_StopWatch (FAILED)
  - Code\_tc\_0 (PASSED)
  - Initial (PASSED) - TC
  - SD\_tc\_0 (FAILED)
    - SD\_tc\_0 (FAILED) - 1

3

The results are shown in the execution window. As always, "Show as SD" resp. "Show assertion" can be used to show the reasons of failed test cases.

# Assessing Test Case Requirement Coverage I

The screenshot shows a table with the following data:

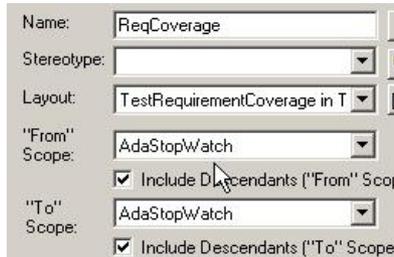
To: Requirement	Scope: JavaStopWatch	REQ_Init	REQ_Running_2	REQ_Stopping	REQ_Running_1	REQ_SetTime
tc_check_init	REQ_Init					
tc_check_time						
tc_check_progress					REQ_Running_1	
SD_tc_0						
Code_tc_0						

Which requirements are covered by my test cases? This important question can be answered either by using a test case requirements matrix or by generating a requirements coverage test report. A test case requirements matrix shows the relationship between test cases and requirements in a matrix view. A requirements coverage test report shows the same information, but presented as a textual report. It can be generated by ReporterPlus using a predefined template.



1

Select the test package "TPkg\_StopWatch" and select "Add New -> TestingProfile -> TestRequirementMatrix".



2

Open the features dialog of the matrix, rename it to "ReqCoverage", and set the "from" scope and the "to" scope to the complete model "AdaStopWatch".

The screenshot shows a table with the following data:

To: Requirement	Scope: AdaStopWatch	REQ_Init	REQ_Running_1	REQ_SetTime
tc_check_init	REQ_Init			
tc_check_time				
tc_check_progress			REQ_Running_1	
SD_tc_0				
Code_tc_0				

3

When double clicking the matrix in the browser, the matrix view shows the relationship between the test cases and the requirements.

# Assessing Test Case Requirement Coverage II

Test Case Reports can be used as an alternative in order to figure out coverage of requirements with the test cases. With ReporterPlus a requirement coverage report can be generated in different formats like Word, Html, etc. The requirements coverage test report shows the same information as the requirements coverage matrix, but presented as a textual report. Besides the requirements coverage report, another predefined template (TestReport.tpl) can be used to generate a complete test report that contains all details about test cases, test architectures, etc.



The screenshot shows a web browser window titled 'C:\Test it\Rhapsody7.5.2\reporterplus\UserFiles\CoverageReport.htm'. The browser displays a 'Table of Contents' on the left and a table titled 'All Requirements' on the right. The table lists requirements and their coverage status.

Name	Specification	Covered by Test Case
REQ_Init	After starting the stopwatch, the stopwatch shall display 0 minutes and 0 seconds (0:0)	tc_check_init (Passed)
REQ_Running_1	After starting the stopwatch, the stopwatch shall count minutes and seconds.	tc_check_progress (Passed)
REQ_Running_2	After starting the stopwatch, the stopwatch shall count minutes and seconds. The colon between displayed minutes and seconds shall blink once in a 1 second time interval.	not covered
REQ_SetTime	The stopwatch shall provide a function "SetTime" that sets the current time.	not covered
REQ_Stopping	When running, pressing the key of the stopwatch shall stop it.	not covered

Check Model

ReporterPLUS

Report on model...

Report on all model elements...

Report on selected package...

1

From Rhapsody's tools menu, select "ReportPlus -> Report on all model elements".

ReporterPLUS Wizard : Select

What would you like to do?

Generate HTML Page

Generate Microsoft PowerPoint Presentation

Suchen in: Templates

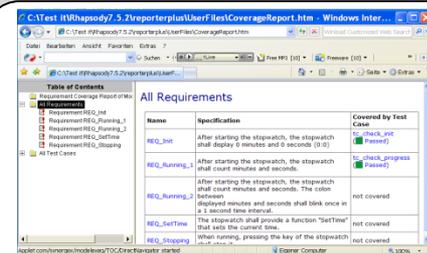
TestReport.tpl

TestRequirementCoverage.tpl

UseCaseDiagramsDetailedReport.tpl

2

As format, select Html. After that, select "TestRequirementCoverage.tpl" as template for the report to generate.



