Rational. Rhapsody

IBM

# IBM® Rational® Rhapsody® Rhapsody TestConductor Add On

# IBM Rational Rhapsody TestConductor Add On Reference Workflow Guide

**Version 1.12**

IBM®

**License Agreement**



**Trademarks**

# Table of Contents

# 1 Purpose

This document describes a reference workflow for testing activities in a model based development process using IBM Rational Rhapsody and IBM Rational Rhapsody TestConductor Add On. It complements the IBM Rational Rhapsody Reference Workflow document [1] that focuses on the model based development with IBM Rational Rhapsody in safety-related projects. The subsequent sections provide further information and describe variations of the IBM Rational Rhapsody Reference Workflow when applied in practice, focusing on testing methods as provided by IBM Rational Rhapsody TestConductor Add On from BTC Embedded Systems.

# 2 Introduction

During translation of textual requirements to final object code, several verification steps need to be done in order to ensure that the translation is performed correctly. In a development process following the V-model, such verification steps are commonly done manually by performing tedious, time-consuming, and error-prone static tests and reviews to compare the input of a step with its respective output.

Model-based development and model-based testing enable the automation of many of these manual tasks. Because formal models have clearly defined operational semantics, they can be simulated and tested for functional correctness very early. Therefore it is possible to perform a requirements-based functional test of the model that ensures the model correctly implements the given requirements. Furthermore, code generators can be used to convert the model to compilable source code such as C-code. Instead of manually reviewing the translation step by comparing code behavior to model behavior, automated back-to-back testing can be used to conduct the comparison. By using the same test cases and observing test results, it is possible to establish an equivalence check of the behavior on the model and code levels. To complement equivalence checking, appropriate model and code-coverage metrics shall be used to demonstrate completeness.

# 3 IBM Rational Rhapsody Reference Workflow Overview and Variations

The IBM Rational Rhapsody Reference Workflow [1] describes an approach for model-based development including automatic code generation and model-based testing.

Figure 1 shows the major activities of this reference workflow. The upper part of the workflow describes activities that are performed without IBM Rational Rhapsody TestConductor Add On. The lower part of the workflow describes activities that are performed with IBM Rational Rhapsody TestConductor Add On. The approach addresses design and implementation together with appropriate test and verification:

- Textual requirements guide the development of a formal UML/SysML model, which then is translated to code using code generation. Both refinement steps are accompanied with appropriate guidelines and checks.

- The refinement step from textual requirements to a model ready for code generation is verified by performing systematic requirements based testing on the model level leveraging from model simulation. The generated object code is verified by executing the same set of test cases as for the verification of the model, and performing an equivalence check of the test results (back-to-back testing).
- Test execution on model and code comes along with structural coverage measurement to assess the completeness of the tests and to avoid unintended functionality, for instance, to identify implementation details introduced by the code generator
- Requirements coverage is measured during execution of the test cases.

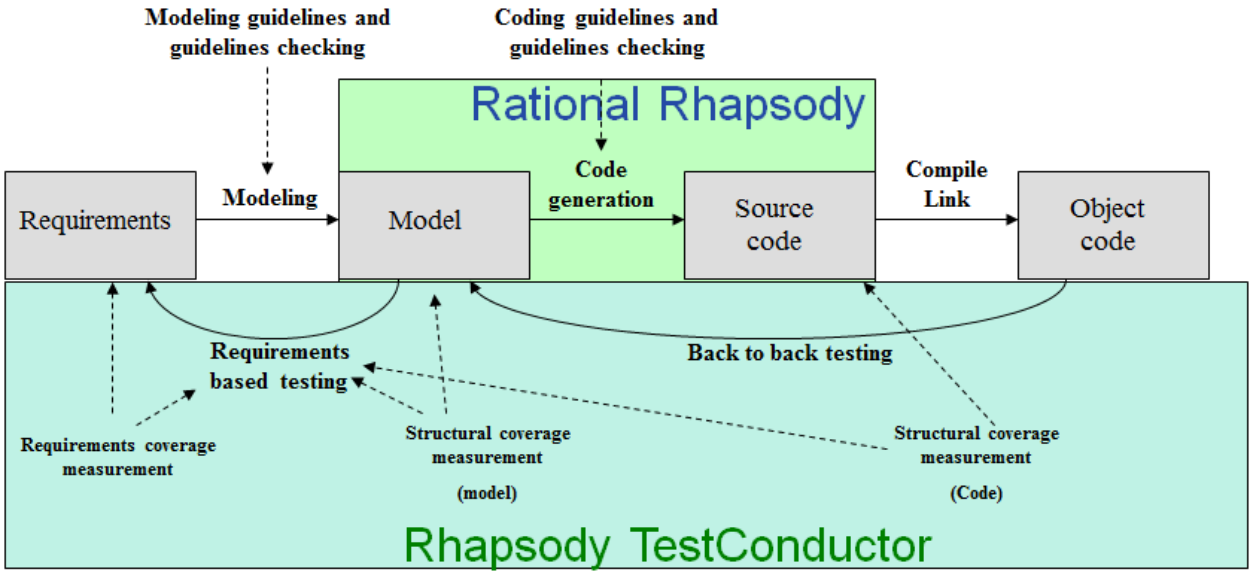



**Figure 1: Activities of the IBM Rational Rhapsody Reference Workflow**

Requirements are translated into an executable model with appropriate modeling guidelines. Model based tests are added in order to ensure that the model indeed correctly captures the requirements. Coverage metrics (requirements coverage and model coverage) guarantees the completeness of the model based test suite. Code generation, either automatic or manual or a mixture of both, may be used to generate an implementation from the model. Back-to-back testing between model and code constitute the key element for code verification. Code coverage metrics are used in order to ensure completeness of the test suite with regard to the predefined code coverage criteria.
The key element of this workflow is the verification of the translation steps from the requirements into the model and from the model to the generated code. These verification steps guarantee that the translation steps are performed correctly. In this document, we focus on the verification activities depicted in the lower part of Figure 1, i.e., the verification activities that can be performed with IBM Rational Rhapsody TestConductor Add On. The verification activities depicted in the upper part of Figure 1 are described in detail in [1]. The coverage measurement activities complement the verification steps in order to ensure completeness of verification.

## 3.1 Model Evolution

Figure 1 roughly sketches the main steps for translating requirements to code suitable for a target-architecture.

In practice, the process is of a more incremental nature. With regard to modeling there is usually an evolution of the model from an early specification model via a design model to an implementation model containing all relevant information for the subsequent code generation. These refinement steps are outlined in Figure 2. Note that such refinements can be functionally motivated or implementation related.

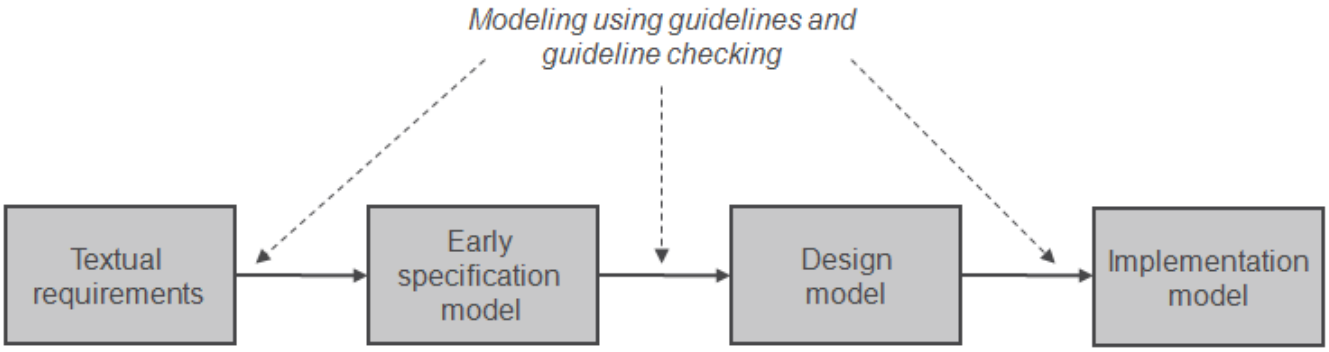



**Figure 2: Evolution of textual requirements into an implementation model ready for production code generation**

Textual requirements are translated to an executable model that helps to verify the correctness and completeness of later modeling stages with respect to the requirements, as well as to improve understanding of the requirements. This first model is known as an early specification model. Such an early specification model grants traceability between requirements to derived model elements and vice versa, as required by the safety standards ISO 26262, IEC 61508, IEC 62304, and EN 50128. A design model is obtained from that specification model by adding software architectural details such as structural hierarchy of components and their interactions. By enhancing the design model with implementation elements such as data types or fixed-point approximations - in case a fixed-point target is used - one finally obtains an implementation model containing all information necessary for subsequent production code generation.

All modeling steps should be conducted in accordance to suitable modeling guidelines which can be checked and established using appropriate tools.

## 3.2 Testing Considerations

The model-based testing process, i.e. the testing process that accompanies the model-based development process, greatly benefits from the ability to execute the model at its different evolutionary stages.

### 3.2.1 Model Verification by requirements based testing

Viewing the IBM Rational Rhapsody model based reference workflow from a test and verification perspective, the first significant activity is the verification of the model by

demonstrating that the model is correct, meets its requirements and does not contain unintended functionality.

Model verification is mainly done by performing functional, requirement-based tests on the executable model. Test cases that cover all functional requirements have to be derived and executed. In order to ensure completeness of the model based test suite, requirements based coverage metrics are used that show the coverage of the requirements by test cases. Additionally, in order to ensure that the model does not contain any additional unintended functionality, model coverage measurement is used to verify the completeness of the test suite with regard to the model.

By all the above verification steps, one obtains a “Golden Model” which is used as a reference in later testing steps.

### 3.2.2 Back-to-Back Testing

From early specification models or executable specifications, subsequent refinement steps (see 3.1 Model Evolution) then shall preserve the semantics of the model. In practice, this can be assured by re-executing the requirement based test cases on the evolving model. The same approach can be used to automate the verification of the generated code, provided that the model based test cases can be executed on the generated code. By comparing the test execution results of the code with the test execution results of the model, one verifies if the code behaves equivalent to the model. In practice, one typically distinguishes between 3 different execution levels of a model, called MIL (model in the loop), SIL (software in the loop), and PIL (processor in the loop). The model based test suite shall reveal equivalent results on all these levels (v. Figure 3).

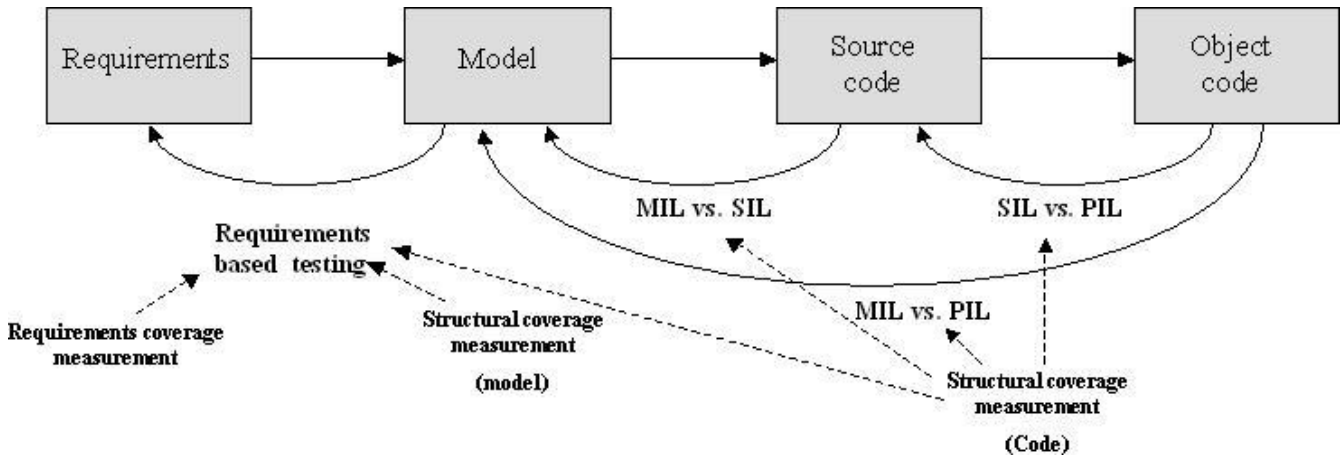



**Figure 3: Back-to-back testing on different execution levels (MIL to SIL, SIL to PIL and MIL to PIL)**

The above mentioned test suite derived from the requirements can be executed on all levels MIL, SIL and PIL, and the results can be compared. Even if all test results are successful, it is important to note that the applied structural coverage metrics can reveal that the test suite is not complete with regard to the measured coverage criteria. In this case it is needed to extend the test suite with additional test cases in order to achieve the desired level of structural coverage, or to remove unintended functionality.

### 3.2.3 Coverage Measurement

For measuring code coverage, IBM Rational Rhapsody TestConductor Add On instruments the code of the SUT. After instrumentation, the test cases are executed on the instrumented code in order to compute the code coverage achieved by the test cases. In order to make sure that this instrumentation of the code does not affect the test results, the test cases can be repeated without instrumentation. This approach is mentioned in Note 3 of ISO 26262-6 (9.4.5) [2].

### 3.2.4 Unit Testing

Like ordinary testing processes, model-based testing approaches can take advantage from design hierarchy for performance and efficiency purposes. For this, IBM Rational Rhapsody TestConductor Add On supports testing of isolated SW components, often called SW units. A system under test (SUT) can be either a leaf SW component without further subcomponents, or a hierarchical SW component that contains further subcomponents. Unit testing means to test SW subcomponents isolated from their integration, allowing to

- stimulate the SUT interface and verify requirements directly on the components they belong to, and
- perform back-to-back testing for this SUT with respect to different abstraction levels (model, code, object-code), and
- Accomplish structural coverage goals for an entire system by hierarchical accumulation of coverage achieved for its subcomponents.

Note that unit testing strategies are more powerful than monolithic ones, as units of a design are tested independent from their integration context. For instance, certain portions of code might be traversable only by stimulating a subsystem's interface, while stimulating the top-level interface cannot be sufficient to achieve this goal. Additionally, complexity of the SUT in the unit testing approach is lower and hence makes it easier to verify correctness and to debug errors. The basics of model and code verification described in the reference workflow remain unchanged, i.e., the workflow can be applied on basic units as well as on more complex units that have internal subunits. This is depicted in Figure 4.

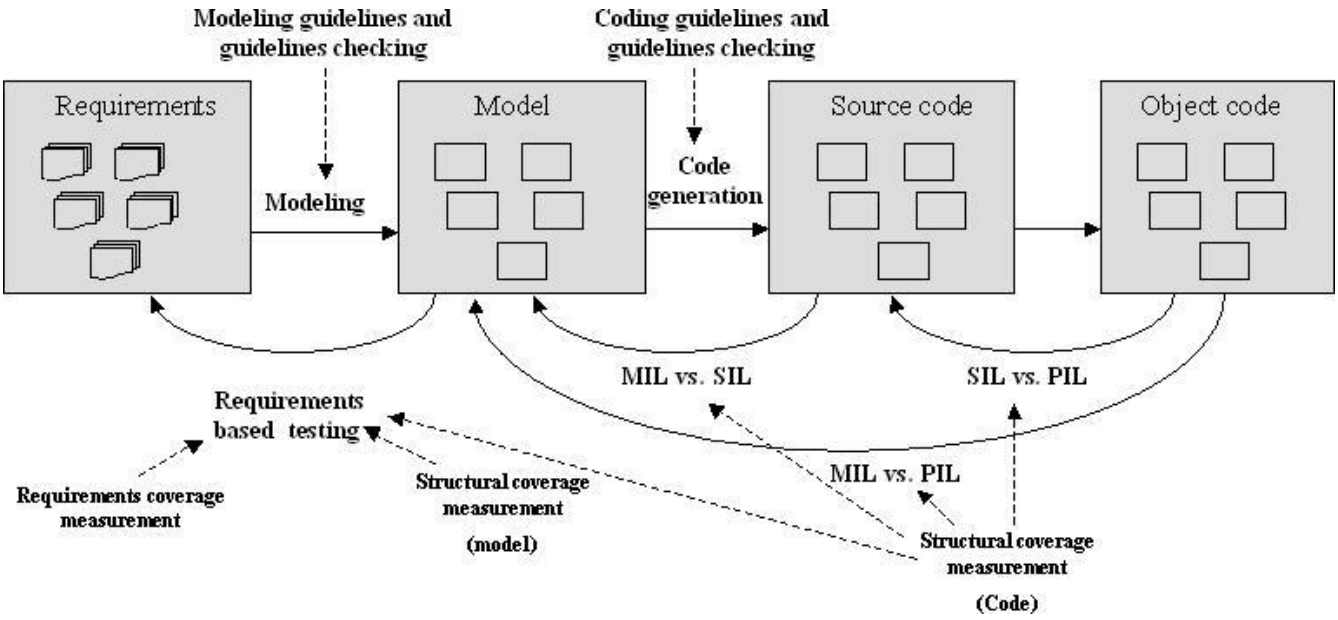



**Figure 4: Elements of the IBM Rational Rhapsody Reference Workflow considering hierarchical and modular partitioning and modular development**

For example, back-to-back tests between model and code using IBM Rational Rhapsody TestConductor Add On can be performed to verify the correct implementation of software units or modules, as well as part of the software integration testing for complete models and the corresponding code.

## 3.3 Variation of Reference Workflow without Explicit Model Verification

Beside the workflow depicted in Figure 1, in practice sometimes the variation of this workflow depicted in Figure 5 is applied. The difference between the workflow depicted in Figure 1 and Figure 5 is that in this variation, there is no explicit verification of the model regarding the given requirements.

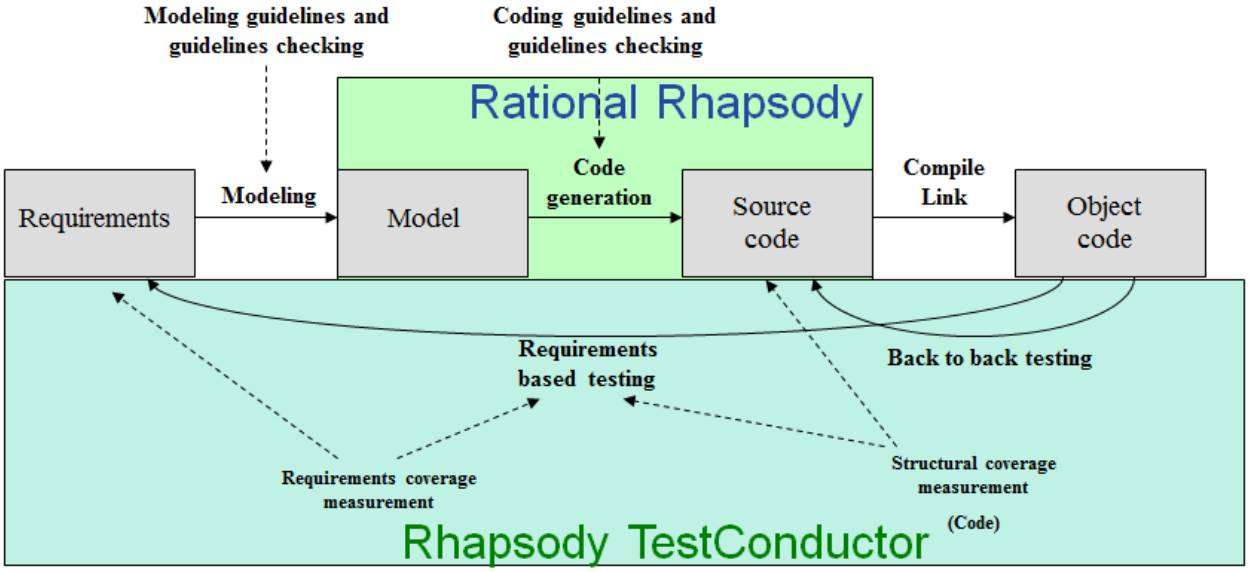



**Figure 5: Variantion of the reference workflow without explicit model verification**

Without explicit model verification, the workflow contains the following steps:

- Creation of a model based on the given requirements. The model is created with respect to modeling guidelines. However, the model is not simulated or dynamically tested. The reason for not performing simulation or dynamic testing of the model can be that the model e.g. contain some target hardware specific parts (e.g.some libraries only existing for the target hardware) that cannot be simulated at all on the model level.

- The model is translated into source code by applying an automatic code generator or manual code development or a mixture of both.

