



# Software test within model-based software development

---

Master's semester project – report

Hochschule Ulm

Elektrotechnik und Informationstechnik Faculty

Master Systems Engineering and Management - Electrical Engineering

Supervisor: Prof. Dr. rer. nat. Marianne von Schwerin

Isabelle Baudvin



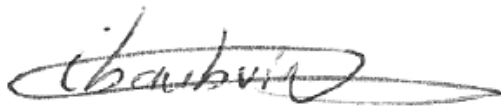
## Plagiarism Statement

I, undersigned BAUDVIN Isabelle, declares to be fully aware that the plagiarism of documents or part of a document published on all forms of support, including the Internet, constitutes copyright infringement as well as fraud Characterized.

The source of any picture, map or other illustration is also indicated, as is the source published of any material not resulting from my own experimentation or observation.

Accordingly, I undertake to cite all the sources I have used to write this report.

Signature:



**Isabelle BAUDVIN**



# Table of Contents

<b>PLAGIARISM STATEMENT .....</b>	<b>2</b>
<b>TABLE OF CONTENTS .....</b>	<b>3</b>
<b>LIST OF TABLES AND FIGURES.....</b>	<b>4</b>
<b>ACKNOWLEDGEMENTS .....</b>	<b>6</b>
<b>INTRODUCTION .....</b>	<b>7</b>
<b>PROBLEM DEFINITION .....</b>	<b>8</b>
<b>1. ABOUT TEST CONDUCTOR .....</b>	<b>9</b>
<b>2. SEVERAL POSSIBILITIES OF TEST CONDUCTOR APPLICATIONS.....</b>	<b>11</b>
3.1 COFFEE MACHINE.....	11
3.2 THE STOPWATCH APPLICATION .....	22
3.2.1 Where is defined the evShow(m=0, s=0, b=FALSE) in the Stopwatch project? .....	23
3.2.2 Where the event evShow is connected? .....	24
3.2.3 Where is defined the b parameter in evShow(m=0, s=0, b=FALSE)? .....	26
3.2.4 Why b=FALSE is correct in the evShow(m=0, s=0, b=FALSE)? .....	27
3.2.5 How this TestScenario is generated? .....	32
3.3 LINK TEST CONDUCTOR WITH RHAPSODY AND KEIL $\mu$ VISION.....	33
3.3.1 Create appropriate profiles .....	33
3.3.2 Create a new project .....	36
3.3.3 Adding Profiles .....	37
3.3.4 Project Settings, Methods and Attributes .....	38
3.3.5 Statechart .....	41
3.3.6 Instances of a Class.....	43
3.3.8 Generate / Make / Run.....	45
3.3.9 Keil $\mu$ Vision Development Environment.....	46
3.3.10 Using Test Conductor .....	50
<b>CONCLUSION .....</b>	<b>60</b>
<b>BIBLIOGRAPHY .....</b>	<b>61</b>
<b>GLOSSARY OF DEFINITIONS AND ACRONYMS USED .....</b>	<b>62</b>



## List of Tables and Figures

Figure 1 - Test Conductor main components .....	9
Figure 2 - Coffeemachine: Model1.....	12
Figure 4 - Coffeemachine: UIPanel.....	13
Figure 5 - Difference between Black Box and White Box Testing .....	14
Figure 6 - Coffeemachine: gb_observation .....	16
Figure 7 - Coffeemachine: wb_observation .....	17
Figure 8 - Coffeemachine: wb_observation_record .....	18
Figure 9 - Coffeemachine: gb_observation_record .....	19
Figure 10 - Coffeemachine: bb_spec.....	20
Figure 11 - Coffeemachine: The path of the folder TestSUTS .....	21
Figure 12 - Stopwatch: Overview .....	23
Figure 13 - Stopwatch: Statechart of Display .....	24
Figure 14 - Stopwatch: Features for the transition ShowTime.....	24
Figure 15 - Stopwatch: Operations of Display .....	25
Figure 16 - Stopwatch: Implementation of ShowTime in Display .....	25
Figure 17 - Stopwatch: Operations in Timer.....	26
Figure 18 - Stopwatch: Arguments in show.....	26
Figure 19 - Stopwatch: Statechart of class Timer with b=FALSE .....	27
Figure 20 - Stopwatch: Statechart of class Timer with b=TRUE .....	28
Figure 21 - Stopwatch: SDTestScenario.....	28
Figure 22 - Stopwatch: Execute TestCase.....	29
Figure 23 - Stopwatch: Results of TestCase.....	29
Figure 24 - Stopwatch: Statechart Test Case.....	30
Figure 25 - Stopwatch: Animated SDTestScenario with b=TRUE.....	31
Figure 26 - Stopwatch: Animated SDTestScenario with b=FALSE.....	31
Figure 27- Stopwatch: Statechart of class Timer .....	32
Figure 28 - The installation of Willert RXF.....	33
Figure 29 - Select path of Willert Software Tools Program.....	34
Figure 30 - Select path of Keil main directory .....	34
Figure 31 - Select the path of Rhapsody data folder.....	35
Figure 32 - Select the path of Rhapsody program files folder.....	35
Figure 33 - Create a new project.....	36
Figure 34 - Name and location for the new project .....	36
Figure 35 - Create the directory .....	36
Figure 36 - Rename in the model browser .....	37
Figure 37 - Location for the profiles.....	37
Figure 38 - Enter the stereotype and Standard Headers .....	38
Figure 39 - Class LED and the model browser .....	39
Figure 40 - Operations in the class LED.....	39
Figure 41 – Attributes of the class LED .....	39
Figure 42 - Two arguments of Init.....	40
Figure 43 - Implementation Init in LED .....	40
Figure 44 - Implementation off in LED .....	40
Figure 45 - Implementation on in LED.....	40