The screenshot shows a web browser window titled 'C:\Test it\Rhapsody7.5.2\reporterplus\UserFiles\CoverageReport.htm'. The browser displays a 'Table of Contents' on the left and a table titled 'All Requirements' on the right. The table lists requirements and their coverage status.

Name	Specification	Covered by Test Case
REQ_Init	After starting the stopwatch, the stopwatch shall display 0 minutes and 0 seconds (0:0)	tc_check_init (Passed)
REQ_Running_1	After starting the stopwatch, the stopwatch shall count minutes and seconds.	tc_check_progress (Passed)
REQ_Running_2	After starting the stopwatch, the stopwatch shall count minutes and seconds. The colon between displayed minutes and seconds shall blink once in a 1 second time interval.	not covered
REQ_SetTime	The stopwatch shall provide a function "SetTime" that sets the current time.	not covered
REQ_Stopping	When running, pressing the key of the stopwatch shall stop it.	not covered

3

After generating the report, the report can be viewed with any browser that can display Html files.

# Assessing Test Case Model Coverage

Detailed Coverage Summary of Button (5/5)		
<b>Operations</b>		
covered	KeySend	
<b>EventReceptions</b>		
covered	evPressKey	
<b>Statechart: StatechartOfButton</b>		
covered	ROOT.Running	State
covered	0	Transition
covered	1	Transition

Detailed Coverage Summary of Display (5/5)		
<b>Operations</b>		
covered	ShowTime	
<b>EventReceptions</b>		
covered	evShow	
<b>Statechart: StatechartOfDisplay</b>		
covered	ROOT.running	State
covered	0	Transition
covered	1	Transition

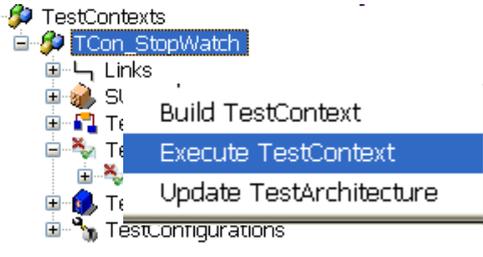
Besides coverage of the requirements, an important orthogonal information is which parts of the model are executed by the test cases, i.e, what is the achieved **Model Coverage** when executing the test cases. TestConductor can compute this information during test case execution. When model coverage computation is turned on, after test case execution TestConductor adds a model coverage report to the test cases, test contexts etc. that shows the achieved model coverage.

1



Open the features dialog of the test package "TPkg\_StopWatch" and turn on property "TestConductor.TestCase.ComputeCoverage".

2



Execute the test context "TCon\_StopWatch".

3

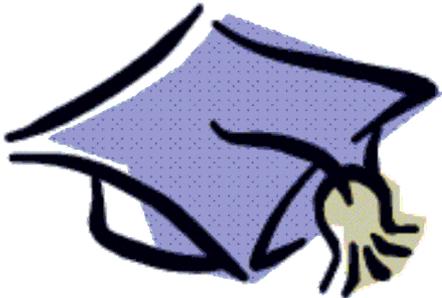


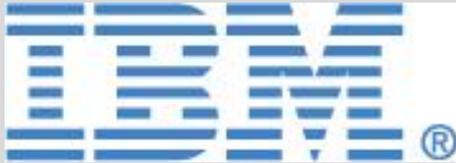
After execution has finished, coverage reports can be found both for individual test cases as well as a cumulative coverage report for the test context.

# Conclusion

## The high-grade automation in the Rhapsody Testing Environment with TestConductor

- /// generates complete, immediately executable test architectures in shortest time with a few mouse clicks.
- /// makes it for the first time possible to implement cyclically quality assurance measures in early phases of the development.
- /// increases substantially the planning reliability for projects, because design errors and subsequent errors will be recognized very early.
- /// makes statements about the coverage rates for both the model elements and model code. Developers can easily and fast analyze reasons for not covered elements.
- /// highly automates the testing process and **can save test development time** compared to traditional approaches.





For further information, especially technical news, visit our internet information portal or contact one of our worldwide sale agencies.

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.