- The source code is compiled for SIL and PIL.

- Test Cases are created on basis of the requirements with IBM Rational Rhapsody TestConductor Add On. These test cases are executed by IBM Rational Rhapsody TestConductor Add On for SIL and PIL. Back to back testing can be performed regarding SIL and PIL.

- IBM Rational Rhapsody TestConductor Add On measures the requirements coverage and the code coverage.

Although this variation of the reference workflow does not contain an explicit verification of the model, the correctness of the model is verified indirectly by verifying the output of the

automatic code generator on the code level. The drawback of such an indirect verification on the code level is the fact that in case of errors the error analysis must be performed on the code level and cannot be done on the model level directly. After the source of the error is identified on the code level, one needs to identify appropriate changes on the model level that will correct the problem on the code level. Reverse-engineering such a problem resolution from the code level to the model level is sometimes time consuming and far from trivial. Nevertheless, by keeping the model in sync with the code, an indirect verification of the model, is achieved by performing a complete requirements based test on the code. The code coverage metrics provided by IBM Rational Rhapsody TestConductor Add On give evidence that the generated code does not contain untested code.

# 4 Guided Tour through the IBM Rational Rhapsody TestConductor Add On Reference Workflow

In this section, we describe by means of a running sample how the workflow described in Figure 1 can be instantiated. The purpose of this section is to provide additional practical information that eases the adaption of the workflow described in Figure 1.

## 4.1 The Stopwatch Project Requirements

As a running sample we want to sketch the development and testing of a stopwatch model. For the stopwatch there exist a couple of requirements. The requirements are kept in a word document (cf. Figure 6).

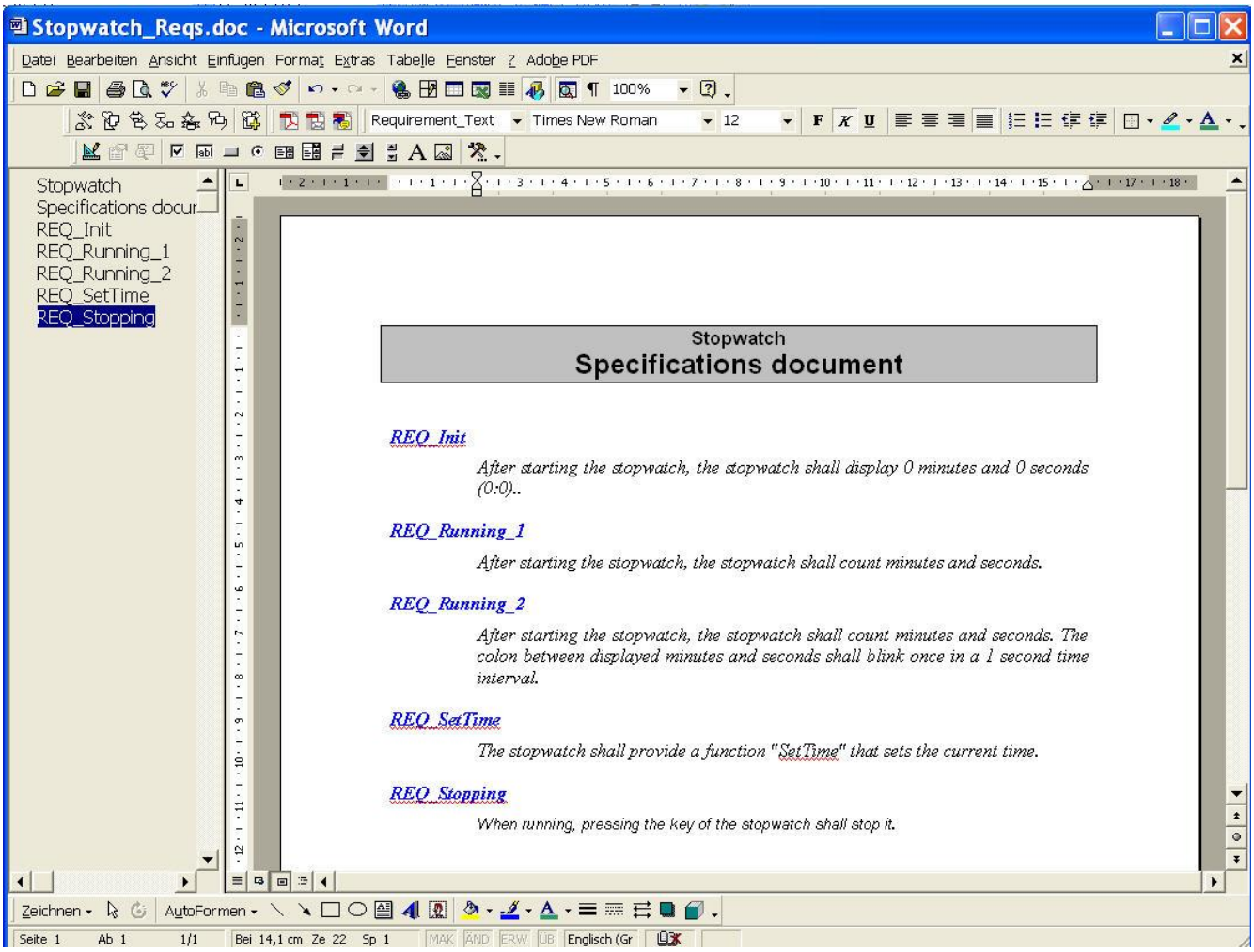



**Figure 6: Textual requirements for the stopwatch listed in a word document.**

As an example, in this document requirement REQ_Init is listed that states:

„REQ_Init: After starting the stopwatch, the stopwatch shall display 0 minutes and 0 seconds (0:0)“.

## 4.2 The Stopwatch Project – Importing Requirements into the Model

Based on the requirements described in the previous section, one can start creating an initial IBM Rational Rhapsody model. At first, the initial model should contain just the requirements specified in the word document. There are different ways in order to make the requirements visible in the IBM Rational Rhapsody model. For instance, one can import requirements from a requirements management system like DOORS by using the DOORS import feature of IBM Rational Rhapsody. However, in this sample we simply manually create requirement elements in the model. After adding these requirement elements, all requirements are now contained in a separate Requirements package in the model (cf. Figure 7).

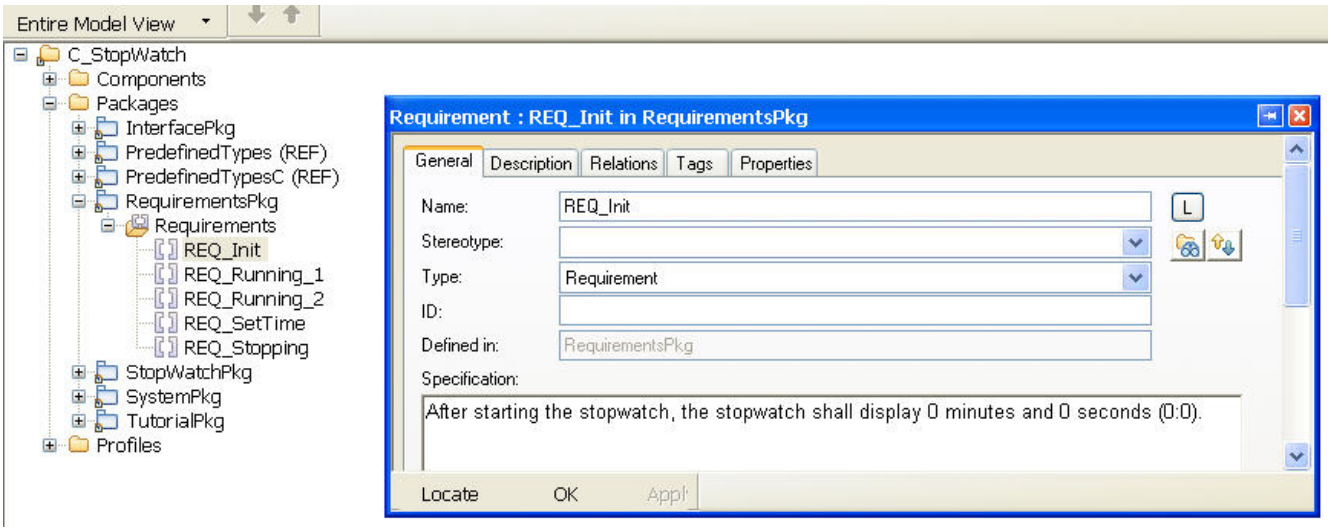



**Figure 7: All requirements from the word document are represented as requirements in the IBM Rational Rhapsody model. The textual specification is stored for each requirement.**

## 4.3 The Stopwatch Project – Design Model Development

Based on the initial model created in the previous section, the next step is to develop the functional design model by means of UML diagrams provided by IBM Rational Rhapsody. At this point, we do not go further into the details how to develop such models with IBM Rational Rhapsody but shortly summarize the final model.

The developed IBM Rational Rhapsody model basically contains 4 different packages. The package “RequirementsPkg” contains all requirements. Package “InterfacePkg” contains so-called “Interfaces” and “events”. The interface package is depicted in Figure 8. An interface is a collection of synchronous (operations) and asynchronous (events) messages that can be used in order to exchange information between system components, e.g. between classes. As an example, the interface “IDisplay” contains the events “evReset” and “evStartStop”.

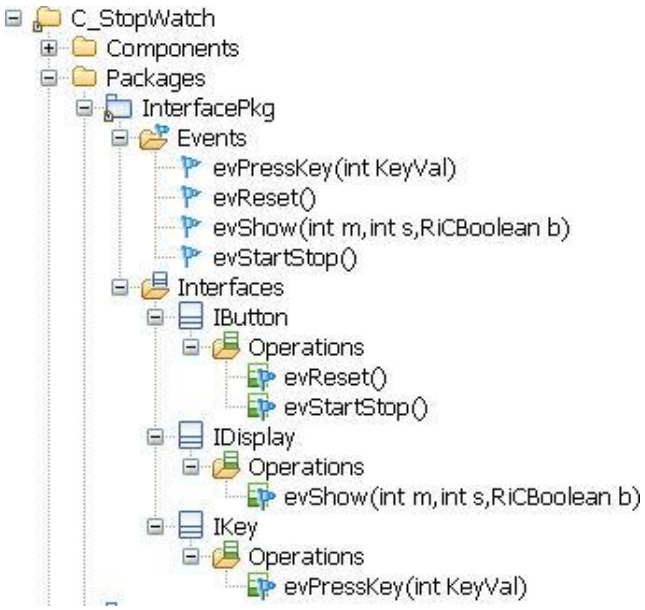



**Figure 8: Interfaces of the stopwatch model**

In the package „StopwatchPkg“ (cf. Figure 9) one can see the classes that implement the functional behavior of the model. The class “Stopwatch” provides the functionality of a stopwatch. The other classes “Button”, “Display” and “Timer” represent internal classes that are used inside the class “StopWatch” (cf. Figure 9, right side).

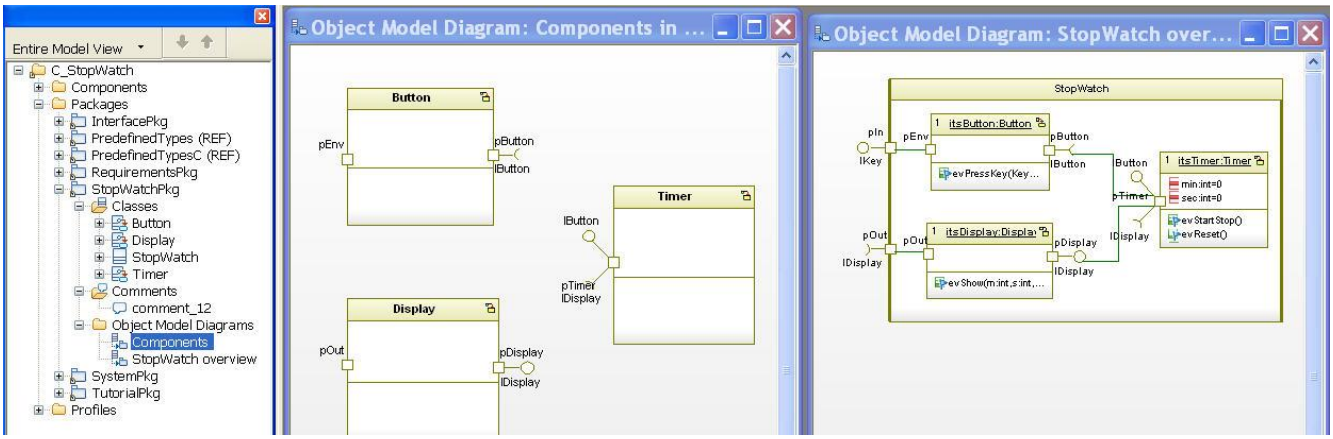



**Figure 9: Classes Button, Timer, Display, and StopWatch**

The behavior of a class can be defined by using operations and statecharts. As an example, the class “Timer” contains a statechart that defines the behavior of the class (cf. Figure 10)

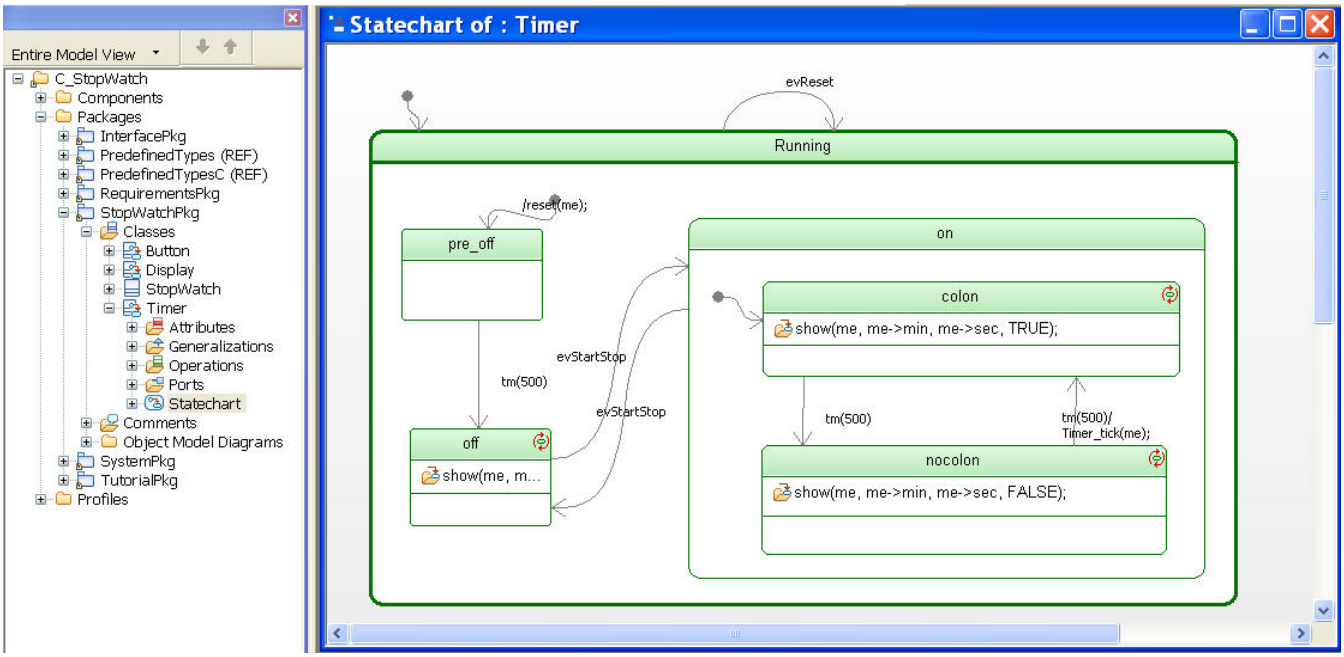



**Figure 10: Statechart of class Timer**

## 4.4 The Stopwatch Project – Design Model Simulation (Model in the loop, MiL)

After the model has been completed, as a next step one can interactively simulate the model (**M**odel **I**n the **L**oop simulation (MiL)) in order to verify that the functional behavior of the design model is as specified in the requirements. In order to simulate a complete model or parts of a model, one needs to define a so-called “component”. A component defines which parts of the model should be considered during Design Model simulation. Within a component, one can define different so-called configurations. A configuration provides several options, e.g. if simulation code or production code shall be generated for the model elements that are in scope of the component. The difference between production code and simulation code is that the production code can later be used in the final production environment. Contrary to that, simulation code contains many additional code parts that are only needed for simulation, animation, visualization and debugging purposes. Additionally, a configuration provides many simulation and code generation options that can be used in order to generate specific source code for e.g. specific compilers. In the stopwatch model there is one component with a simulation configuration defined (cf. Figure 11), where instrumentation mode is set to animation.

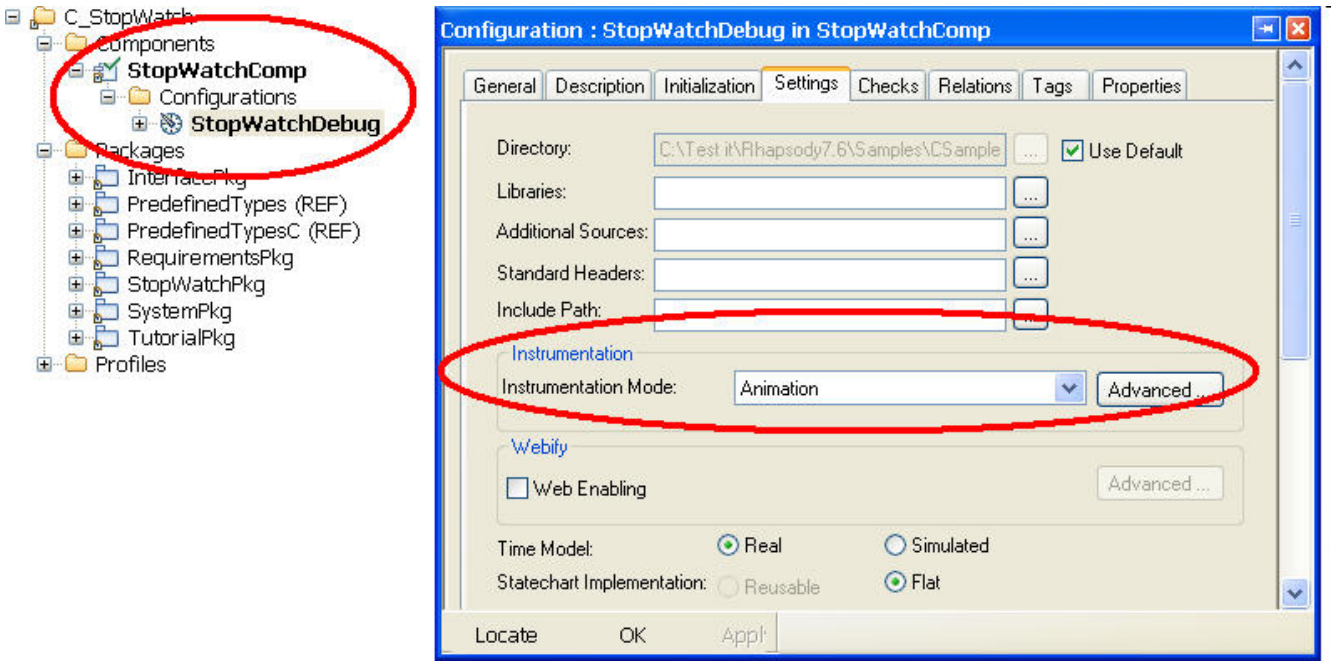



**Figure 11: IBM Rational Rhapsody component with simulation configuration. By setting the instrumentation mode to "Animation" the configurations becomes a simulation configuration. (MiL, Model in the loop).**

With this configuration one can now generate simulation code including model animation that can be compiled and executed. Executing simulation code means that the model simulation is started. During a model simulation, the model can be executed, controlled and observed. The so-called animation toolbar (cf. Figure 12) allows a step-by-step simulation of the model, where the steps can have different granularity. Alternatively, one can also simulate the model in real time. Additionally, during simulation one can stimulate the model by providing inputs to model objects. For instance, one can send events to specific model objects. The reactions of the model to the provided inputs can be observed by means of so-called animated diagrams. An animated diagram is a diagram that highlights the current state graphically (cf. Figure 12). Moreover, the model browser supports to inspect the values of object attributes during a simulation run (cf. Figure 12).