Figure 46 - Add new statechart on the LED.....	41
Figure 47 - Build the statechart.....	41
Figure 48 - Action on entry 'off' .....	42
Figure 49 - Action on entry 'on'.....	42
Figure 50 - Add a trigger.....	42
Figure 51 - Final statechart .....	43
Figure 52 - Add new OMD called 'Runtime' .....	43
Figure 53 - Drag 'LED' class in Runtime OMD .....	43
Figure 54 - Make an object .....	44
Figure 55 - Initialization of itsLED .....	44
Figure 56 - First set value in Init .....	44
Figure 57 - Second set value in Init .....	45
Figure 58 - Create the debug directory .....	45
Figure 59 - The deployer configuration.....	46
Figure 60 - Open Keil $\mu$ Vision .....	46
Figure 61 - Keil $\mu$ Vision environment .....	47
Figure 62 - Switch Target from Simulator to Hardware .....	47
Figure 63 - 'Embedded_Uml_Target_Debugger.bat' .....	48
Figure 64 - Configure communication plugin.....	48
Figure 65 - $\mu$ Vision Settings.....	49
Figure 66 - Configuration in Keil $\mu$ Vision.....	49
Figure 67 - Keil Message evaluation mode .....	50
Figure 68 - Create Test Architecture on LED .....	50
Figure 69 - Add TestingProfiles .....	51
Figure 70 - Set the stereotype in 'TPkg_LED_Comp' .....	51
Figure 71 - Set the stereotype in 'TestingConfiguration DefaultConfig' .....	51
Figure 72 - Set the settings in 'TestingConfiguration DefaultConfig' .....	52
Figure 73 - Set the tags in 'TestingConfiguration DefaultConfig' .....	53
Figure 74 - Add arguments in itsLED .....	53
Figure 75 - Code test case in the Model Browser .....	54
Figure 76 - Initialization code in 'DefaultConfig' .....	54
Figure 77 - Update TestContext .....	55
Figure 78 - Build TestContext .....	55
Figure 79 - Location of html result of TestCases .....	55
Figure 80 - Result of Code Test Case .....	56
Figure 81 - Statechart for SD test case .....	56
Figure 82 - SD Test Scenario.....	57
Figure 83 - Message to update and built before execute test case .....	58
Figure 84 - SD test case passed .....	58
Figure 85 - Show as SD .....	58
Figure 86 - Verification when Show as SD.....	58
Figure 87 - Result of SD Test Case .....	59



## Acknowledgements

Before presenting this project in my first semester of my Master at the Hochschule Ulm, it seems me important to thank all the people who have contributed to the smooth running of my project.

A special gratitude I give to my professor Dr. rer. nat. Marianne von Schwerin, whose contribution in stimulating suggestions and encouragement, helped me to coordinate my project. And for accepting that I can make this report in English.

Furthermore, I would also like to acknowledge Dr. rer. nat. Marianne von Schwerin for her permission to use all the necessary equipments and materials from the Hochschule Ulm to carry out my project.

I would also like to thank Mr. Fuchs for guiding me in the use of the material in the early days.

A special thanks goes to Prof. Dr. rer. nat. Bank, who accept me in this Master in double degree in Germany.

Finally, I would like to express my deepest appreciation to all those who provided me the possibility to do this Master.



## Introduction

Four steps exist when you are in a project: Requirements, Design, Implementation and Test. However, more you wait in the process to test your project, more the project's final cost is high. That why, all projects have to be tested. IBM® Rhapsody® Test Conductor Software allow to have a quality design. With this software, engineers can locate errors early in the process and automate tedious testing.



# 1. Problem definition

This project involves to get familiar with the software Test Conductor and to make an example with these functionalities. Test Conductor adds model based testing (MBT) capabilities to IBM® Rhapsody®. In more, Test Conductor is based on Object Management Group (OMG) standard UML Testing Profile.

During this project, we will discover a new software that is IBM® Rational® Rhapsody®, the version 8.1.5 - 64bit is used. With IBM® Rational® Rhapsody®, we can create UML Diagrams and generate ANSI 'C' -code from them. In more, the software Keil MDK ARM is needed, the Keil µVision IDE to flash our executable model to our target. Finally, to work these two software together, we need the Embedded UML RXF™ from Willert Software Tools (WST) which is an interface between a UML model and the target platform in order to combines Rhapsody with the Keil µVision IDE.

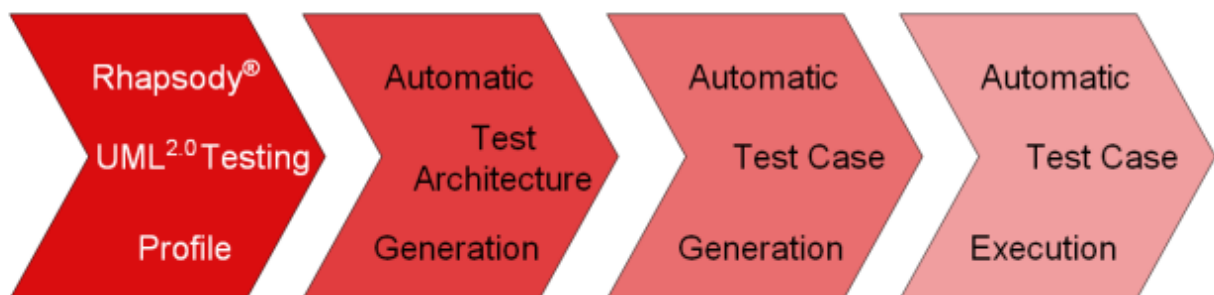
This document describes the process of the project.





## 2. About Test Conductor

« Test Conductor is part of the Rhapsody Testing Environment which is based on three main components: “Automatic Test Architecture Generation”, “Automatic Test Case Execution” and “Automatic Test Case Generation”. These three components are developed along the UML Testing Profile as implemented in Rhapsody.



*Figure 1 - Test Conductor main components*

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

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

- Sequence diagrams
- Statechart (only Rhapsody in C/C++/Java/Ada)
- Flow charts (only Rhapsody in C/C++)
- Pure code (only Rhapsody in C/C++/Java/Ada)

Test Conductor is a model based testing environment used to debug and test object-oriented embedded software designed in Rhapsody. Test Conductor supports unit testing as well as software integration testing based on graphical test definitions using sequence diagrams. In Rhapsody in C++, Rhapsody in C, Rhapsody in Java, and Rhapsody in Ada test cases can be defined also by statechart, flow charts (only C/C++), or pure code. Using sequence diagram related test cases, Test Conductor supports an advanced graphical failure analysis. These features make it easy to define and execute extensive test suites, as well as to create complex tests drivers and test monitors. » [1]



With all these functionalities, using the IBM Rational Rhapsody Test Conductor Add On solution, engineers and developers can create unit test cases graphically using UML sequence diagrams, state charts, activity diagrams or flowcharts. They can also develop test cases in code. Not only can graphical test cases help customers and project stakeholders better understand code tests, they can also help communicate intended behavior more effectively.



## 3. Several possibilities of Test Conductor applications

### 3.1 CoffeeMachine

We will explain the example done in the Webinar by BTC Embedded Systems AG, a IBM business partner the November 08, 2016.

IBM® Rational® Test Conductor Add-on is the add-on package that allows you to add model-based software or system development to model-based testing.

« The UML test profiles defined by the Object Management Group (OMG) provide the necessary modeling resources for the test.