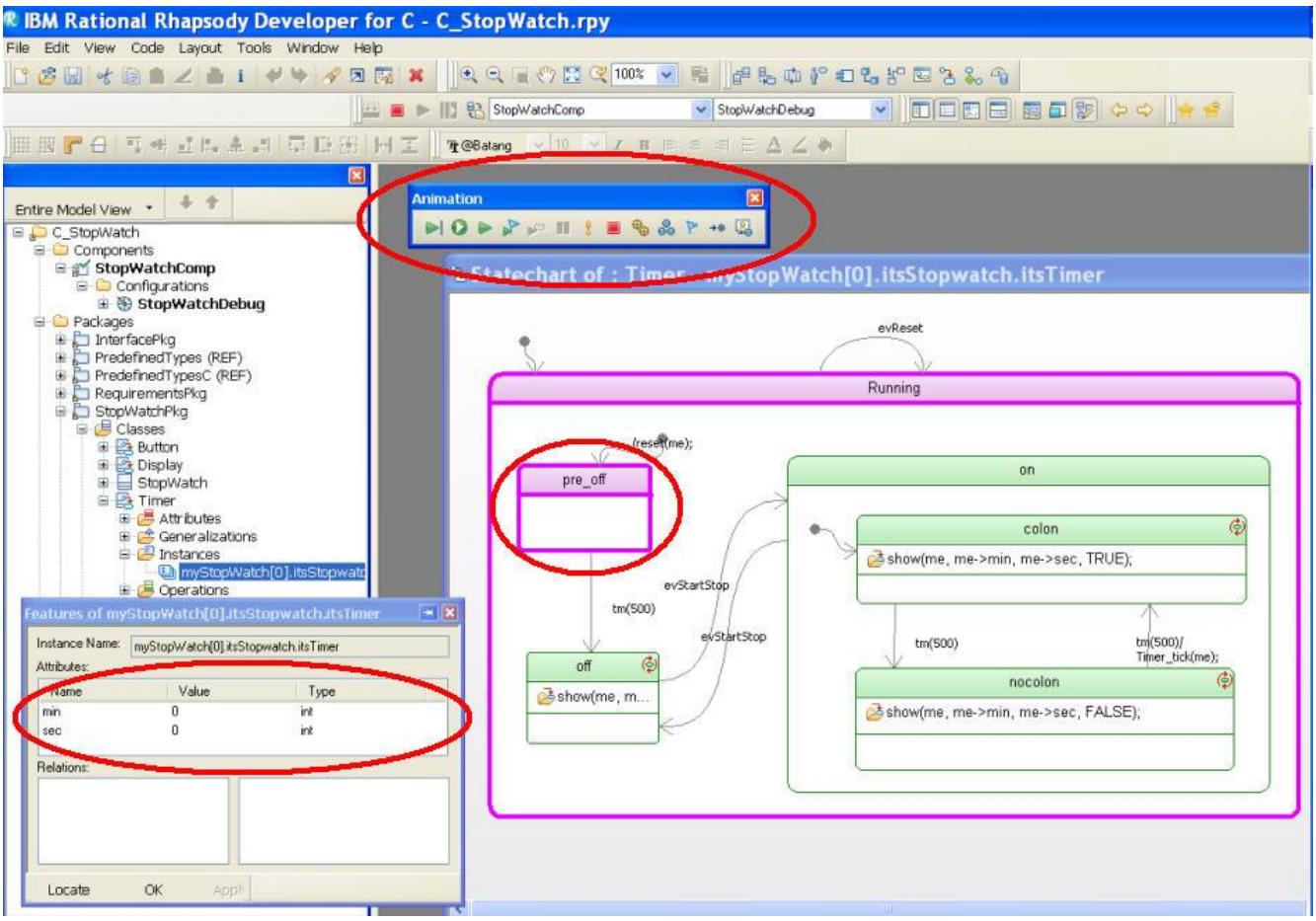



**Figure 12: A simulation of the model allows to execute the model step by step as well as to watch attribute values and states of the model during execution.**

The described model simulation can be used in order to analyze the model behavior interactively and graphically. The concept of MiL simulation with animation is applied in order to verify that the functional behavior of the design model is as specified in the requirements.

## 4.5 The Stopwatch Project – Generation of Production Code for Execution on the Host (Software in the loop, SiL)

In addition to model simulation that we described in the previous section, the generation of production code is an important step in the model based development process. Regarding generation of production code one usually distinguishes between execution of the generated production code on the host environment and on the final target environment. These two different execution environments are usually called SiL (Software in the Loop) and PiL (Processor in the Loop). In order to generate code for SiL, one needs to create another code generation configuration in IBM Rational Rhapsody. Similar as for the MiL, by defining several code generation options, SiL code can be generated. For SiL an important code generation option is that “Instrumentation” option needs to be set to “None”, i.e., the generated code does not contain any instrumentation code which is only needed for simulation. Additionally, one needs to define the compile environment for SiL. For the stopwatch sample we use a cygwin environment for SiL. The SiL configuration is depicted in Figure 13.

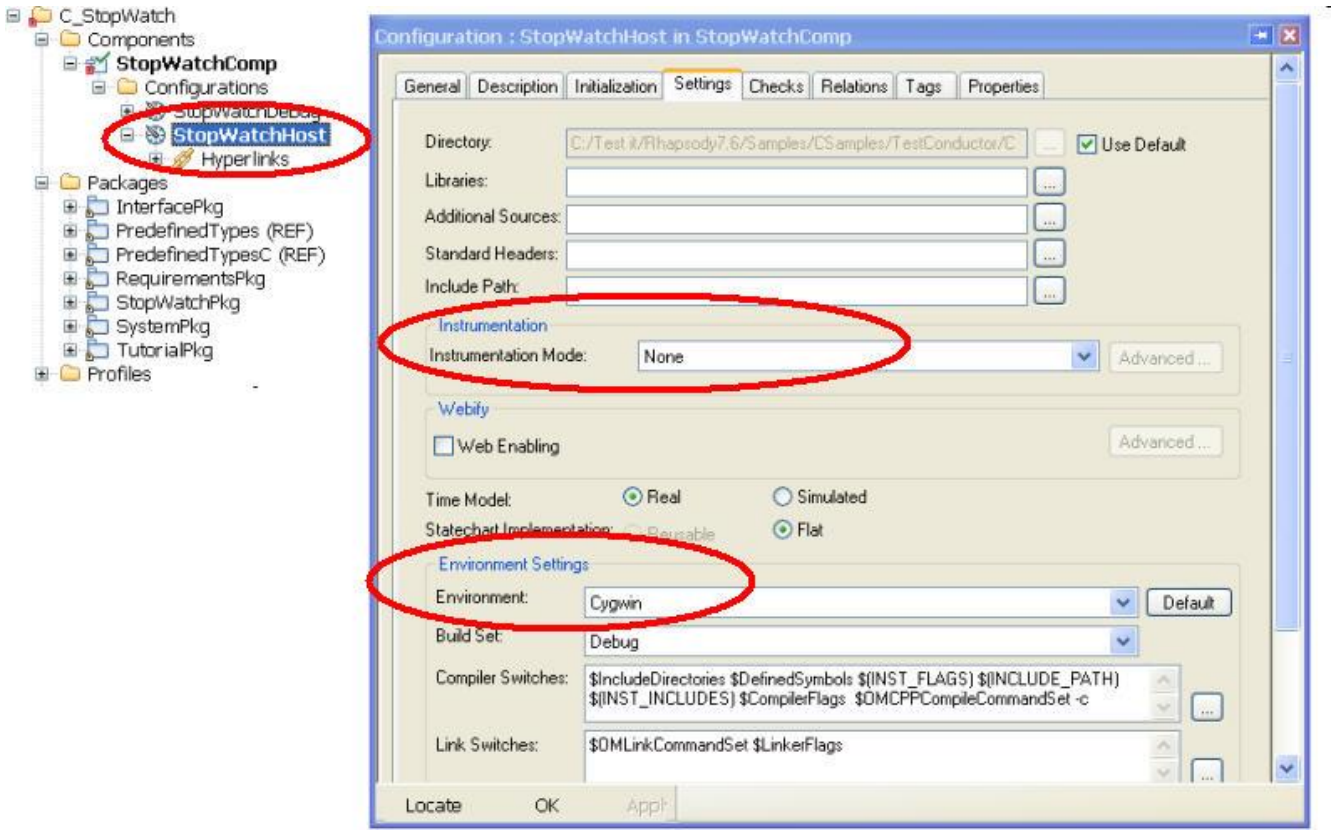



**Figure 13: Configuration for generating code for execution on the host system (SiL, Software in the loop).**

After defining a code generation configuration one can generate code for SiL with IBM Rational Rhapsody's production code generator (cf. Figure 14). Additionally, a makefile is generated that is used in order to build the generated source code for the selected compile environment. After all generated source files have been compiled the created application can be executed on the host system.

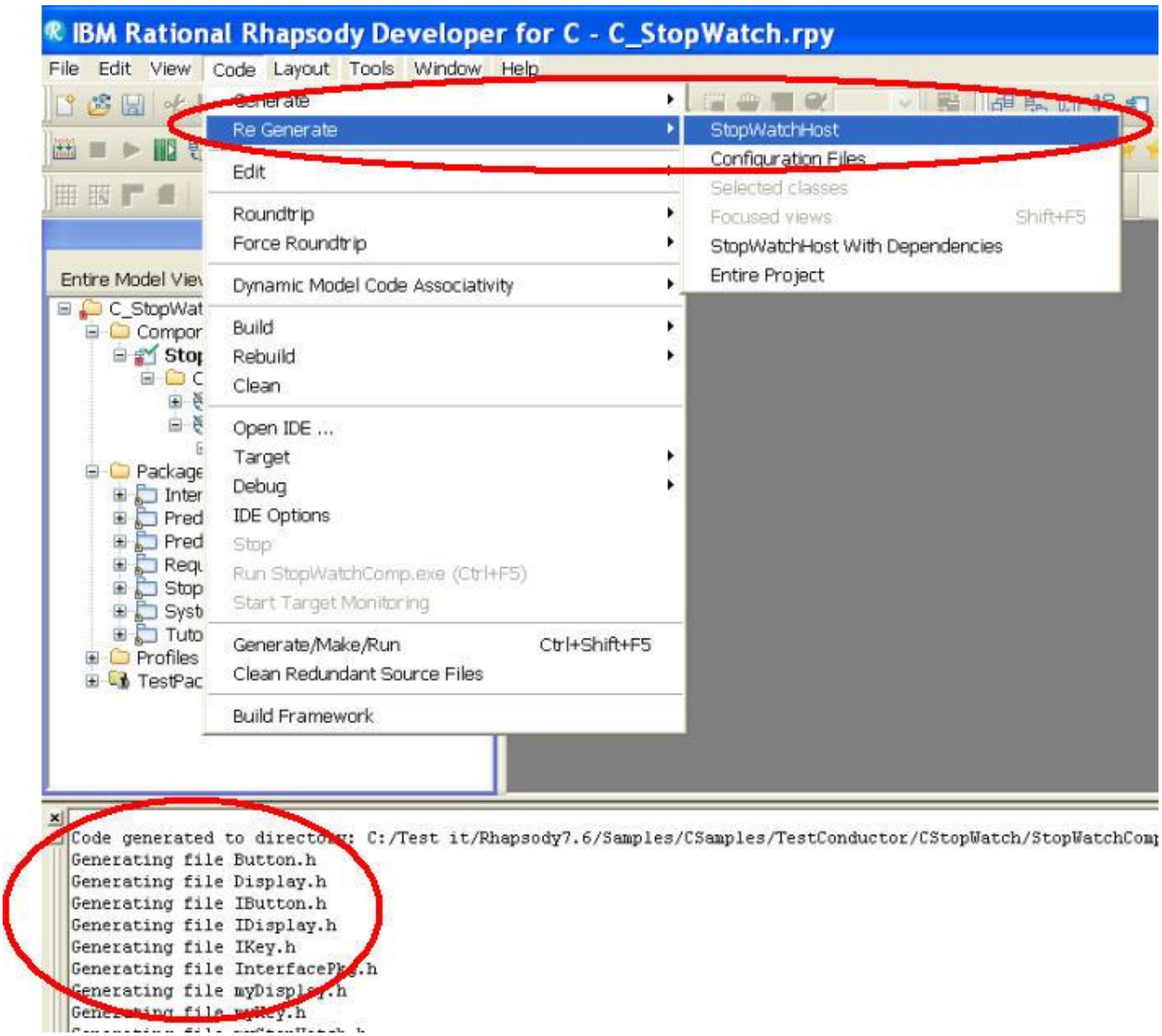



**Figure 14: Generation of production code**

The concept of SiL simulation is applied in order to verify that the functional behavior of the production code on the host system is as specified in the design model and requirements, respectively. During SiL execution on the host system some abstractions are applied regarding the final hardware and operating system.

## 4.6 The Stopwatch Project – Generation of Production Code for the Target Environment (Processor in the Loop, PiL)

For PiL code generation again a separate code generation configuration is needed quite similar to generating code for SiL. In the stopwatch sample we assume that the target environment runs an Integrity operating system (OS). Thus, this OS is chosen as environment in the code generation configuration for PiL (cf. Figure 15). As already described in the previous section one can now generate code for the target environment. The generated code can be compiled e.g. by using a cross compiler. By using a dedicated development

environment for the target system one can download the created application to the target system (either an evaluation board or the real target system) and execute it.

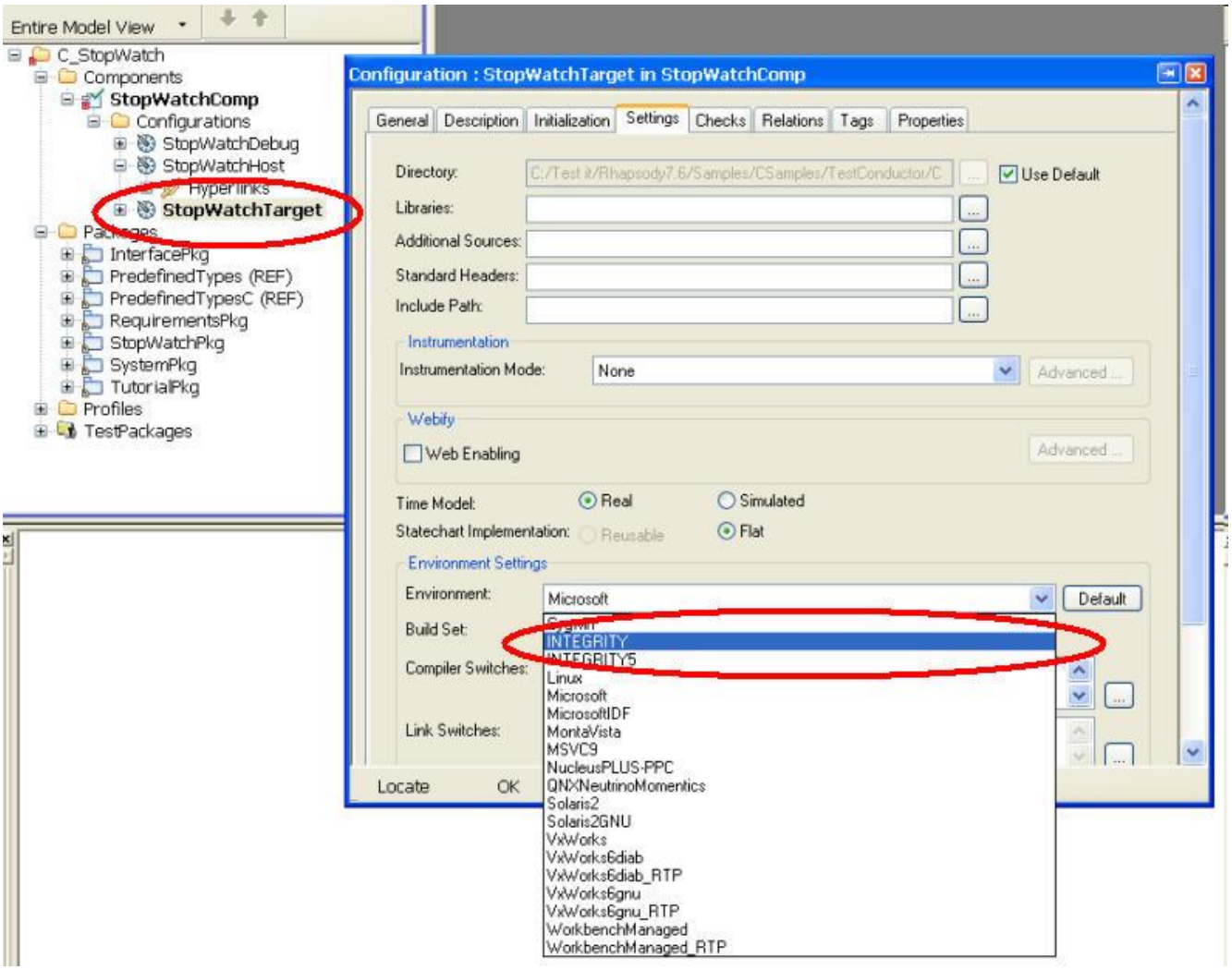



**Figure 15: IBM Rational Rhapsody configuration for generating code for the target environment.**

The concept of PiL simulation is applied in order to verify that the functional behavior of the production code on the target hardware is as specified in the design model and requirements, respectively. During PiL execution on the target system the production code is running on a processor close to the final hardware and operating system.

## 4.7 The Stopwatch Project – Verification Steps

In the previous sections we developed the stopwatch sample model, and we showed how manual and interactive simulation can be used in order to analyze and verify the behavior of the model. In this section we want to show how the developed model can be systematically verified with IBM Rational Rhapsody TestConductor Add On by means of model based test cases.

Before describing the individual verification steps in detail, we shortly sketch the general working principle of IBM Rational Rhapsody TestConductor Add On that is depicted in Figure 16.

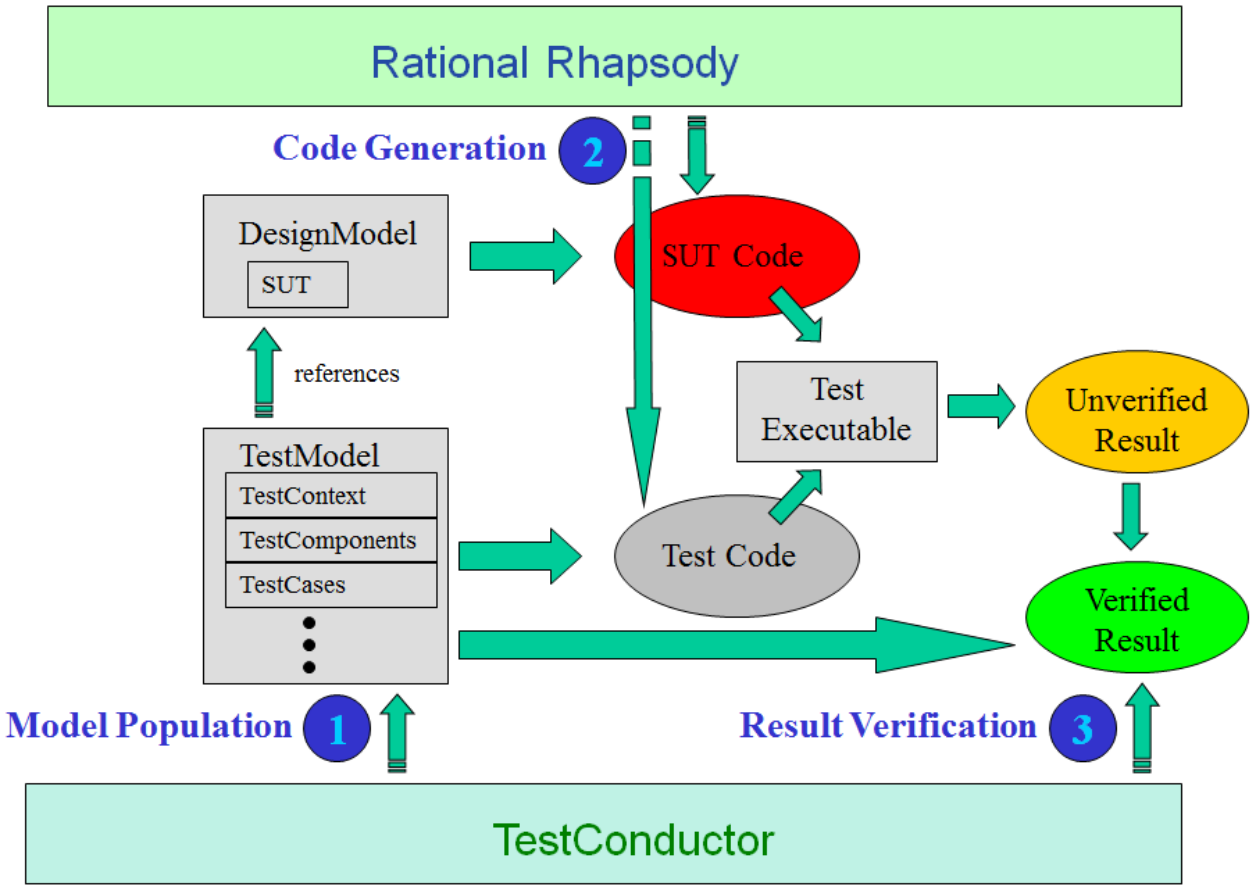



**Figure 16: Technical concepts of IBM Rational Rhapsody TestConductor Add On.**

Starting point is always a IBM Rational Rhapsody model or a part of a IBM Rational Rhapsody model. The part of the model that should be verified is called System Under Test (SUT). The SUT is depicted in Figure 16 in the upper left part. Based on the selected SUT and the test cases that are specified by the user, IBM Rational Rhapsody TestConductor Add On creates a so-called test model that defines the test architecture as well as the test behavior by means of UML diagrams and operations. For instance, one can choose a single class as SUT. In a first step, IBM Rational Rhapsody TestConductor Add On creates a test architecture for the selected class, i.e., IBM Rational Rhapsody TestConductor Add On creates additional model classes and objects solely for the purpose of testing the SUT. All test artifacts that are created by IBM Rational Rhapsody TestConductor Add On form the so-called test model. The test model is always created separately from the design model in order to make sure that the design model is not changed accidently. The test model just references the model elements in the design model, but does not make any changes to the design model. If a test case is specified by the user, IBM Rational Rhapsody TestConductor Add On creates additional classes and staecharts that realize the specified behavior of the test case. The creation of additional test artifacts based on specified test cases is called “model population” (step 1 in Figure 16).

After model population, for the purpose of test execution, IBM Rational Rhapsody TestConductor Add On uses IBM Rational Rhapsody‘s code generator in order to generate code for the SUT as well as test code for the populated test model. The generated code, both SUT code and test code, is compiled and linked into one test executable. By running the test executable the specified test cases can be executed and test results are generated. The

generated test results are considered to be intermediate results and are subject to cross verification. This is because potential errors of IBM Rational Rhapsody's code generator may have influenced the test results. In order to detect such unwanted influences on the test results, IBM Rational Rhapsody TestConductor Add On performs so-called result verification on the generated test results. The process of result verification executes a consistency check on the generated test results. The consistency check is based on the specified test cases in the model and is totally independent from IBM Rational Rhapsody's code generator. After result verification has been performed, the final test verdicts and test reports for the executed test cases are available.

Note: the granularity of the result verification check goes down to code blocks, but does not completely verify the content of code blocks. Code blocks can be used in code test cases, in flowchart and statechart test cases on e.g. transitions and in states, and in sequence diagram test cases e.g. in test actions. Rhapsody code generation copies the code blocks from the model elements into the generated source code, but does not modify the code blocks. The result verification check does not verify that the Rhapsody code generator does a proper copy action for the content of all code blocks.

Example code block:

```
i1=itsCashRegister.isNoMoreProducts();
RTC_ASSERT_NAME("check_1.1", i1==1);
itsCashRegister.addProduct(new Product(1234,"apple",100));
i2=itsCashRegister.isNoMoreProducts();
RTC_ASSERT_NAME("check_1.2", i2==0);
```

This code block might be attached to an operation body in the model. It is assumed that the Rhapsody code generator just copies the whole body into the source code. The result verification verifies that the first assertion is indeed executed during test execution. But it is not verified that the second assertion is also executed.

It is important to note that the principle testing activities (as described in Figure 16) are the same for MiL, SiL and PiL. The only difference between these execution levels is that if test cases are executed on MiL, IBM Rational Rhapsody TestConductor Add On uses IBM Rational Rhapsody's simulation information in order to compute which parts of the model are executed during execution of a test case (model coverage, cf. section 4.7.4).

In the following, we describe all testing activities that are depicted in Figure 1. The first step of all testing activities is the creation of suitable test architecture for the selected SUT.

### 4.7.1 Verification Step 1 – Creation of Test Architectures

The basis of all testing activities is a test architecture. A test architecture defines which parts of the model are tested. The term "test architecture" is defined in the so-called "UML Testing Profile". The UML Testing Profile is a UML profile that contains several new elements for the purpose of modeling test architectures, test cases and test data. For instance, the term "test case" is defined in the UML Testing Profile as an operation. This means, that a test case has the same properties as a UML operation. Furthermore, new elements can have additional properties (compared to the original element). These additional properties can be defined as so-called "tags" for the new term. Further information about UML, Profiles and the UML Testing Profile can be found in [4] and [5].

The UML Testing profile is installed together with IBM Rational Rhapsody TestConductor Add On. All testing activities are based on the UML Testing Profile. Thus, the profile needs to be added to the model before the testing activities can be started. Adding the profile can be done either manually or automatically by IBM Rational Rhapsody TestConductor Add On. In the following, we describe how IBM Rational Rhapsody TestConductor Add On adds the profile automatically. For instance, when invoking a IBM Rational Rhapsody TestConductor Add On function the first time, IBM Rational Rhapsody TestConductor Add On checks if the Testing Profile is already part of the model. If not, then it is added to the model. Usually, the first IBM Rational Rhapsody TestConductor Add On function that is invoked is the creation of a test architecture.

For the stopwatch model, we decide that class “Stopwatch” that realizes the stopwatch functionality shall be tested. Thus, we select class Stopwatch in the IBM Rational Rhapsody model and invoke the IBM Rational Rhapsody TestConductor Add On function “Create TestArchitecture” (cf. Figure 17).

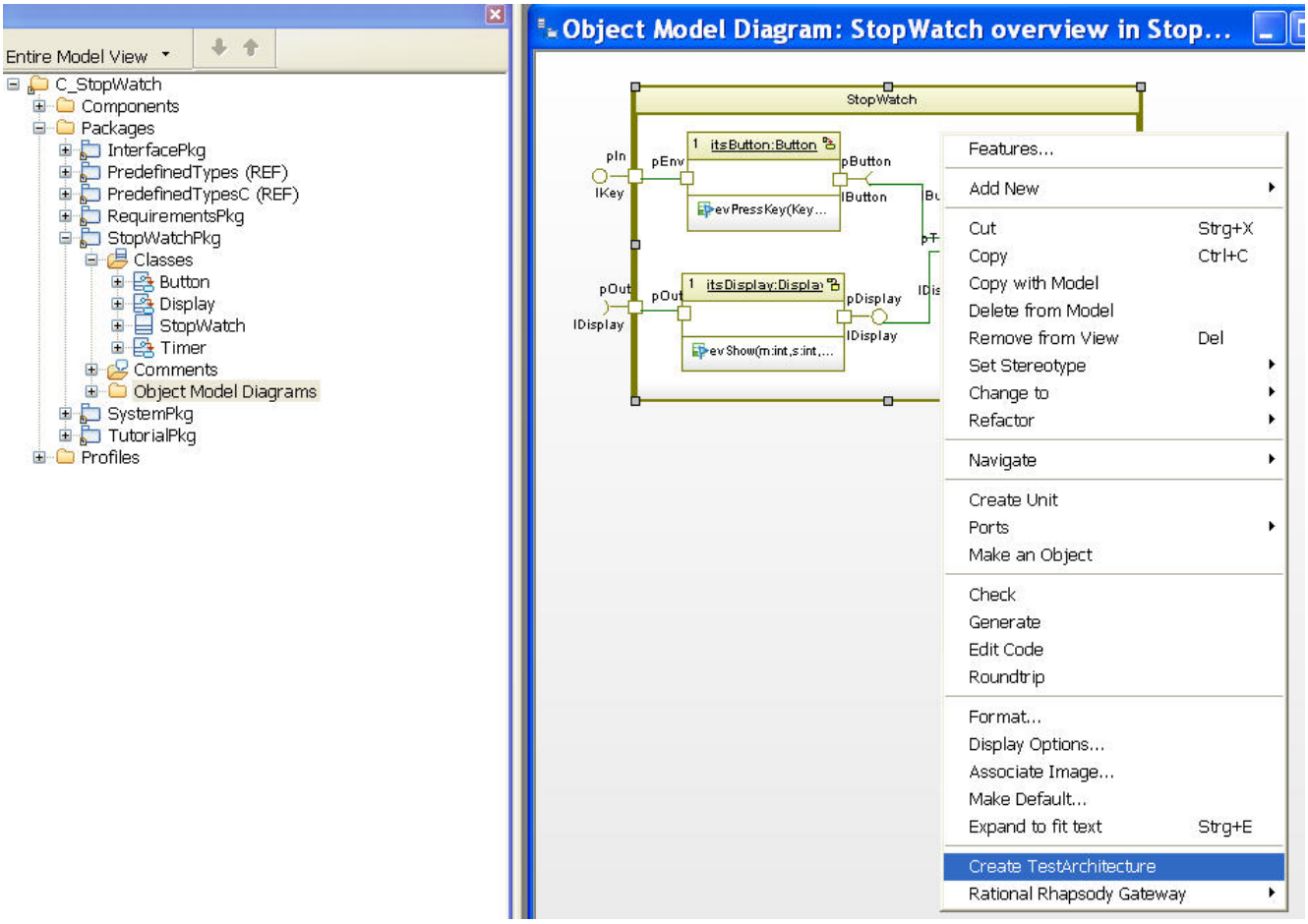



**Figure 17: Automatic creation of a test architecture with IBM Rational Rhapsody TestConductor Add On for class StopWatch.**

When this function is invoked, IBM Rational Rhapsody TestConductor Add On creates a test architecture for the selected class. The chosen class (more precisely, an instance of the chosen class) is called “SUT” (System Under Test), another new term defined in the UML Testing Profile. In addition to the SUT IBM Rational Rhapsody TestConductor Add On creates so-called “TestComponents” that are connected to the interfaces of the SUT. A test component is a class that is purely created for testing purposes. TestComponents are used in order to stimulate the SUT with inputs and to evaluate the reactions of the SUT to the

provided inputs. The test architecture that is created for the Stopwatch class can be seen in Figure 18.

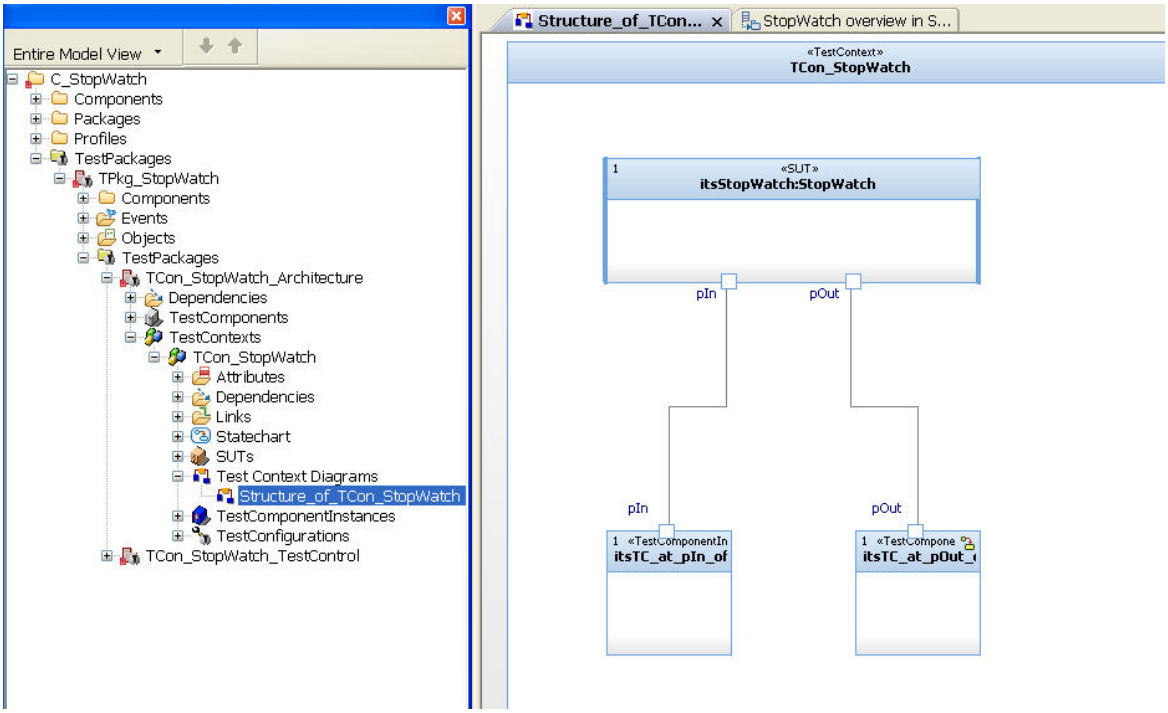



**Figure 18: Test architecture for class StopWatch**

The complete test system containing the SUT and test components is called „TestContext” in the UML Testing Profile. The structure of the test context can be seen in Figure 18 (right side). An instance of class Stopwatch (the SUT) is connected to two test components. The test components are created such that they can be connected to the ports of the SUT. With one test component one can provide inputs to the input port of the SUT (all events of the port “pIN”), and with the other test component one can evaluate the responses of the SUT to the provided inputs (all events of the port “pOUT”). In Figure 18 (left side) one can see the created test elements in the browser, e.g. the test context “TCon_Stopwtach”.

### 4.7.2 Verification step 2 – Requirements Based Testing

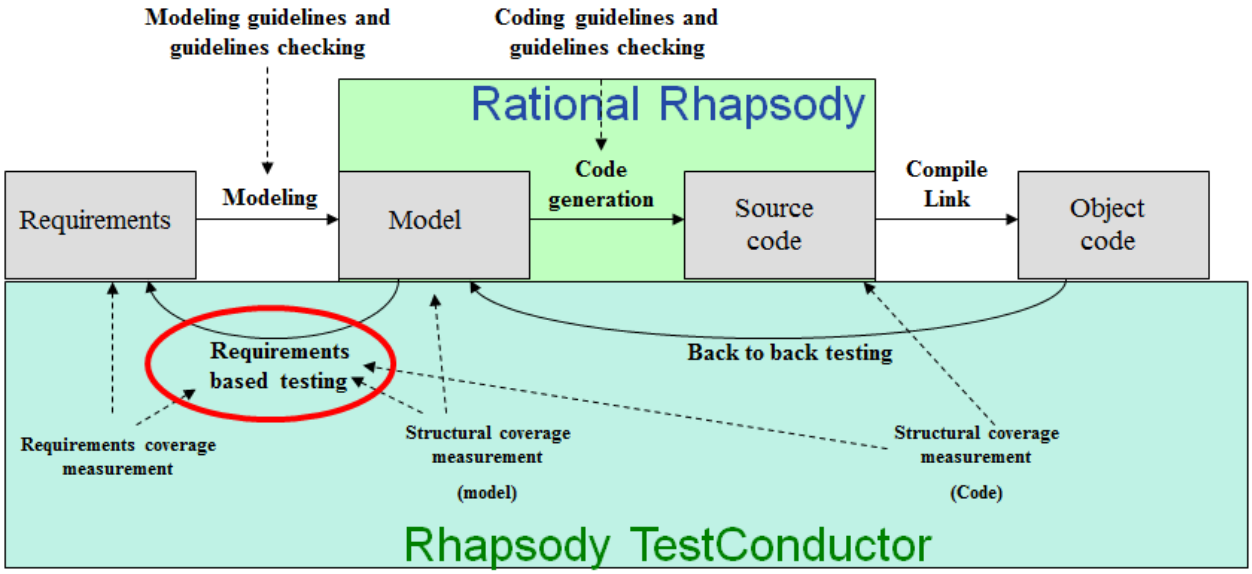



**Figure 19: Requirements based testing**

After creating a suitable test architecture for class Stopwatch, in the next step one can systematically verify if the SUT behaves as specified in the requirements. For each requirement one or more test cases are defined that shall check the behavior of the SUT. IBM Rational Rhapsody TestConductor Add On offers different ways to specify the behavior of test cases:

- Sequence diagrams
- Statecharts
- Flowcharts
- Pure test code with assertions

### 4.7.2.1 Test Case Specification with Sequence Diagrams

Depending on the requirement that shall be checked, one of these formalisms is more suitable than others. In the stopwatch sample we want to create a test case for the requirement “REQ_INIT: After starting the stopwatch, the stopwatch shall display 0 minutes and 0 seconds (0:0)”. In order to verify and test this requirement we will use a sequence diagram. Thus, we choose the IBM Rational Rhapsody TestConductor Add On function “Create SD TestCase”. As a result, we get an empty sequence diagram template that already contains instance lines for the SUT and the test components, but no messages. Now we need to add messages to the sequence diagram that specify the behavior of the test case. For the mentioned requirement the completed sequence diagram can be seen in Figure 20.

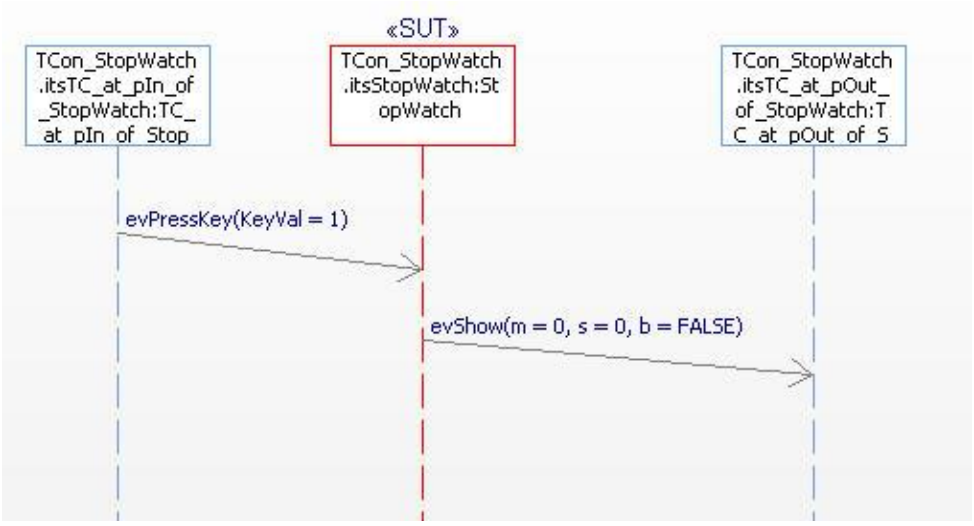



**Figure 20: Defining the behavior of a test case with a sequence diagram.**

First, an input „evPressKey(KeyVal=1)“ is sent to the SUT. This input means that the stopwatch is started. As expected reaction the sequence diagram specifies that the SUT shall emit event „evShow(m=0,s=0,b=FALSE)“. This means that the stopwatch shall display time “0:0”.

After we have defined the behavior of the test case, we need to link the test case to the requirement that shall be tested. This can be done by adding a so-called “TestObjective” to the test case that points to the requirement. The test objective explicitly links the test case to the requirement which can be seen in Figure 21. It enables traceability between the requirement and the test case.