Creation of the test architecture and execution of the tests in the model are largely automated. Test cases are defined by sequence diagrams (SD), state automates or flow charts. In more, tests can run individually or in a network, manually or automatically. Dependencies between test results and test requirements simplify proof of the test cover of the model. » [2]

The Webinar is a demonstration of an example in order to see the functionalities of the software Test Conductor. In this project, we use IBM® Rational® Rhapsody® Developer for C++, the version 8.1.5, 64bit.

The Coffeemachine consists of:

- **Controller**: contains the statechart who control everything. It allow to link all differentes parties.
- **CoffeeBeans**: a store for beans with a sensor sensing supply of beans.
- **WaterTank**: tank with sensor sensing filling level.
- **Grinder**: crushing the beans on demand.
- **BrewingUnit**: heating water and brewing coffee. Part of the BrewingUnit is also a reservoir with sensor collecting used coffee grounds.

We follow the enhancement in Grey Box Testing which will be demonstrated using a composite Rhapsody in C model of a Coffeemachine.



The Model 1 is located in the folder 'Packages' → 'Default' → 'Object Model Diagrams'. The Model 1 is a diagram where we can see all dependencies between all objects in the Coffeemachine project.

In this Model1, the Controller reacts on events and invokes operations in other parts of Coffeemachine. We can see this in the second part of the object **itsControl:Control** from where objects are inherited from the Controller object.

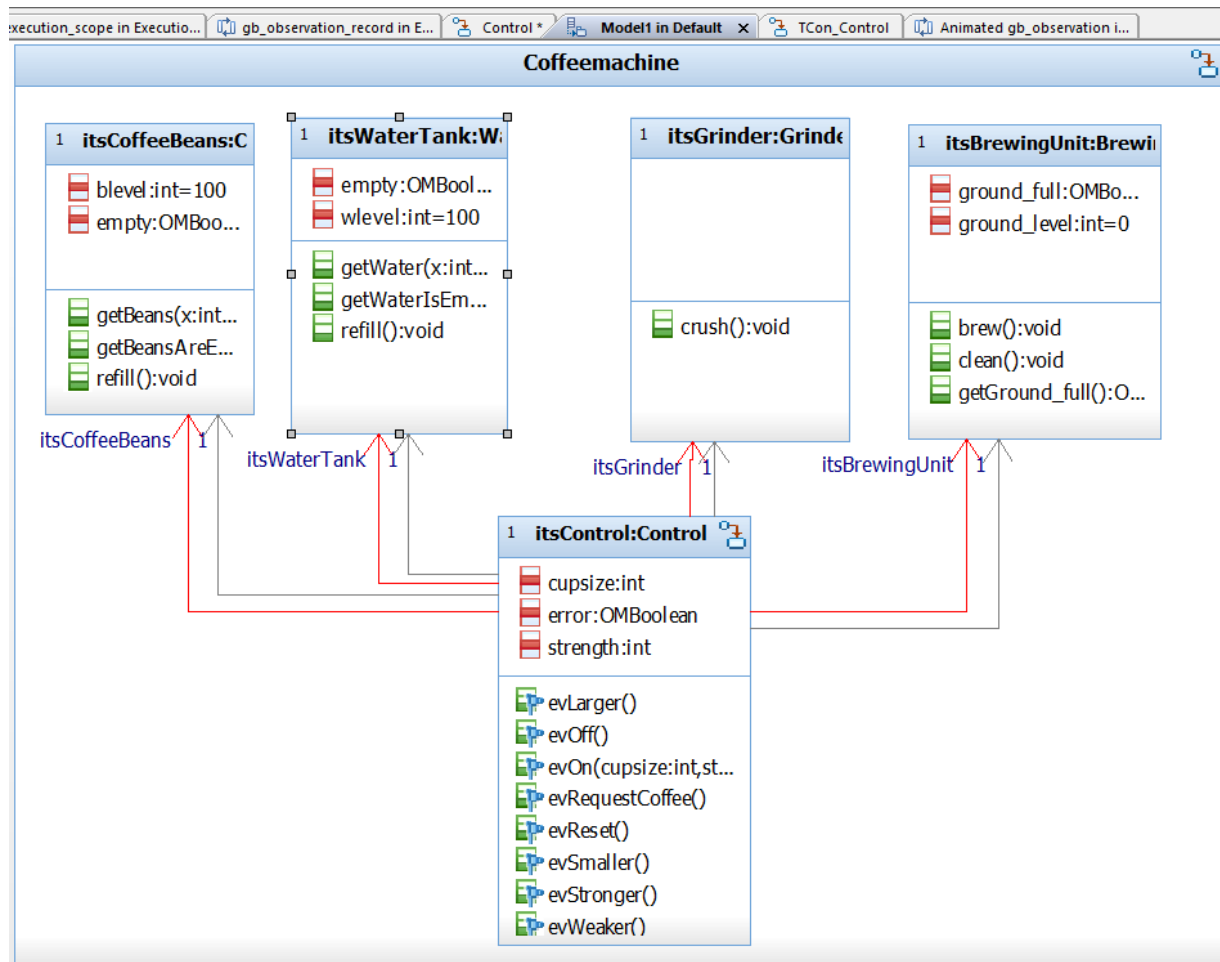


Figure 2 - Coffeemachine: Model1

Here, we can recognize the different classes that the Coffeemachine consists of.

Then, in **Packages** → **ExecutionScope** → **Object Model Diagrams**, we can find the diagram called 'execution\_scope'.

Actor User Interface (UI) is used to model user interactions with the Coffee machine on user interface level.



In more, in **Packages** → **ExecutionScope** → **Panel Diagrams**, we can discover the file named 'UIPanel'.

The UIPanel looks like the figure below.

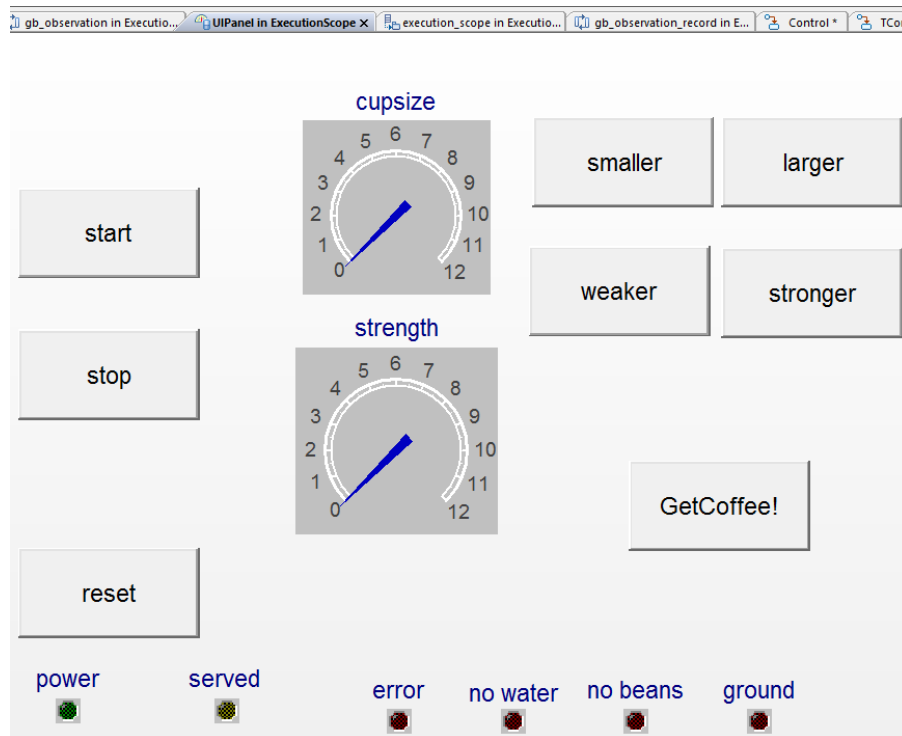


Figure 3 - Coffeemachine: UIPanel

We can identify the buttons to start and stop the Coffeemachine. Underneath it, the 'reset' button is to refill beans, water and the ground reservoir when is empty.

At the bottom right of the figure, we observe many LEDs. The first one entitled 'power' is to indicate if the system is being started. In the same way the button labeled 'served' is to designate if the coffee is being served. Finally, the last four LEDs are error indications.

The gauge called 'Cup Size' with the two buttons at his right are here to determine and display the cup-size. The buttons above work on the same principle that means to determine and display strength of the coffee.

Thus the user can choose to have a coffee smaller or larger and also weaker or stronger. At the end, the user can push the button 'GetCoffee!' to request his coffee.



The next step to understand is for testing. In the next folder named 'Sequence Diagrams', we will:

- 1) Record the behavior as a grey box sequence diagram
- 2) Record the behavior as a white box sequence diagram
- 3) Manually specify the black box behavior

First, we have to know what is a Black Box (BB), a White Box (WB) and a Grey Box (GB). To understand better, let's focus on the difference between the Black and White Box testing.

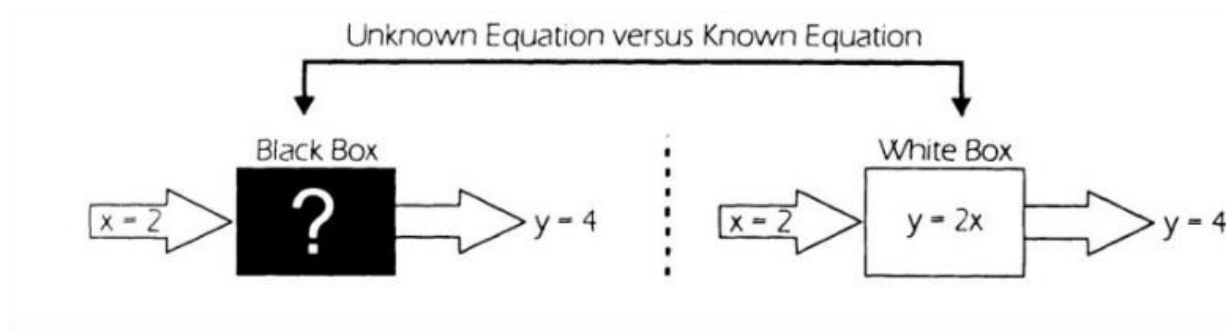


Figure 4 - Difference between Black Box and White Box Testing ([http://images.slideplayer.com/23/6665521/slides/slide\\_22.jpg](http://images.slideplayer.com/23/6665521/slides/slide_22.jpg))

**The Black Box Testing:** «is a software testing method in which the internal structure / design/ implementation of the item being tested is not known to the tester. These tests can be functional or non-functional, though usually functional. This method is named so because the software program, in the eyes of the tester, is like a black box, inside which one cannot see. This method attempts to find errors in the following categories:

- Incorrect or missing functions
- Interface errors
- Errors in data structures or external database access
- Behavior or performance errors
- Initialization and termination errors

#### **Black Box Testing ADVANTAGES:**

- Tests are done from a user's point of view and will help in exposing differences in the specifications.
- Tester don't need to know programming languages or how the software has been implemented.
- Tests can be conducted by a person independent from the developers, allowing for an objective perspective and keeping away from developer-bias.
- Test cases can be designed as soon as the specifications are complete.



### **Black Box Testing DISADVANTAGES:**

- Only a small number of possible inputs can be tested and many programs paths will be left untested.
  - Without clear specifications, which is the situation in many projects, test cases will be difficult to design.
  - Tests can be redundant if the software designer/ developer has already run a test case.
- **The White Box Testing:** is the opposite of the Black Box. It is also a software testing method in which the internal structure/design/implementation of the item being tested is known to the tester. The tester chooses inputs to exercise paths through the code and determines the appropriate outputs. Programming know-how and the implementation knowledge is essential. This method is named so because the software program, in the eyes of the tester, is like a white/ transparent box, inside which one clearly sees.

However, it is mainly applied to Unit Testing. The White Box Testing method is applicable to the following levels of software testing:

- **Unit Testing:** For testing paths within a unit.
- **Integration Testing:** For testing paths between units.
- **System Testing:** For testing paths between subsystems.

### **White Box Testing ADVANTAGES:**

- Test Testing can be initiate at an earlier stage. One don't need to wait for the GUI to be available.
- Testing is more thorough, with the possibility of covering most paths.
- The tester can find bugs earlier that are hard to find with Black Box Testing.

### **White Box Testing DISADVANTAGES:**

- Tests can be very complex, highly skilled resources are required, with thorough knowledge of programming and implementation.
- Test script maintenance can be a handicap if the implementation changes too frequently.
- Since this method of testing it closely tied with the application being testing, tools to gratify to every kind of implementation/platform may not be quickly available.



- **The Grey Box Testing:** is a software testing method which is a combination of Black Box Testing method and White Box Testing method, which means the internal structure is partially known. This involves having access to internal data structures and algorithms for purposes of designing the test cases, but testing at the user, or at black-box level. Gray Box Testing is named so because the software program, in the eyes of the tester is like a gray/ semi-transparent box, inside which one can partially see. Though Gray Box Testing method may be used in other levels of testing, it is primarily useful in Integration Testing. » [4]

In this project, we do a grey box testing.