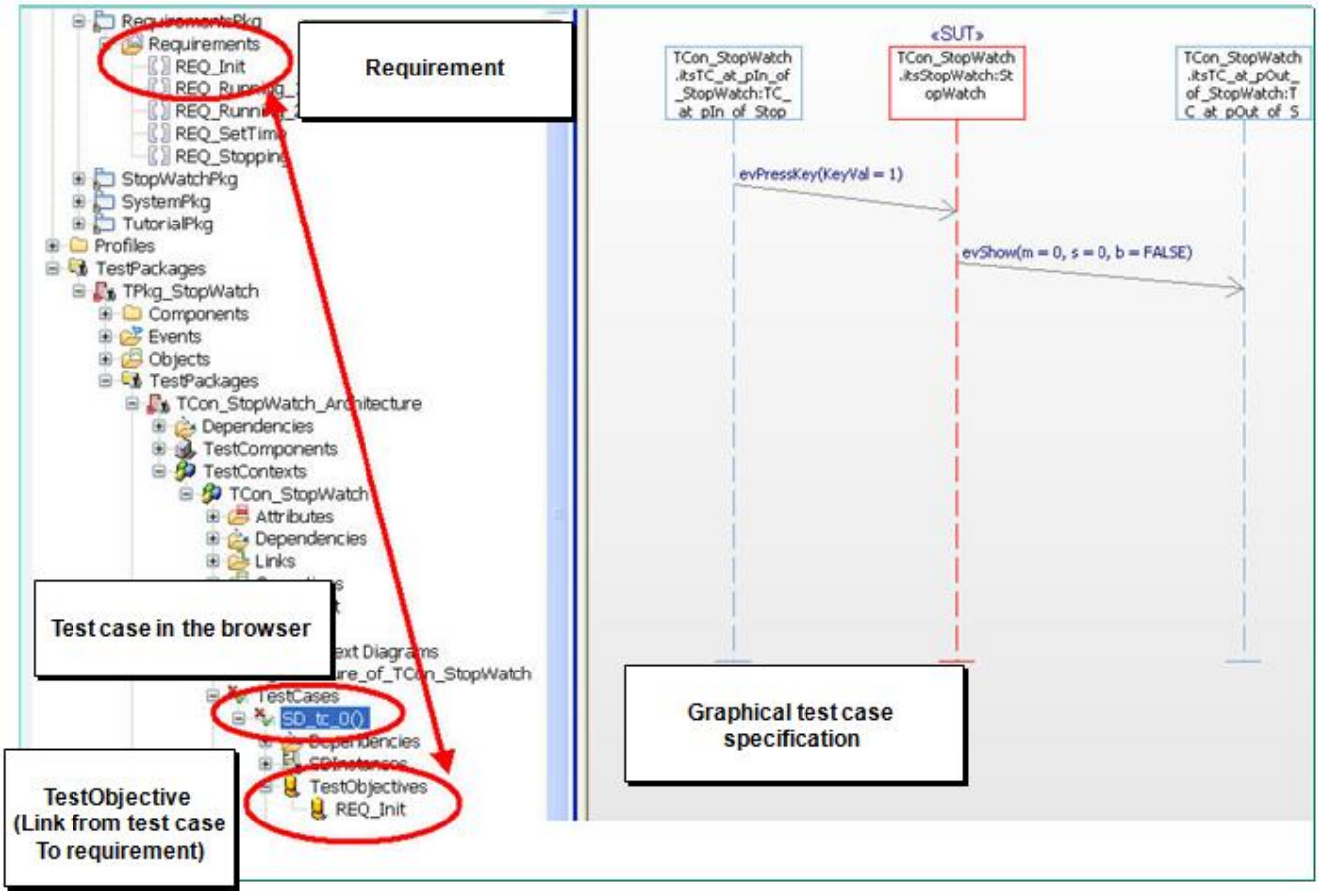



**Figure 21: Linking a test case to a requirement with a test objective element.**

After defining the test case and linking it to a requirement, in the next step the test case is executed. In order to execute a test case we first need to define if the test case shall be executed for MiL, SiL, or PiL. As described in section 4.4, we need to have an appropriate IBM Rational Rhapsody component and configuration. When creating a test architecture, IBM Rational Rhapsody TestConductor Add On automatically creates a component and configuration suitable for MiL. This is depicted in Figure 22.


**Figure 22: Test configuration for MiL execution.**

In order to execute the test case for MiL, the behavior specified graphically must be „populated" to the test model. This population step is necessary since IBM Rational Rhapsody TestConductor Add On needs to generate test code that implements the specified test behavior. In order to generate that testing code, IBM Rational Rhapsody TestConductor Add On first adds additional testing artifacts to the test model (this process is called "model population") that realize the specified testing behavior. After that, IBM Rational Rhapsody TestConductor Add On utilizes IBM Rational Rhapsody's code generator to generate the testing code from the testing model. As a concrete example, let's have a look at the test case from Figure 20. Before this test case can be executed, during model population IBM Rational Rhapsody TestConductor Add On automatically adds so-called "DriverOperations" and "StubOperations" to the testing model. Driver operations are dedicated operations that realize generation of inputs to the SUT. Stub operations are dedicated operations that realize the verification of the reactions of the SUT to the provided inputs. For the test case depicted in Figure 20, a driver operation is populated for the input message and a stub operation is populated for the output message. Within these operations, C test code is used in order to generate the input to the SUT and to check the reaction of the SUT.

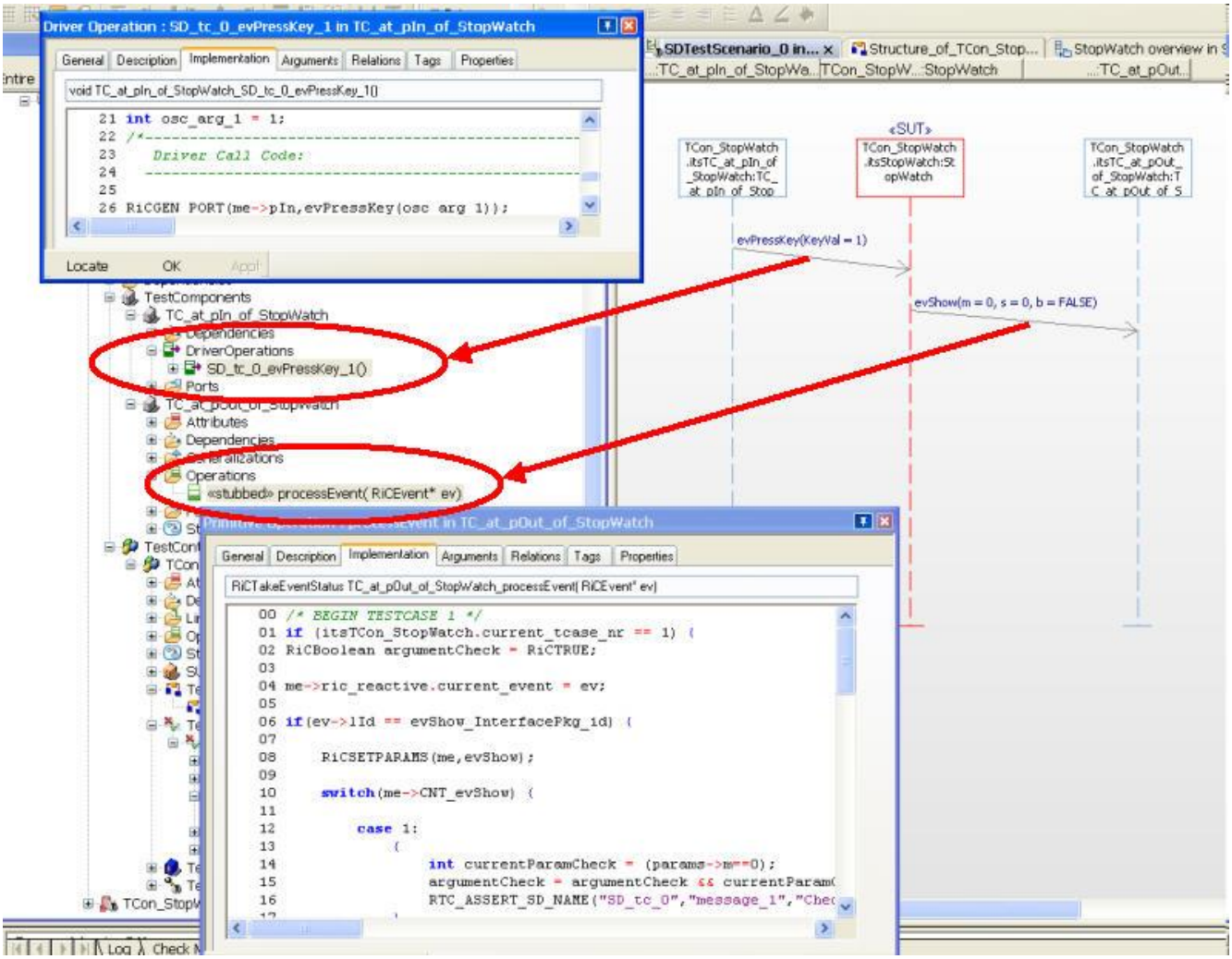



**Figure 23: Model population adds test elements to the model that realize the behavior of the test case.**

In addition to driver operations and stub operations, for sequence diagram test cases IBM Rational Rhapsody TestConductor Add On populates a so-called „Arbiter". An arbiter is a test component that contains a statechart controlling the arbitration of the different test

components that interact during execution of a sequence diagram test case. In addition to that, the arbiter also checks and verifies that the reactions of the SUT are indeed observed as specified in the scenario specification. This is realized by means of control events that are sent from the test components to the arbiter. The arbiter uses these control events in order to detect if reactions of the SUT are performed in the specified order. The arbiter communicates with the test components in order to fully control the test execution. If the SUT does not produce outputs in the order as specified in the test case, the statechart of the arbiter changes into a dedicated “fail” state, and the test case is evaluated as failed. The arbiter for the test case depicted in Figure 20 can be seen in Figure 24.

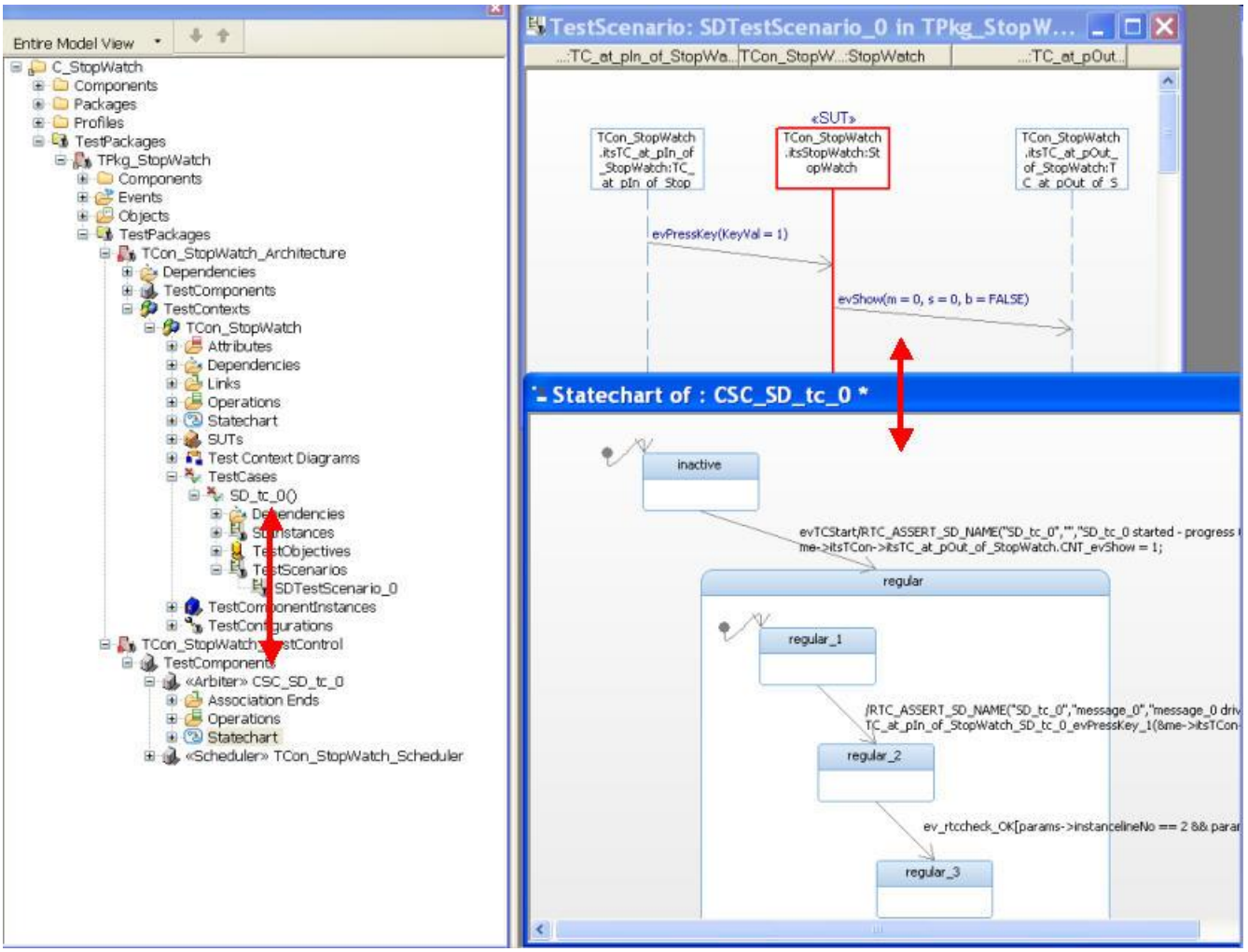



**Figure 24: Arbiter statechart to control the behavior of the test components that realize the test case.**

After model population has populated all needed test artifacts to the testing model, IBM Rational Rhapsody TestConductor Add On utilizes IBM Rational Rhapsody’s code generator in order to generate test code for the SUT and the testing model. After code generation, the code is compiled and linked to a test executable. This test executable can now be executed by invoking the “Execute TestCase” function of IBM Rational Rhapsody TestConductor Add On. If the test executable is invoked, it starts the IBM Rational Rhapsody simulation. After the simulation has started, the test executable executes the test case. After test case execution has finished, the test results are shown in the so-called “Test Execution Window” within the IBM Rational Rhapsody environment (cf. Figure 25 bottom left). Besides the test results shown in the test execution window, also a test result report is generated and stored underneath the test case in the IBM Rational Rhapsody model. The test execution report

contains additional information about the test execution, e.g. the test execution time, as well as the test result.

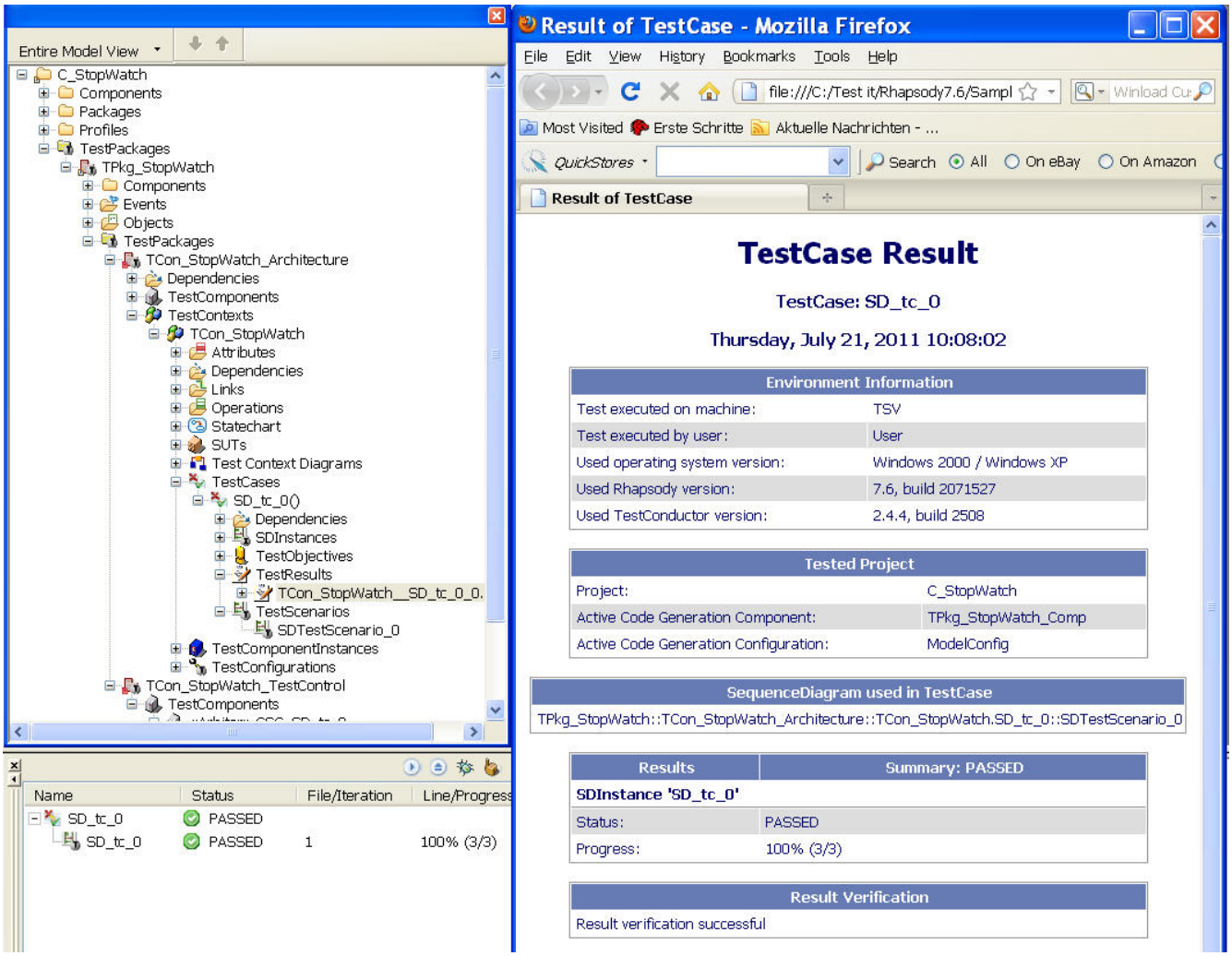



**Figure 25: Test execution window (bottom left) and test report (right).**

### 4.7.2.2 Test Case Specification with Statecharts, Flowcharts, and Code

As an alternative to defining the behavior of a test case with a sequence diagram, IBM Rational Rhapsody TestConductor Add On provides the possibility to describe the behavior of test cases with statecharts, flowcharts, or pure test code. As an example, we study requirement “REQ_SetTime: The stopwatch shall provide a function SetTime that sets the current time”. This requirement can be tested e.g. by a statechart test case as depicted in Figure 26. In a statechart test case, similar as in sequence diagram test cases inputs can be provided for the SUT. In order to check outputs of the SUT as e.g. return values, IBM Rational Rhapsody TestConductor Add On provides several predefined check functions like e.g. “RTC_ASSERT_NAME”. This function takes two arguments, a reference string and a boolean expression. The Boolean expression realizes the check that is evaluated by IBM Rational Rhapsody TestConductor Add On during test case execution. If the test case is executed, all executed assertions are logged by IBM Rational Rhapsody TestConductor Add On and shown in the test execution window. Similar to sequence diagram test cases, also a test

report is generated that contains all executed assertions as well as further details about the test execution like e.g. execution time.

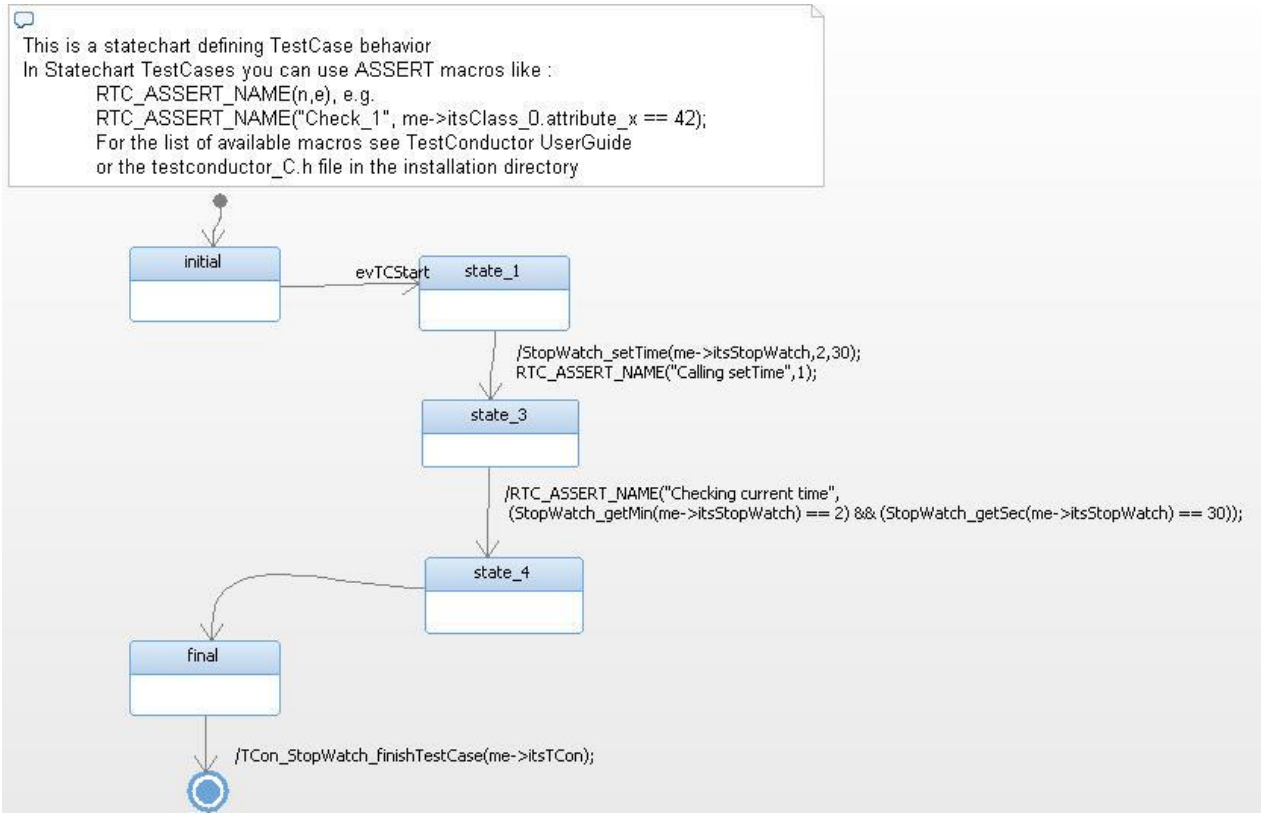



**Figure 26: Test case definition by means of a statechart.**

The test execution window that contains the executed assertions as well as the generated execution report is depicted in Figure 27.