Now, we can go to the folder named 'Sequence Diagrams'. In the file called 'gb\_observation'. So like I explain before, 'gb' concern the Grey Box Testing.

For recording a Grey Box sequence diagram, we use a sequence diagram with instance lines for actor UI and for the entire system. The class CoffeeMachine is on the right side but the UI correspond to the actor.

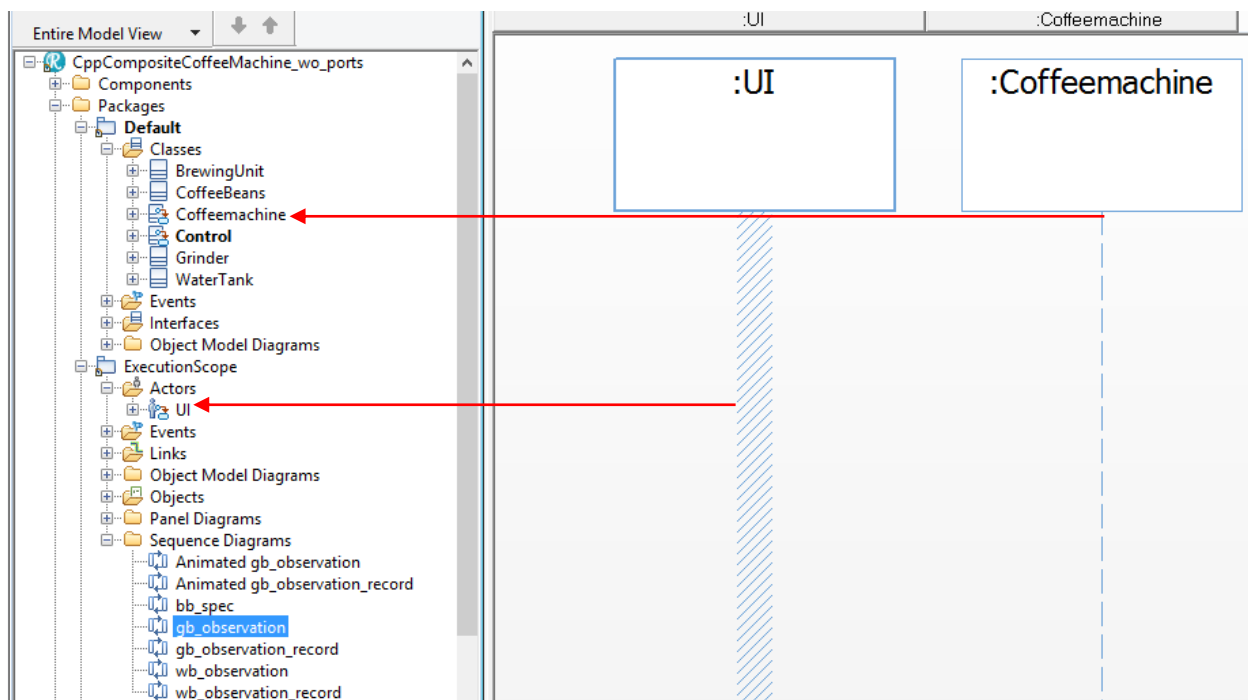


Figure 5 - Coffeemachine: gb\_observation

If we click right on the Coffeemachine, we can access to the 'Features' and after go to the 'Properties'. We can see Animations which will map all messages from and to Coffeemachine as well as all messages among parts of Coffeemachine to instance line ':Coffeemachine'.





For the White Box Testing, described as 'wb\_observation', we will record a sequence diagram from animated execution where each part of Coffeemachine is represented in the 'Packages' folder → 'Default' folder → 'Classes' folder, like we saw before in the 'gb\_observation' example.

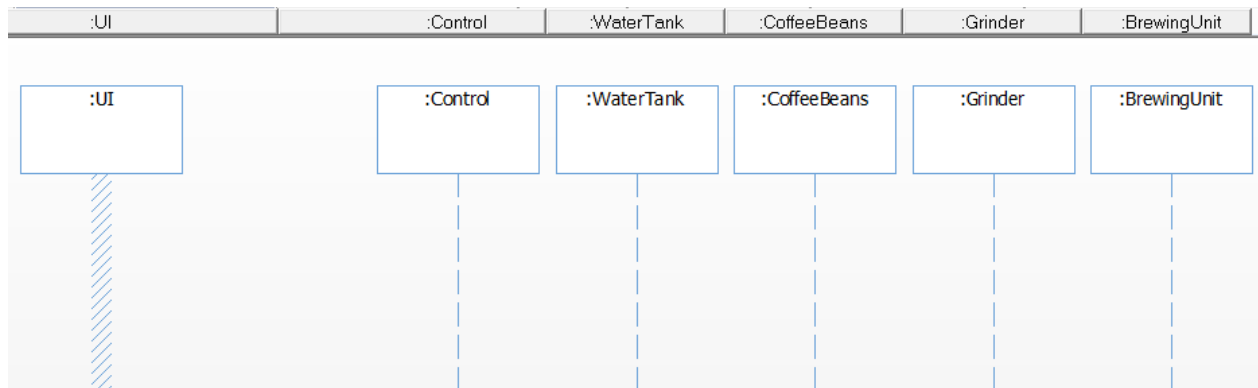


Figure 6 - Coffeemachine: wb\_observation

For recording Grey Box and White Box scenario in one sweep, we invoke Generate/Make/Run to run the model interactively with panel by clicking on this icon:



After, we go back to the UIPanel explained in Figure 4.

It can propose you to create a new directory for the executable or to create a new executable on the one already created before.

Following the succeeded animated execution, you can on the icon 'Go' (or F4).

Now the UIPanel is in mode animation. At this moment the LED labelled 'power' is not yet started. By clicking on the button 'Start', it will send an event named 'evOn(cupsize, strength)' to the controller via delegation port. Thereby, we can see the LED became in a green color, that's means the Coffee machine is activated. In the same way, by clicking on the button 'GetCoffee!' the LED named 'served' changed into yellow.

At this time we can click on the icon pause in orange to stop the animation. We have enough to see what happened. Afterwards have exited the 'Animation Session' mode, we have a look on the recorded White Box animation sequence diagram as we can see in the Figure 8 below.

First, this sequence diagram named automatically 'Animated\_wb\_observation\_record'.

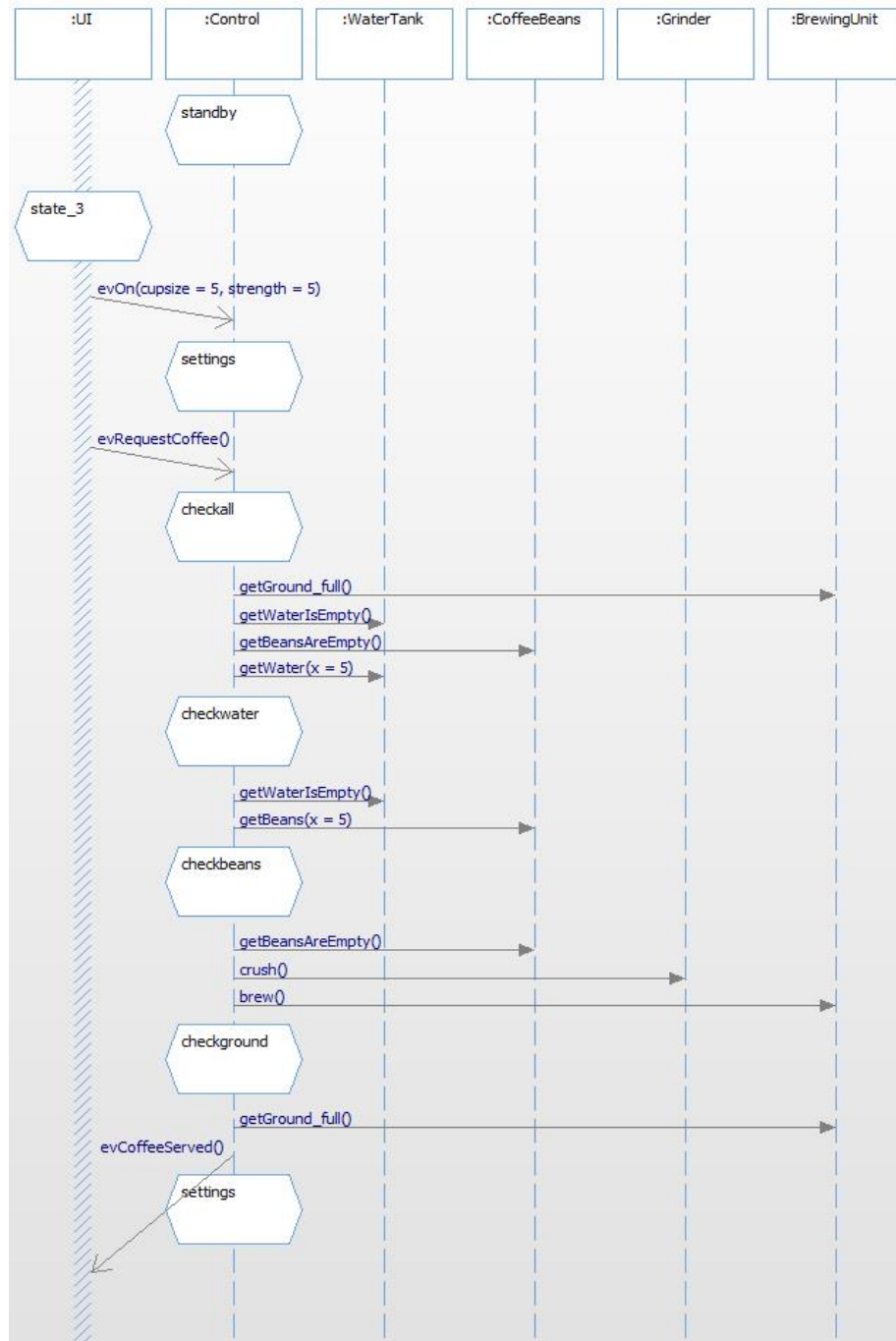


Figure 7 - Coffeemachine: wb\_observation\_record

Now, we can inspect recorded Grey Box animation sequence diagram.

Messages among Coffeemachine's parts (Control, WaterTank, CoffeeBeans, Grinder and BrewingUnit) were recorded as self-messages of Coffeemachine. Note, that operation-names were recorded with class-prefix automatically (Rhapsody in C).



However, we can to eliminate the class-prefixes automatically generated which corresponding with the White Box's class where they are from. The figure below is showing after eliminate this.



Figure 8 - Coffeemachine: gb\_observation\_record



Now we manually create a Black Box specification with copy by drag and drop. And we can create three messages like it is show in the figure 10 below.

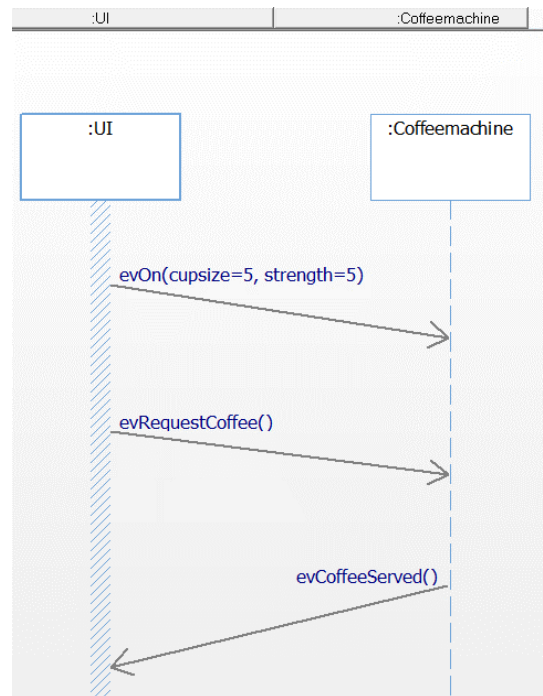


Figure 9 - Coffeemachine: bb\_spec

As a consequence, we are ready to create a Grey Box TestArchitecture for Coffeemachine and to create TestCases from the sequence diagrams generated so far. Therefore, we will have to change Test Conductor settings.

So go in 'Tools' → and in 'Test Conductor', we have to select 'GreyBox' in the setting 'Create TestArchitecture Transparency'. In that way, it lets Test Conductor create the TestArchitecture with a clone of the SUT (and clone of its part-classes). These clones will be instrumented for testing purposes without affecting the original model elements in the model.