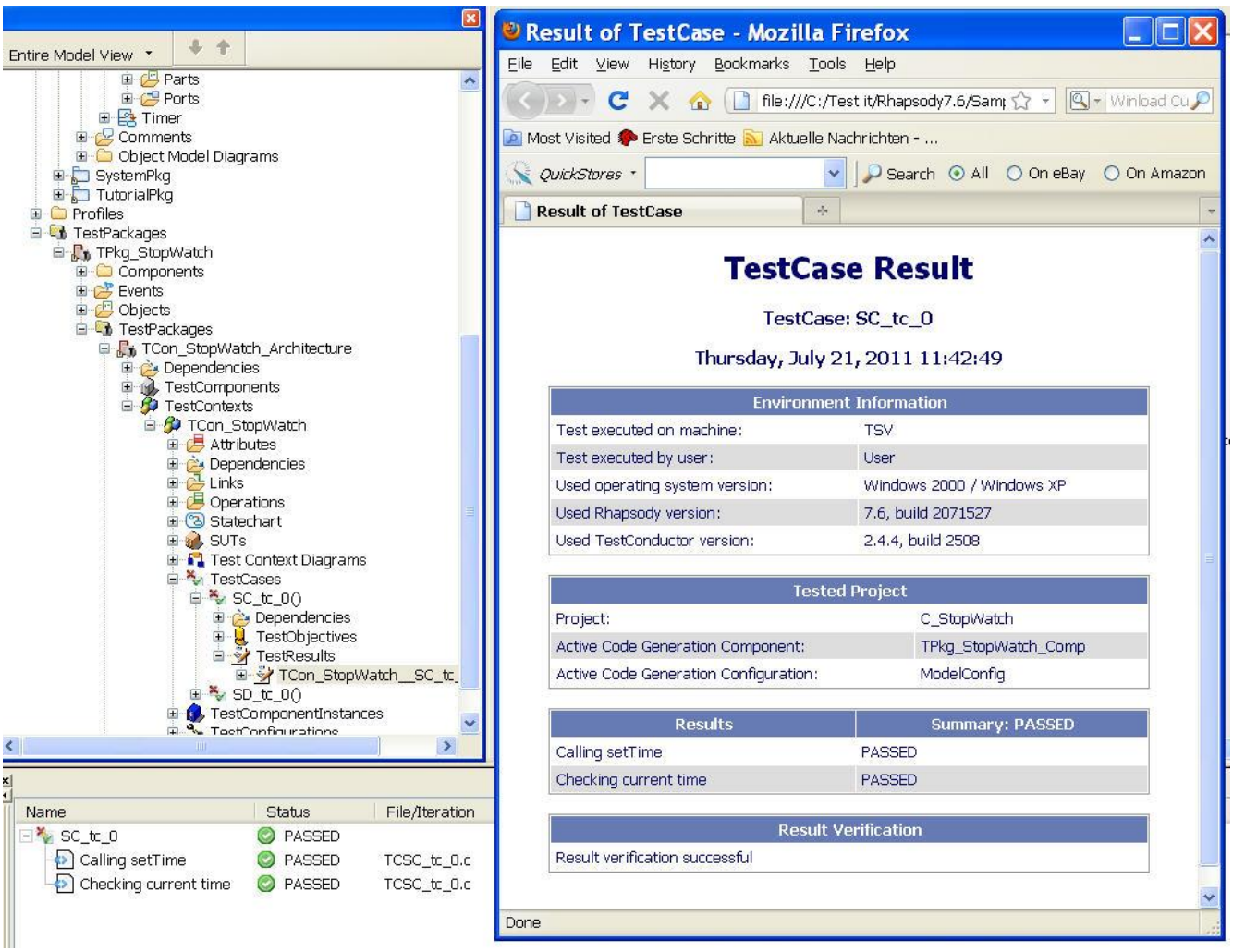



**Figure 27: Test execution of a statechart test case.**

As an alternative to statecharts, the behavior of test cases can also be defined by specifying a so-called flowchart. A flowchart specification for the requirement “REQ_SetTime” is depicted in Figure 28.

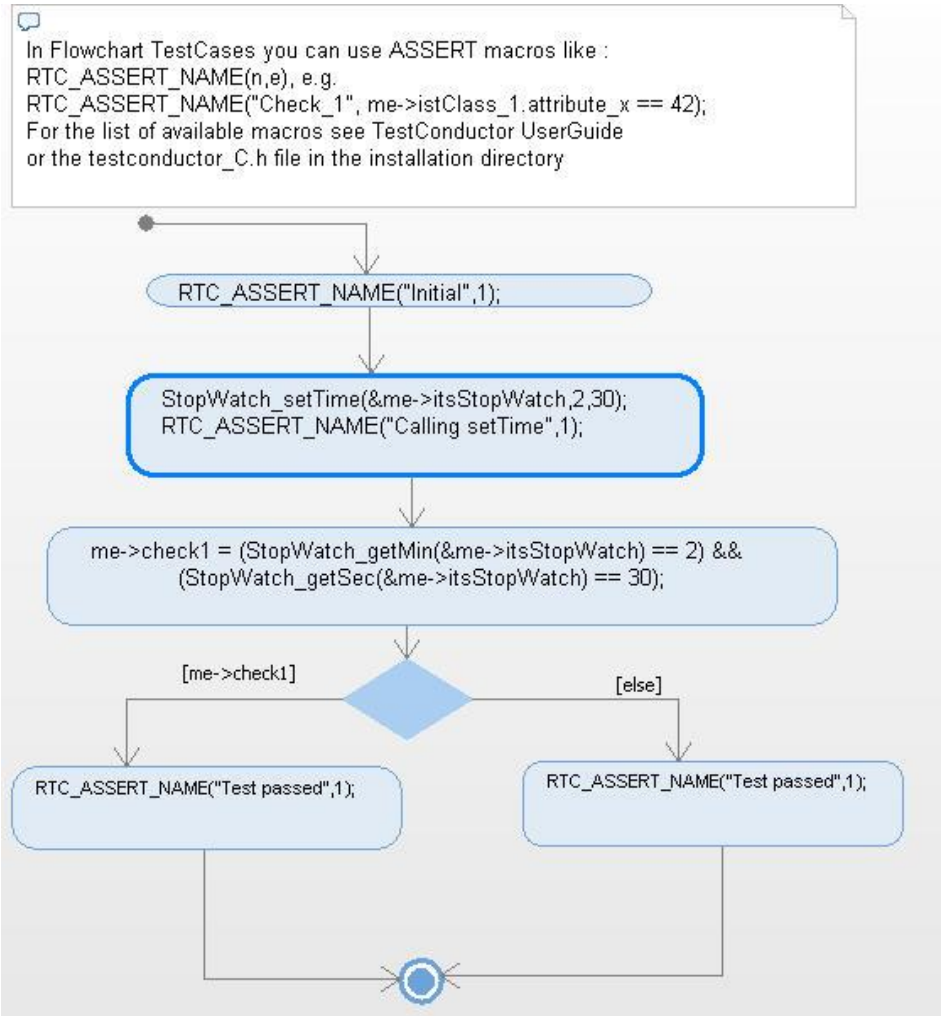



**Figure 28: Test Case definition by means of a flowchart.**

As a last alternative, the behavior of a test case can also be specified by providing C or C++ test code containing assertion functions to check the correctness of the reactions of the SUT regarding provided inputs. Such a code test case can be seen in Figure 29.

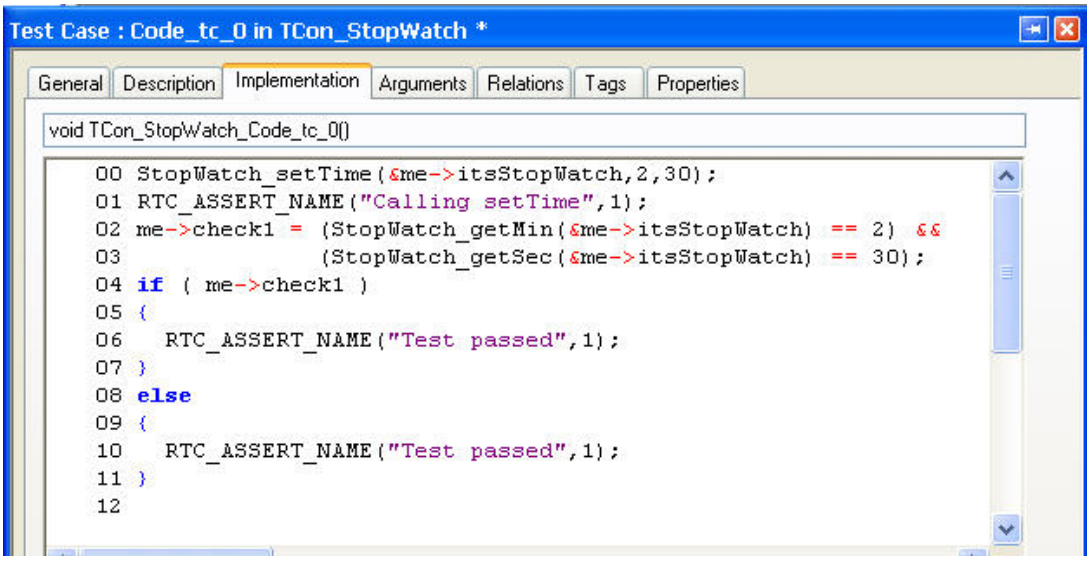



**Figure 29: Test case definition by means of C code.**

Both flowcharts and code test cases can be executed in the same way as other test cases.

### 4.7.3 Verification Step 3 – Coverage of the Requirements by Test Cases

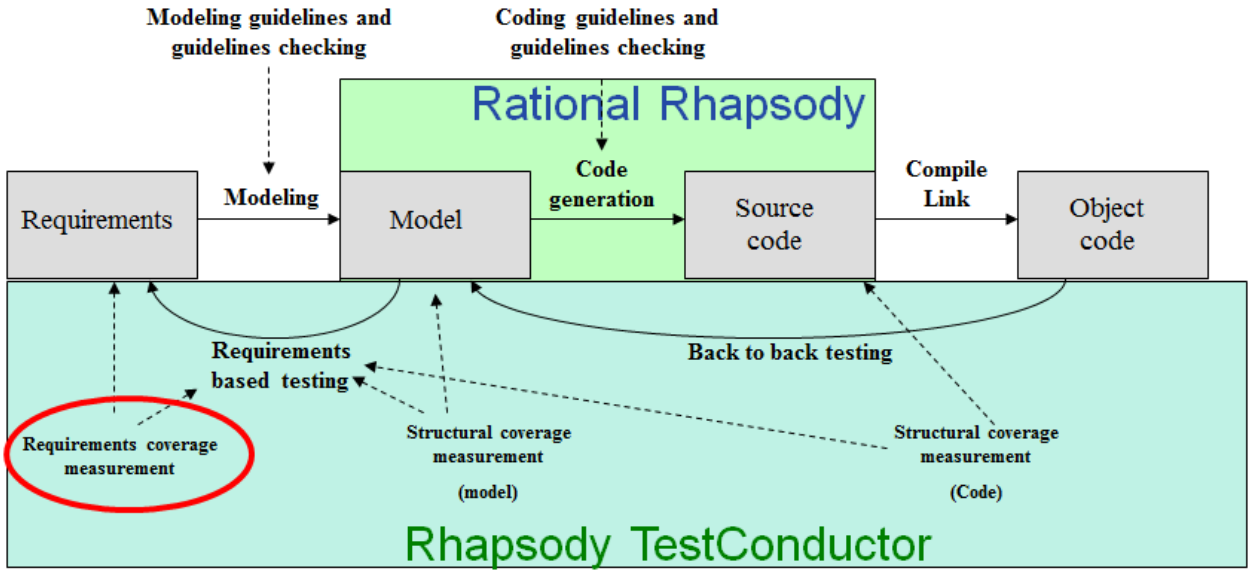



**Figure 30: Requirements coverage**

In the previous section we showed how to create test cases for requirements by means of different UML diagrams, and how such test cases can be linked to requirements. An imported question is which requirements are tested by which test cases, and even more important, which requirements have not been tested by a test case. IBM Rational Rhapsody TestConductor Add On provides two mechanisms in order to answer these questions. Firstly, a so-called “TestRequirementsMatrix” can be used in order to automatically visualize the relationship between requirements and test cases. This matrix is pre-defined in the testing profile and can be added to the test model in order to get an overview about the relationship between requirements and test cases. After adding the matrix to the testing model, the user needs to specify the scope of the matrix, i.e., which parts of the model should be shown in the matrix. After defining the scope of the matrix, the matrix shows the current coverage of requirements by test cases as it is depicted in Figure 31.

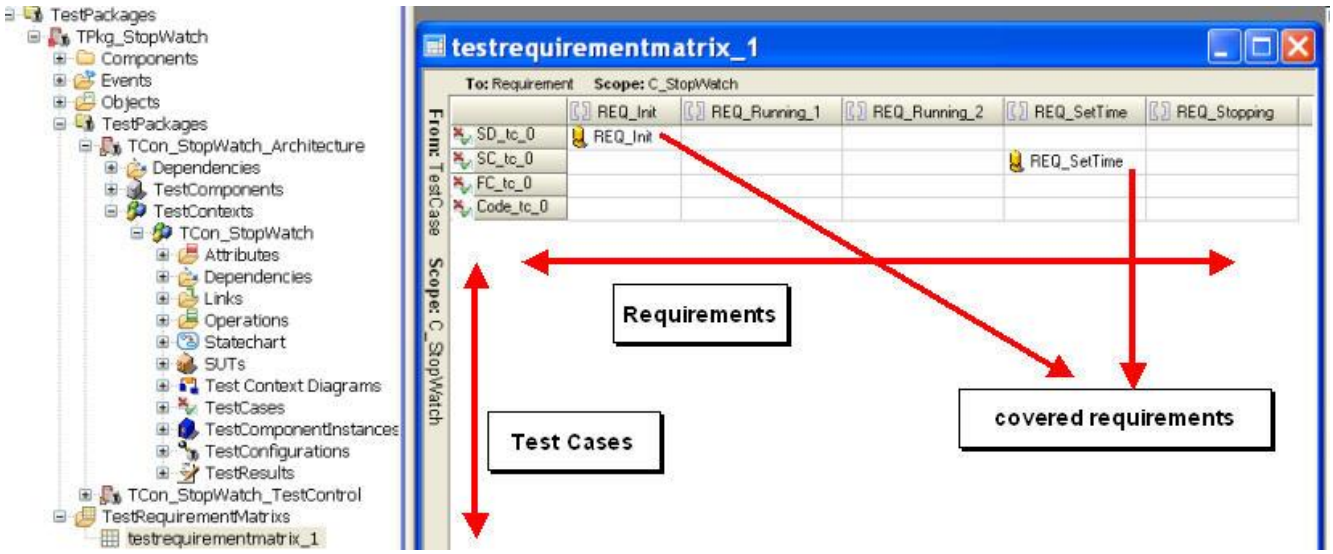



**Figure 31: Requirements coverage visualized by a test requirements matrix.**

The requirements are shown on the horizontal axis, the test cases are shown on the vertical axis. If a test case is linked to a requirement by a test objective a yellow test objective symbol is shown at the intersection point within the matrix. By looking at the test requirements matrix one can visually see which requirements are covered by which test cases and which requirements are not covered by a test case.
As an alternative to the test requirements matrix one can also generate a dedicated test requirements report that provides similar information. The test requirements report can be generated with the ReporterPlus AddOn of IBM Rational Rhapsody. IBM Rational Rhapsody TestConductor Add On provides a so-called ReporterPlus template that can be used in order to generate such a report. Such a report is depicted in Figure 32.

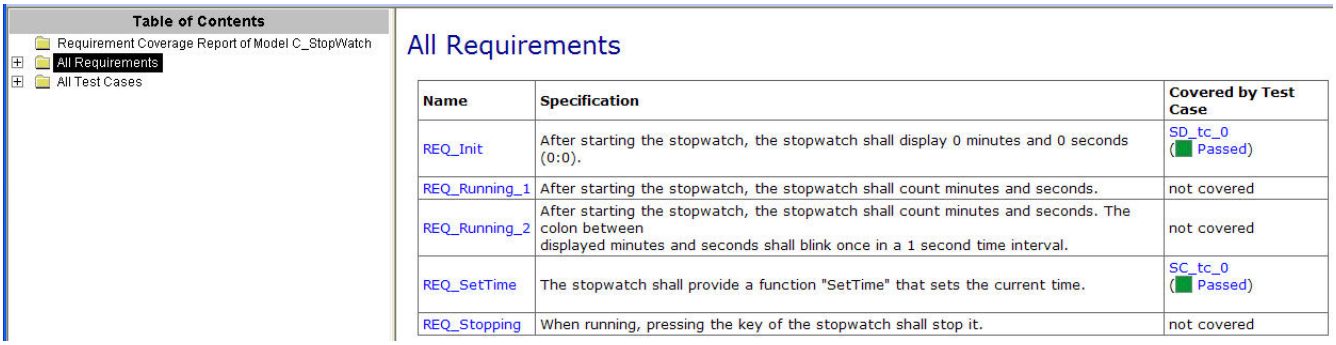



**Figure 32: Requirements coverage information shown in a test requirements coverage report.**

The TestRequirementsCoverage report can be generated in different formats, e.g. html or word format. The report basically provides two orthogonal views. The first view shows a list of all requirements together with linked test cases and test results (if available). The second view shows a list of all test cases together with linked requirements. Both the test requirements matrix as well as the TestRequirementsCoverage report provide information about which requirements are covered by which test cases and which requirements are not covered by test cases. In order to achieve full requirements coverage in the stopwatch sample, we would need to add more test cases that cover all requirements. After adding these test cases, the requirement coverage would look like the one depicted in Figure 33.