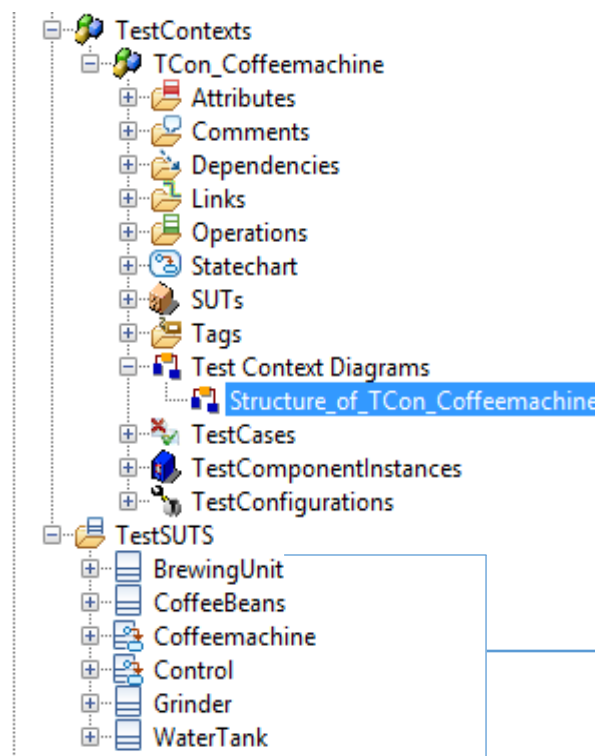
**System Under Test (SUT):** instance of the class to be tested. Classes to be tested are not modified by testing in any means, they are instantiated as is for testing purposes. The test architecture provides testing artefacts for stimulating, observing and assessing reactions of the SUT.

In the same way, we have to select 'Weak' in the setting 'MapSDToTestArchitectureMode'. With this, we can allow to map sequence diagrams to a TestArchitecture that refers to 'compatible' classes (in terms of their input/output messages) or to TestArchitecture that refer to e.g. classes that contain the instance line classifiers as parts. Otherwise, the setting 'Strict' allow only mapping sequence diagrams to TestArchitectures that refer to at least one instance line classifier as SUT.



Now, we create a Grey Box TestArchitecture for class Coffeemachine by clicking right on the 'Coffeemachine' in the Packages → Default → Classes. And select 'Create TestArchitecture'.

In TestPackages → TPkg\_Coffeemachine\_GB → TestPackages → TCon\_Coffeemachine\_Architecture → TextContexts → TCon\_Coffeemachine → Test Context Diagrams → Structure\_of\_TCon\_Coffeemachine.



Local copies (clones) of Coffeemachine and classes of Coffeemachine's parts. This copies will replace original classes in code generation scope and will be instrumented for testing purposes.

Figure 10 - Coffeemachine: The path of the folder TestSUTs

Now we create TestCases from the sequence diagrams. First BlackBox, then GreyBox and finally WhiteBox.

And now we can build and execute TCon\_Coffeemachine by click right on this.

To conclude this example, that demonstrated on a Rhapsody in C model how one single Grey Box TestArchitecture can be used to test with BB, GB and WB specifications.



## 3.2 The Stopwatch application

The tutorial with the project provided by the Rhapsody's installation can be used to better understand how work Test Conductor on a project.

The Stopwatch application is a C example. «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.

The Stopwatch model, show in figure 12, 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.  
» [7]

In this stop watch project, we will open the project „CStopWatch“ from the folder „Samples/CSamples/TestConductor“ in the Rhapsody installation. Test Conductor is characterized by the execution of test cases. We can have different kinds of test cases like sequence diagram, flow chart, statechart or source code.

After have created a test case sequence diagram manually and drew the message evShow from the SUT to the test component “TCon\_StopWatch.itsTC\_at\_pOut\_of\_StopWatch”, we can specify arguments values like “m = 0, s = 0, b = FALSE” for the message. Thus, the test execution passed correctly.

The following questions are about where in the project the parameters are defined in order to understand why the tests are passed.

In this project, we use IBM® Rational® Rhapsody® Developer for C, the version 8.1.5, 64bit.

After following the tutorial, we advanced step by step to understand all the functionalities of Test Conductor.



### 3.2.1 Where is defined the evShow(m=0, s=0, b=FALSE) in the Stopwatch project?

To find where evShow is defined, we have to go in Packages → StopwatchPkg → Object Model Diagrams → Stopwatch Overview:

Then we can find evShow in itsDisplay:Display.

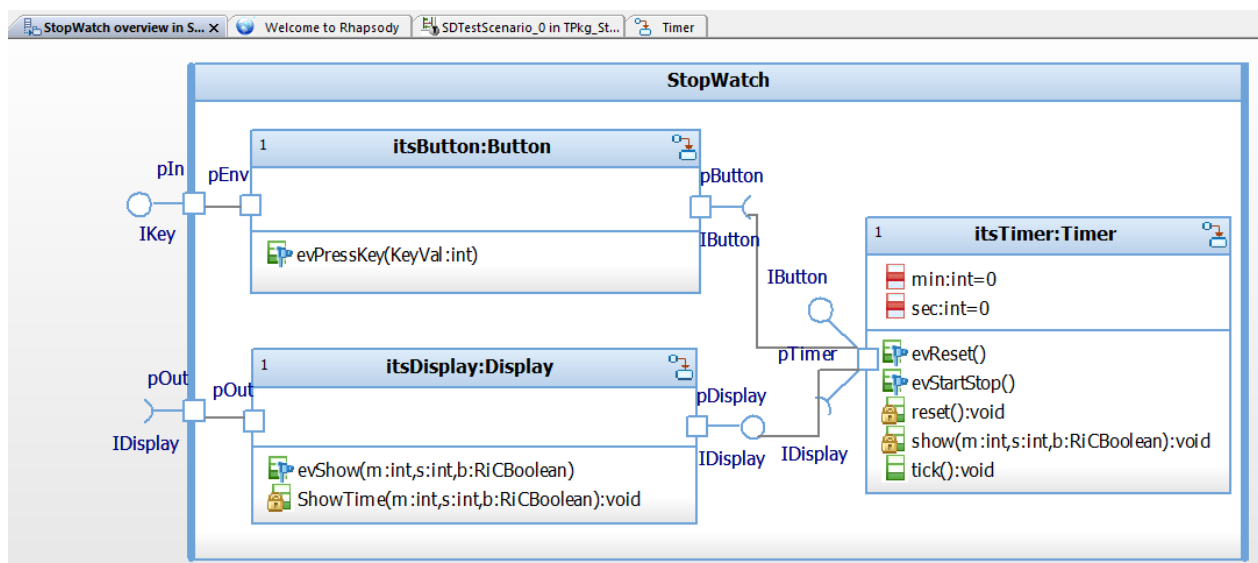


Figure 11 - Stopwatch: Overview



### 3.2.2 Where the event evShow is connected?

The event evShow is sent to port pTimer of class StopWatchPkg::Timer in its private operation show() like show the Figure 12. In more, show() is invoked on entering states colon and no colon (name of the sub-states in the Statechart of class Timer). Port pTimer of StopWatchPkg::Timer is connected to port pDisplay of StopWatchPkg::Display (Timer and Display are instantiated as parts in Stopwatch).