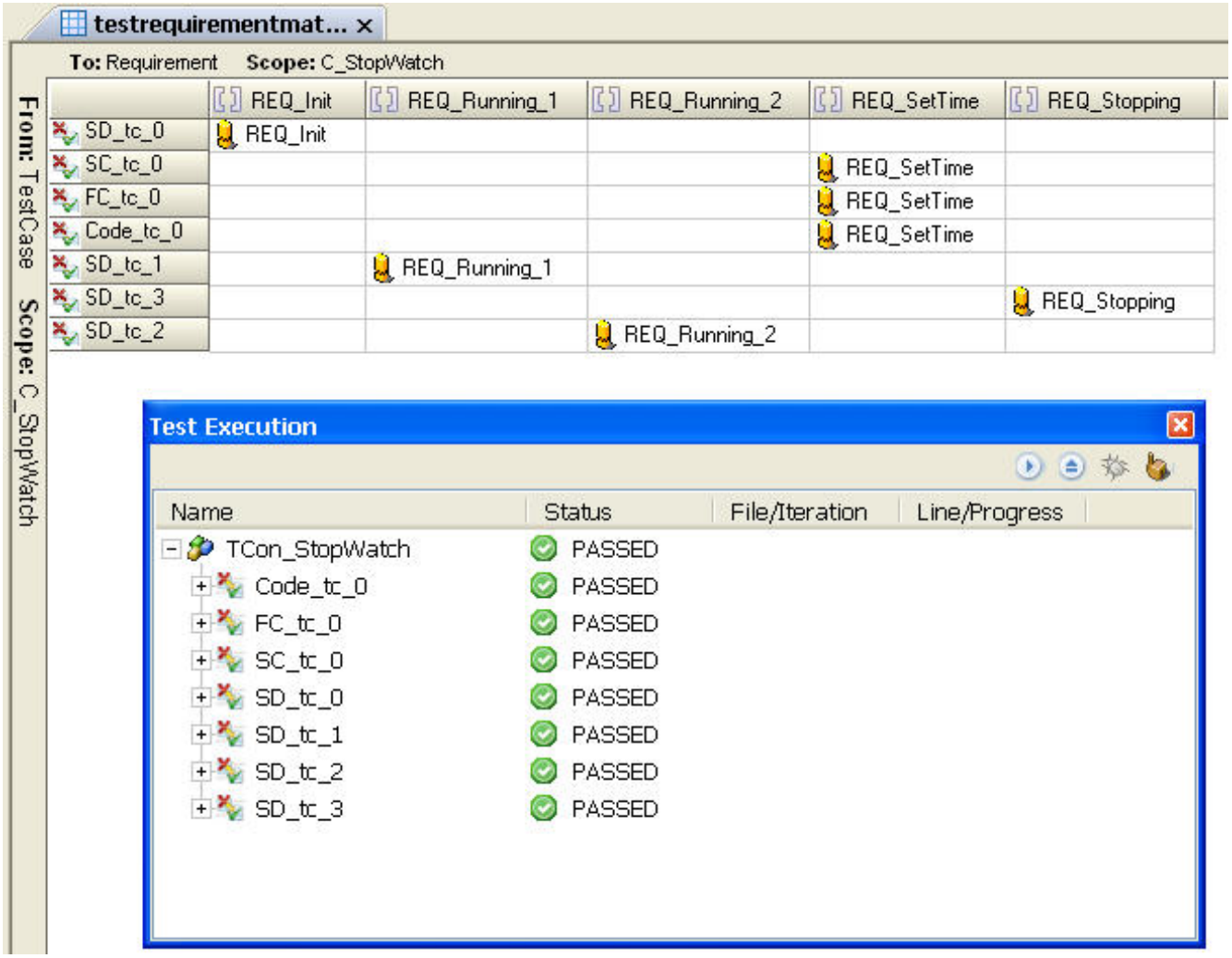



**Figure 33: Full requirements coverage by test cases, and all test cases are passed.**

### 4.7.4 Verification Step 4 – Coverage of the Model by Test Cases

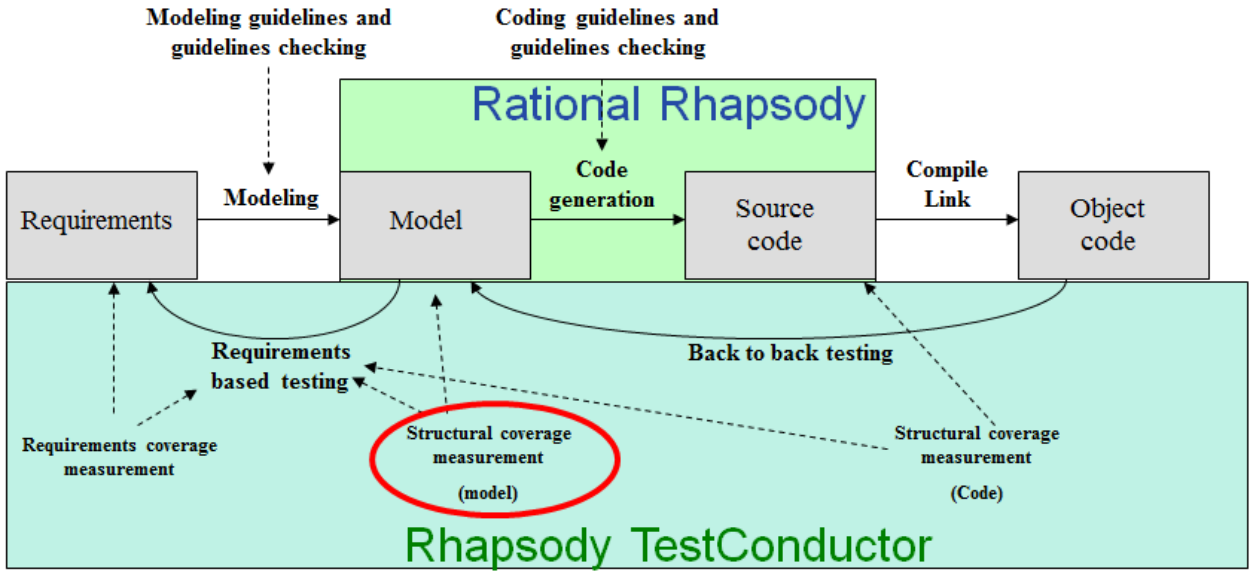
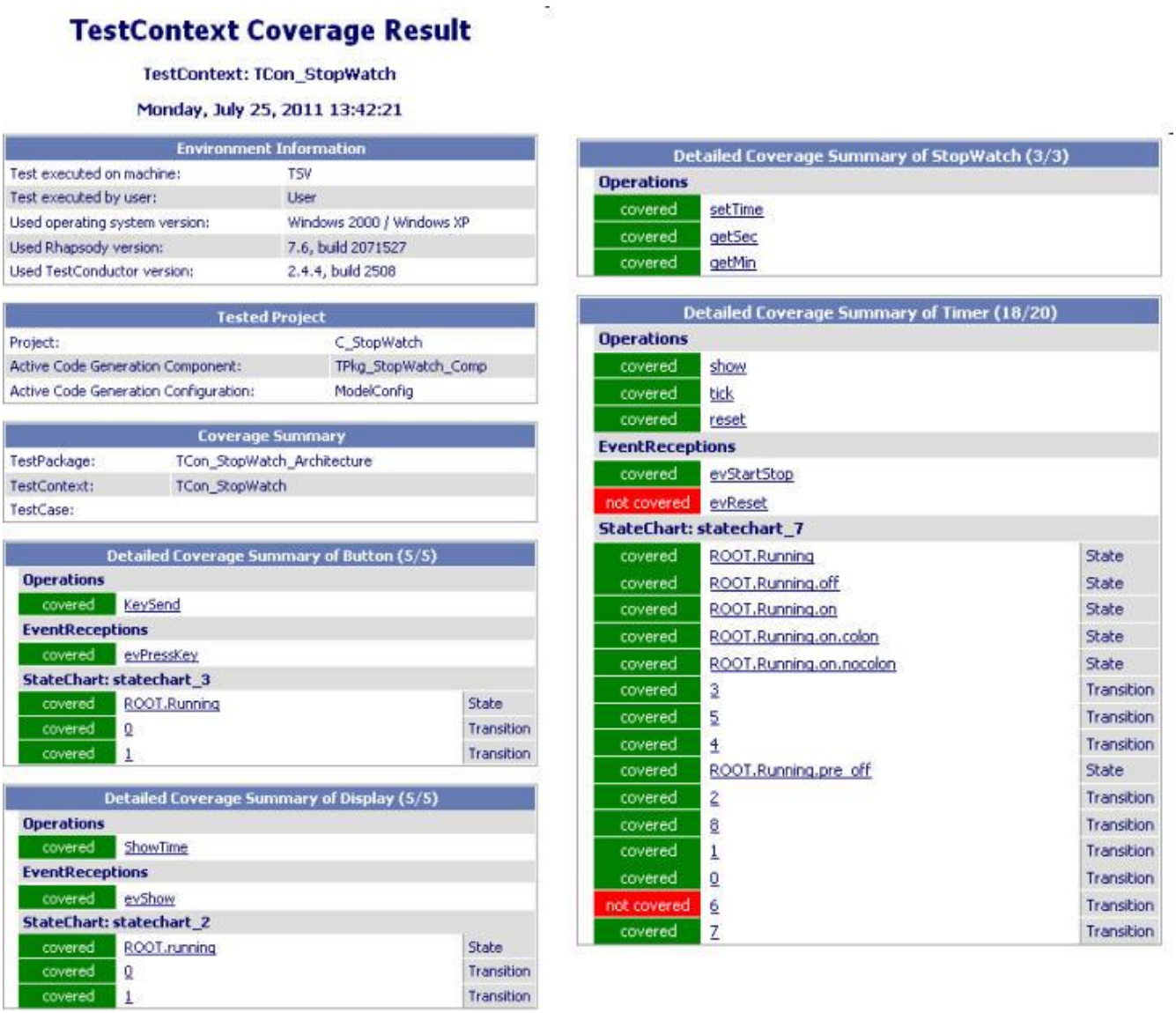



**Figure 34: Model coverage**

In the previous section we have shown how to verify that all requirements are covered by test cases. An important orthogonal information is the information which parts of the model are covered when executing all the test cases that are needed for full requirements coverage. To retrieve this information, IBM Rational Rhapsody TestConductor Add On provides the option to compute the achieved model coverage during test case execution of MiL configurations. If this option is enabled, after test case execution IBM Rational Rhapsody TestConductor Add On generates a so-called model coverage report that shows which parts of the model have been covered by the executed test cases and which parts have not been executed by the test cases. For the test cases developed in the previous section, a model coverage report as shown in Figure 35 is generated. The model coverage report shows all states, transitions, events and operations of the SUT (and all inner components of the SUT). For all listed model elements it is specified if the model element has been executed or not, i.e. covered or nor. Model coverage reports can be generated for individual test cases as well as for complete test suites.



**TestContext Coverage Result**

TestContext: TCon_StopWatch

Monday, July 25, 2011 13:42:21

| Environment Information | |
|---|---|
| Test executed on machine: | TSV |
| Test executed by user: | User |
| Used operating system version: | Windows 2000 / Windows XP |
| Used Rhapsody version: | 7.6, build 2071527 |
| Used TestConductor version: | 2.4.4, build 2508 |

| Tested Project | |
|---|---|
| Project: | C_StopWatch |
| Active Code Generation Component: | TPkg_StopWatch_Comp |
| Active Code Generation Configuration: | ModelConfig |

| Coverage Summary | |
|---|---|
| TestPackage: | TCon_StopWatch_Architecture |
| TestContext: | TCon_StopWatch |
| TestCase: | |

| Detailed Coverage Summary of Button (5/5) | | |
|---|---|---|
| **Operations** | | |
| covered | KeySend | |
| **EventReceptions** | | |
| covered | evPressKey | |
| **StateChart: statechart_3** | | |
| covered | ROOT.Running | State |
| covered | 0 | Transition |
| covered | 1 | Transition |

| Detailed Coverage Summary of Display (5/5) | | |
|---|---|---|
| **Operations** | | |
| covered | ShowTime | |
| **EventReceptions** | | |
| covered | evShow | |
| **StateChart: statechart_2** | | |
| covered | ROOT.running | State |
| covered | 0 | Transition |
| covered | 1 | Transition |

| Detailed Coverage Summary of StopWatch (3/3) | |
|---|---|
| **Operations** | |
| covered | setTime |
| covered | getSec |
| covered | getMin |

| Detailed Coverage Summary of Timer (18/20) | | |
|---|---|---|
| **Operations** | | |
| covered | show | |
| covered | tick | |
| covered | reset | |
| **EventReceptions** | | |
| covered | evStartStop | |
| not covered | evReset | |
| **StateChart: statechart_7** | | |
| covered | ROOT.Running | State |
| covered | ROOT.Running.off | State |
| covered | ROOT.Running.on | State |
| covered | ROOT.Running.on.colon | State |
| covered | ROOT.Running.on.nocolon | State |
| covered | 3 | Transition |
| covered | 5 | Transition |
| covered | 4 | Transition |
| covered | ROOT.Running.pre_off | State |
| covered | 2 | Transition |
| covered | 8 | Transition |
| covered | 1 | Transition |
| covered | 0 | Transition |
| not covered | 6 | Transition |
| covered | 7 | Transition |

**Figure 35: Model coverage achieved by requirements based test cases.**

As can be seen in Figure 35, all elements except event “evReset” and transition 6 of class “Timer” (an inner part of the SUT) are executed by the test cases. The model elements in the model coverage report are linked to the model elements in the IBM Rational Rhapsody model, i.e., when clicking on a model element in the report the corresponding model element in the IBM Rational Rhapsody model is highlighted. When clicking on transition 6 in the report, the not covered transition gets highlighted in the IBM Rational Rhapsody model (cf. Figure 36). This transition is not covered by the test cases since the modeled reset functionality of the stopwatch is not specified in any of the requirements of the stopwatch In such a case one needs to decide if the reset functionality is wanted or unwanted functionality. In our example, we assume that it is wanted behavior, and we add new requirement “REQ_Reset” that specifies this functionality. Additionally, we add a new test case that tests this functionality. The updated model coverage report is depicted in Figure 37.

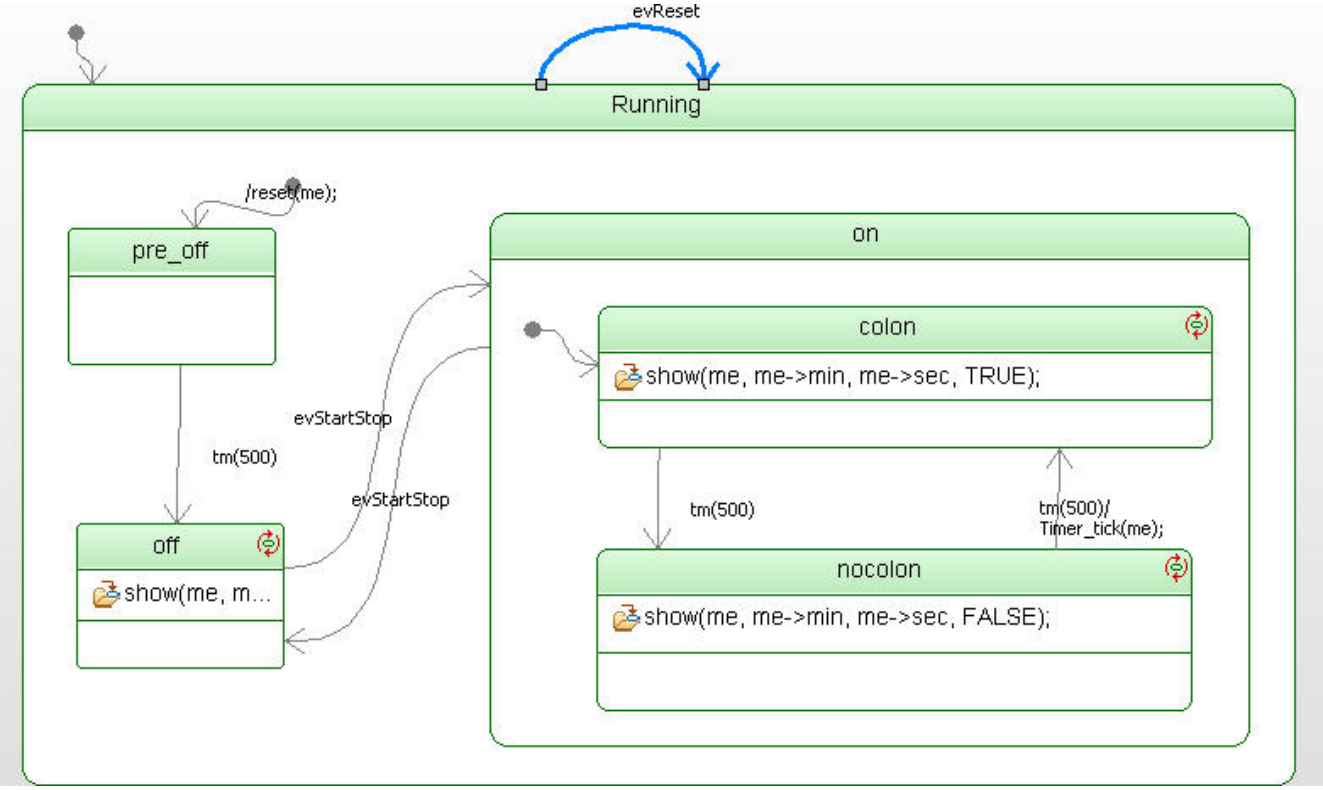
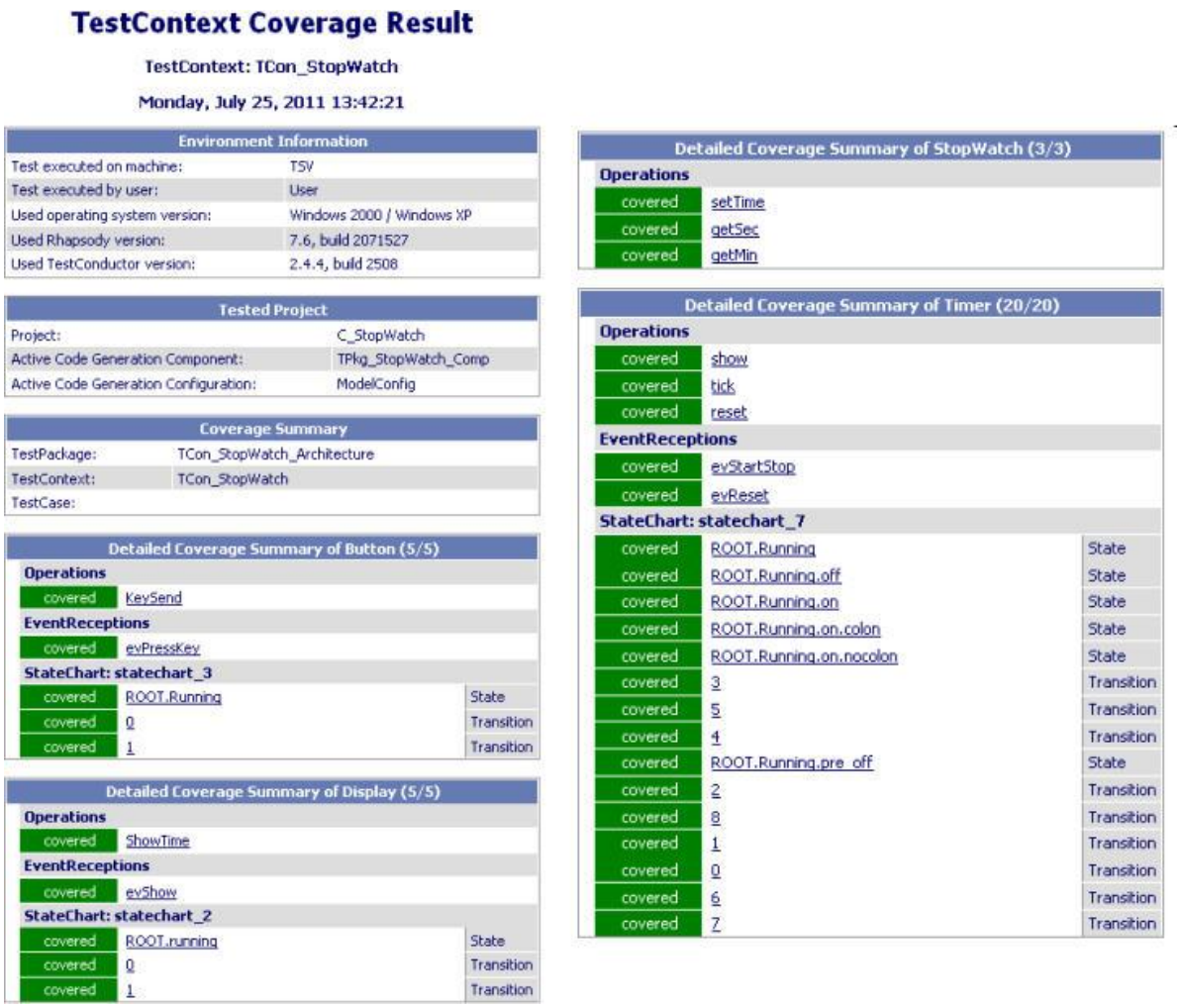



**Figure 36: Not covered transition of class StopWatch.**



**TestContext Coverage Result**

TestContext: TCon_StopWatch

Monday, July 25, 2011 13:42:21

| Environment Information | |
|---|---|
| Test executed on machine: | TSV |
| Test executed by user: | User |
| Used operating system version: | Windows 2000 / Windows XP |
| Used Rhapsody version: | 7.6, build 2071527 |
| Used TestConductor version: | 2.4.4, build 2508 |

| Tested Project | |
|---|---|
| Project: | C_StopWatch |
| Active Code Generation Component: | TPkg_StopWatch_Comp |
| Active Code Generation Configuration: | ModelConfig |

| Coverage Summary | |
|---|---|
| TestPackage: | TCon_StopWatch_Architecture |
| TestContext: | TCon_StopWatch |
| TestCase: | |

| Detailed Coverage Summary of Button (5/5) | | |
|---|---|---|
| **Operations** | | |
| covered | KeySend | |
| **EventReceptions** | | |
| covered | evPressKey | |
| **StateChart: statechart_3** | | |
| covered | ROOT.Running | State |
| covered | 0 | Transition |
| covered | 1 | Transition |

| Detailed Coverage Summary of Display (5/5) | | |
|---|---|---|
| **Operations** | | |
| covered | ShowTime | |
| **EventReceptions** | | |
| covered | evShow | |
| **StateChart: statechart_2** | | |
| covered | ROOT.running | State |
| covered | 0 | Transition |
| covered | 1 | Transition |

| Detailed Coverage Summary of StopWatch (3/3) | |
|---|---|
| **Operations** | |
| covered | setTime |
| covered | getSec |
| covered | getMin |

| Detailed Coverage Summary of Timer (20/20) | | |
|---|---|---|
| **Operations** | | |
| covered | show | |
| covered | tick | |
| covered | reset | |
| **EventReceptions** | | |
| covered | evStartStop | |
| covered | evReset | |
| **StateChart: statechart_7** | | |
| covered | ROOT.Running | State |
| covered | ROOT.Running.off | State |
| covered | ROOT.Running.on | State |
| covered | ROOT.Running.on.colon | State |
| covered | ROOT.Running.on.nocolon | State |
| covered | 3 | Transition |
| covered | 5 | Transition |
| covered | 4 | Transition |
| covered | ROOT.Running.pre_off | State |
| covered | 2 | Transition |
| covered | 8 | Transition |
| covered | 1 | Transition |
| covered | 0 | Transition |
| covered | 6 | Transition |
| covered | 7 | Transition |

**Figure 37: Full model coverage by adding additional test case.**

## 4.7.5 Verification Step 5 – Coverage of the Generated Code by Test Cases

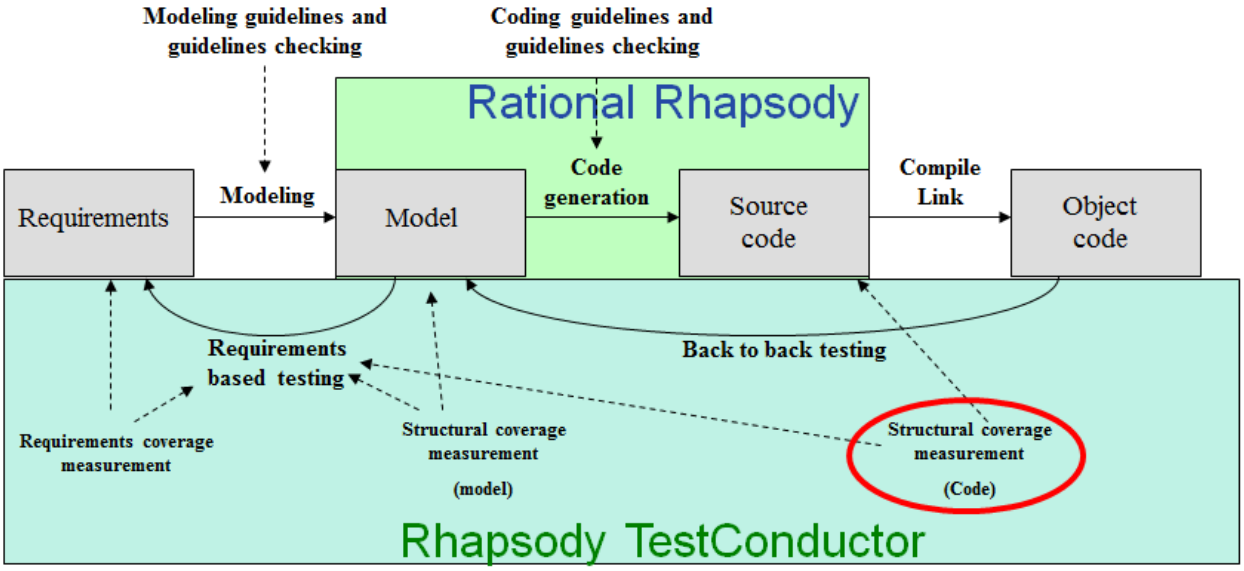



**Figure 38: Code coverage**

In the previous section we showed how IBM Rational Rhapsody TestConductor Add On can be used in order to assess the achieved model coverage by test cases. In this section we want to complement this by computing the achieved code coverage of the test cases. In order to compute code coverage it is important to define a SiL configuration for the SUT since we are only interested in the coverage of the pure SUT code. For MiL configurations, instrumented code is generated by IBM Rational Rhapsody, and the instrumented code contains a lot of additional code fragments that are only generated for simulation purposes and which are not relevant regarding code coverage. Thus, we define a new code generation configuration “HostConfig”. We define the configurations options such that SiL code is generated. Additionally, we specify that for this configuration, IBM Rational Rhapsody TestConductor Add On shall compute code coverage when test cases are executed (cf. Figure 39).


**Figure 39: Host configuration without animation code (SiL) for computing code coverage.**

After these changes are made one can compile the test cases for the configuration “HostConfig”. The computation of code coverage information is based on an source code instrumentation of the source code of the SUT, i.e., before compiling the source code of the SUT IBM Rational Rhapsody TestConductor Add On instruments the code with code fragments that performs the coverage measurement. After compilation, the test cases can be executed, and after execution a code coverage report is generated that shows the code coverage of the executed test cases (cf. Figure 40).

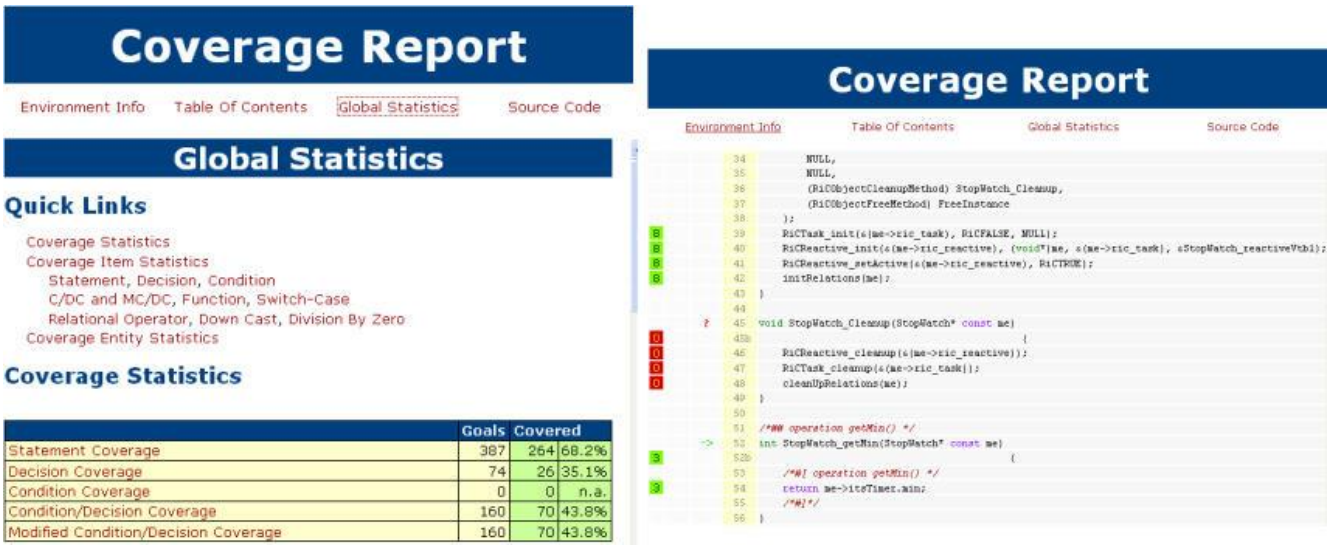



**Figure 40: The code coverage report shows the coverage achieved by the test cases.**

The code coverage report provides different views on the computed coverage information. One view focuses on statistical information like the overall statement, decision, condition, condition/decision as well as modified condition/decision coverage. Another view provides detailed coverage information for each line of the source code of the SUT. For that purpose, the source code of the SUT is highlighted with different colors that indicate to what extend a certain statement, condition or decision is executed. Additionally, for each statement or decision one can get information about which test case has participated in the coverage of the statement or decision, In order to get the needed degree of code coverage it might be needed to add more test cases that cover the parts of the code that has not been executed enough so far. The thresholds for the code coverage that needs to be achieved may differ from project to project.

### 4.7.6 Verification Step 6 – Back to Back Testing

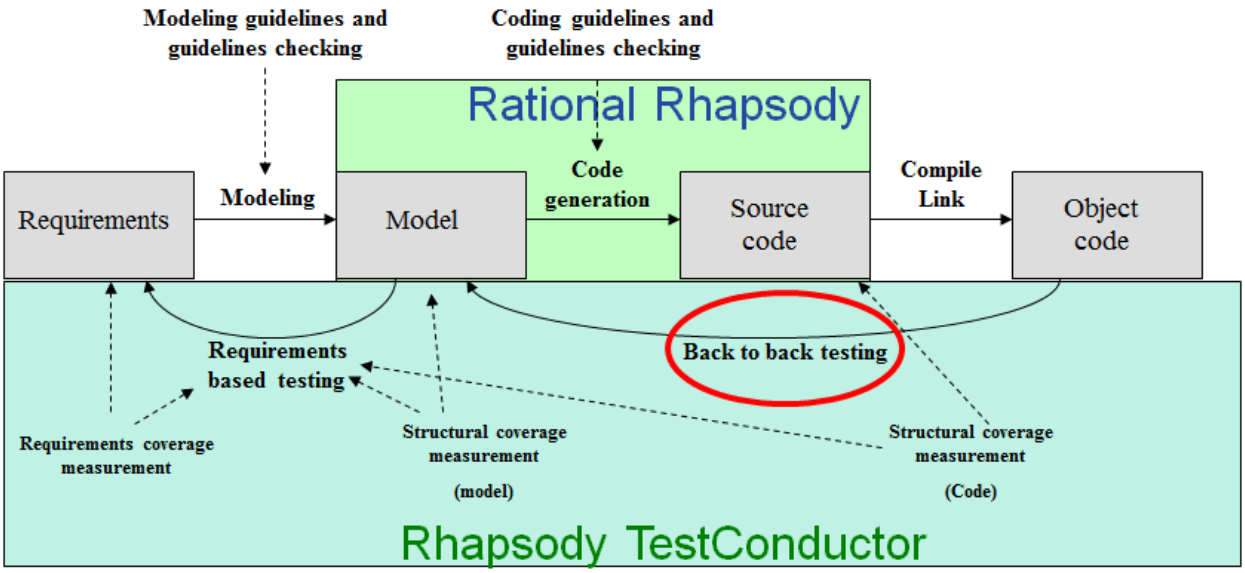



**Figure 41: Back-to-back testing**

In the previous section we showed how to get information about the code coverage that is achieved by the test cases. In this section we describe how we can make sure that the test cases evaluate to the same test result on all different execution levels MiL, SiL, and PiL. The execution of test cases on different execution levels and the comparison of the test results are called “back to back testing”. In the following, we describe how back to back testing can be performed with IBM Rational Rhapsody TestConductor Add On.

As described in sections 4.4, 4.5, and 4.6, for the different execution levels MiL, SiL, and PiL dedicated code generation configurations are created. Besides the MiL configuration “ModelConfig” and the SiL configuration “HostConfig”, we add a PiL configuration “TargetConfig” to our test model (cf. Figure 42).


**Figure 42: Configurations for MiL (ModelConfig), SiL (HostConfig", and PiL (TargetConfig) execution.**

In order to perform back to back testing, the user needs to do the following steps: first, the MiL configuration „ModelConfig" becomes the active configuration, and all test cases are executed for this configuration. The computed test report must be manually moved to a different location in the IBM Rational Rhapsody model in order to prevent that the test report is overridden by subsequent test executions with for instance SiL or PiL configurations. After that, the SiL configuration "HostConfig" shall become the active configuration, and all test cases are executed. Again, the generated test report is moved to a different location in the IBM Rational Rhapsody model for to prevent that it is overridden. Finally, the PiL configuration "TargetConfig" becomes the active configuration, and all test cases are executed again. After execution has finished, now three different test reports are stored in the model (cf. Figure 43).

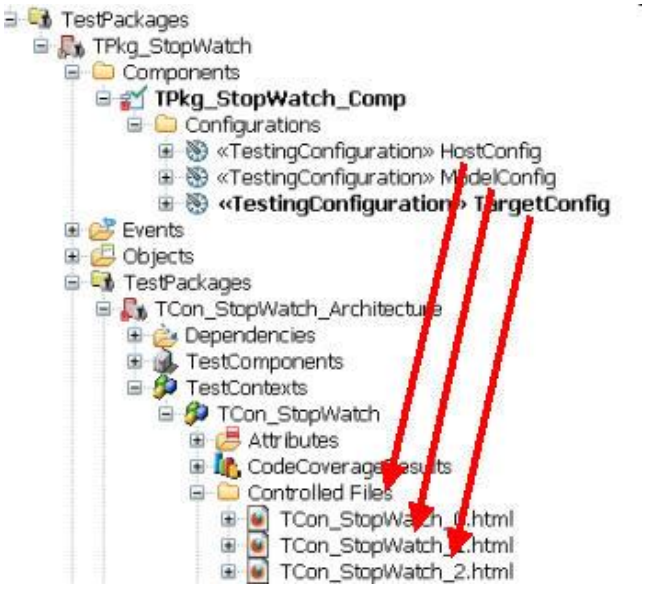
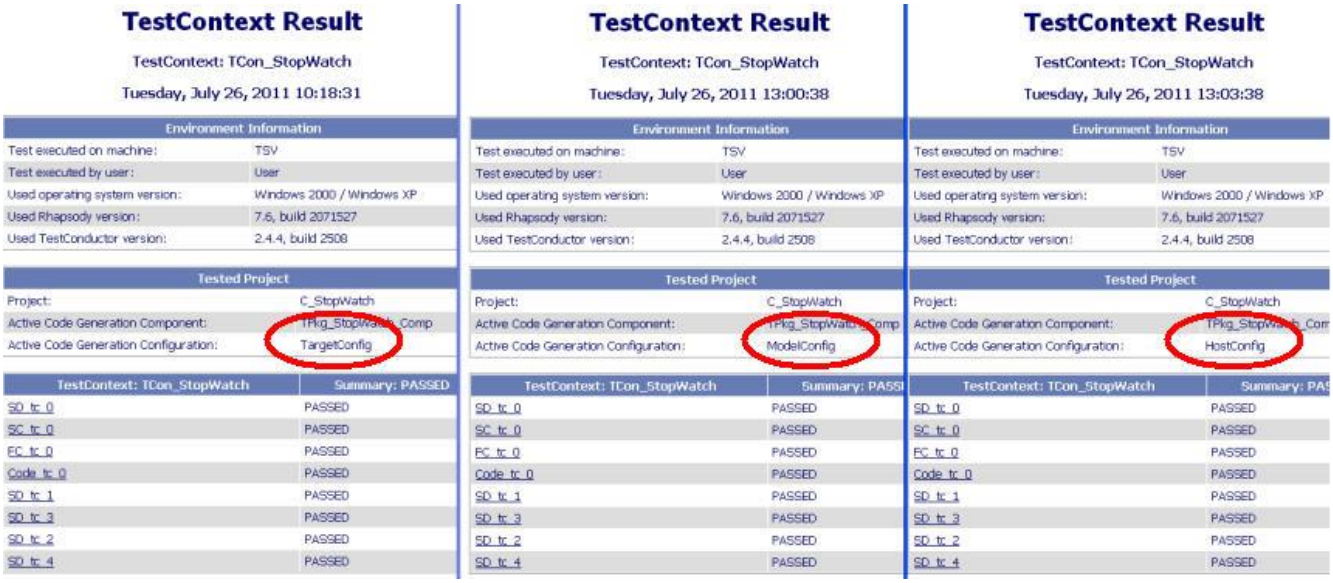



**Figure 43: Test results for MiL, SiL, and PiL execution.**

Since all test results are stored in the model, one can now compare the test results for the different execution levels. This can be done either manually by reviewing the report data, or automatically by applying a diff tool (cf. Figure 44).



**TestContext Result**

TestContext: TCon_StopWatch

Tuesday, July 26, 2011 10:18:31

| Environment Information | |
|---|---|
| Test executed on machine: | TSV |
| Test executed by user: | User |
| Used operating system version: | Windows 2000 / Windows XP |
| Used Rhapsody version: | 7.6, build 2071527 |
| Used TestConductor version: | 2.4.4, build 2508 |

| Tested Project | |
|---|---|
| Project: | C_StopWatch |
| Active Code Generation Component: | TPkg_StopWatch_Comp |
| Active Code Generation Configuration: | TargetConfig |

| TestContext: TCon_StopWatch | Summary: PASSED |
|---|---|
| SD_tc_0 | PASSED |
| SC_tc_0 | PASSED |
| FC_tc_0 | PASSED |
| Code_tc_0 | PASSED |
| SD_tc_1 | PASSED |
| SD_tc_3 | PASSED |
| SD_tc_2 | PASSED |
| SD_tc_4 | PASSED |

**TestContext Result**

TestContext: TCon_StopWatch

Tuesday, July 26, 2011 13:00:38

| Environment Information | |
|---|---|
| Test executed on machine: | TSV |
| Test executed by user: | User |
| Used operating system version: | Windows 2000 / Windows XP |
| Used Rhapsody version: | 7.6, build 2071527 |
| Used TestConductor version: | 2.4.4, build 2508 |

| Tested Project | |
|---|---|
| Project: | C_StopWatch |
| Active Code Generation Component: | TPkg_StopWatch_Comp |
| Active Code Generation Configuration: | ModelConfig |

| TestContext: TCon_StopWatch | Summary: PASS |
|---|---|
| SD_tc_0 | PASSED |
| SC_tc_0 | PASSED |
| FC_tc_0 | PASSED |
| Code_tc_0 | PASSED |
| SD_tc_1 | PASSED |
| SD_tc_3 | PASSED |
| SD_tc_2 | PASSED |
| SD_tc_4 | PASSED |

**TestContext Result**

TestContext: TCon_StopWatch

Tuesday, July 26, 2011 13:03:38

| Environment Information | |
|---|---|
| Test executed on machine: | TSV |
| Test executed by user: | User |
| Used operating system version: | Windows 2000 / Windows XP |
| Used Rhapsody version: | 7.6, build 2071527 |
| Used TestConductor version: | 2.4.4, build 2508 |

| Tested Project | |
|---|---|
| Project: | C_StopWatch |
| Active Code Generation Component: | TPkg_StopWatch_Com |
| Active Code Generation Configuration: | HostConfig |

| TestContext: TCon_StopWatch | Summary: PA |
|---|---|
| SD_tc_0 | PASSED |
| SC_tc_0 | PASSED |
| FC_tc_0 | PASSED |
| Code_tc_0 | PASSED |
| SD_tc_1 | PASSED |
| SD_tc_3 | PASSED |
| SD_tc_2 | PASSED |
| SD_tc_4 | PASSED |

**Figure 44: Comparing test results for MiL, SiL, and PiL.**

As one can see in Figure 44, in the stopwatch example the back to back test is successful, because all test results on all three execution levels MiL, SiL, PiL are the same. If one of the test results, for instance on PiL level, would differ from the test results on the other levels, one needs to analyze why the test result is different, e.g., by using a debugger for the target environment.

# Appendix A: List of Figures

# Appendix B: List of References


1. *IBM Rational Rhapsody Reference Workflow Guide*.
2. *Road Vehicles – Functional Safety, International Organization for Standardization, ISO 26262.* 2011.
3. *IBM Rational Rhapsody TestConductor Add On, [Online]* http://www-01.ibm.com/software/awdtools/IBM Rational Rhapsody/
4. *UML Testing Profile, OMG, June 2011.* [Online] http://www.omg.org/spec/UTP/1.1/PDF/.
5. *Model Driven Testing: Using the UML Testing Profile: Springer,* 2006.