Display sends evShow (on reception of evShow in statechart action = with 'on reception of evShow in statechart action') to port pOut via delegation port pOut of StopWatchPkg::Stopwatch. Since TestComponent TC\_at\_pOut\_of\_StopWatch is connected to pOut by the TestArchitecture (link in TestContext TCon\_StopWatch) evShow is sent to this TestComponent.

The statechart of Display is in Statechart Action in Packages → StopWatchPkg → Classes → StatechartDiagram → StatechartDiagram:

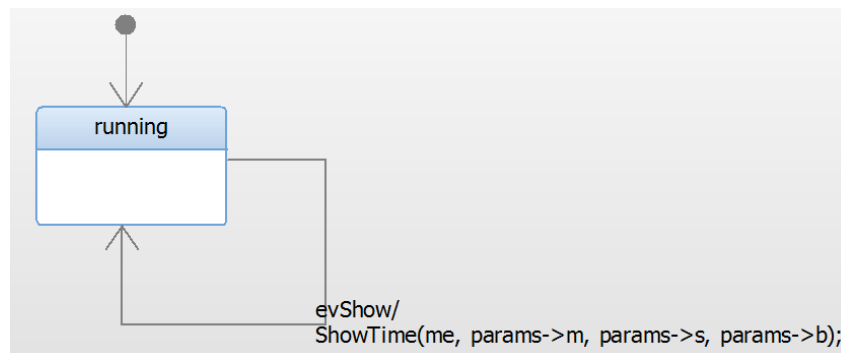


Figure 12 - Stopwatch: Statechart of Display

In particular: when Display receives evShow, it takes the transition. Taking that transition involves calling ShowTime - which is the action part of the transition (open the features dialog for the transition):

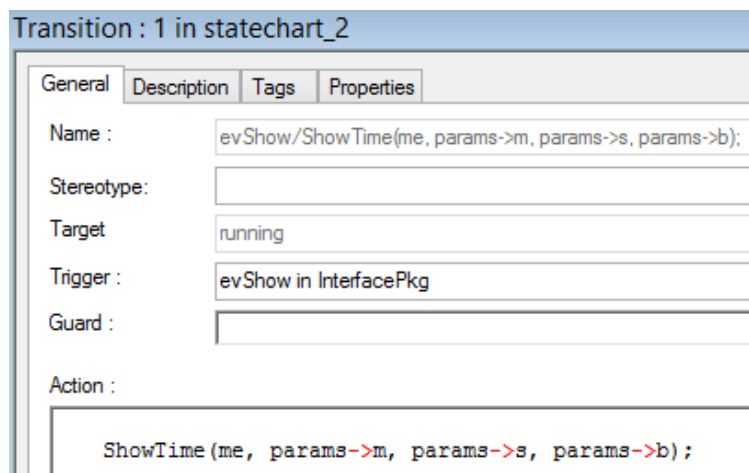


Figure 13 - Stopwatch: Features for the transition ShowTime





ShowTime is a private operation of Display - if we look into its implementation, we will see that the operation send evShow to out port **pOut** of Display. In StopWatch Overview → itsDisplay → in Operations:



Figure 14 - StopWatch: Operations of Display

Public operation can be invoked by every other class that knows the owner of the operation, whereas private operations can only be invoked by the owner itself.

Technically, this is achieved in Rhapsody in C by defining such operations file-static, i.e. they are invisible from outside the file containing the declarations and definitions of a particular class. By making operations private, one can prevent them from being invoked from anywhere else, except from invocation by the owner.

Restricting operation (and also attribute) visibility can improve code structuring, help avoid undesired side-effects and hence contribute to safety aspects of an application. In our case, the developer of the stop watch model wanted to make sure that ShowTime() will always only be invoked by Display and by no other class of the model.

In the implementation of ShowTime in private, we have:

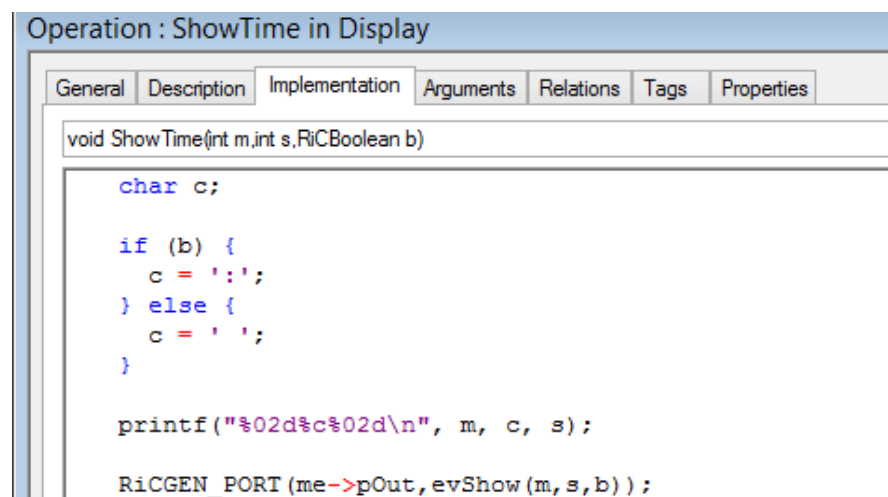


Figure 15 - StopWatch: Implementation of ShowTime in Display



Here, we can see that into the implementation of ShowTime (private operation of Display). On one hand, if *b* have the value of 'true', it's means that the stop watch show the colon. For example: **12:45**. In another hand, if *b* have the value of 'false', it's means that the colon don't appear in the stop watch.

Then, we can understand in the **printf** the meaning of:

- **%d** refers to an integer and **%c** refers to a character.
- **%02d** it means output of integer values in 2 or more digits, the first being zero if number less than or equal to 9.

In more, we can see that the operation send *evShow* to out port **pOut** of the class Display.

### 3.2.3 Where is defined the *b* parameter in *evShow(m=0, s=0, b=FALSE)*?

To find where the parameter *b* is defined, we have to go in StopWatch Overview → *itsTimer:Timer* → in Operations:

Object : itsTimer in StopWatch






General	Description	Attributes	Operations	Ports	Flow Ports	Relations	Tags	Properties
<input type="checkbox"/> Show Inherited								
Name	Visibility	Return Type						
 show	Private	void						
 tick	Public	void						
 reset	Private	void						
 evStartStop	Public							
 evReset	Public							

Figure 16 - StopWatch: Operations in Timer

Then in show → in Arguments:

Operation : show in Timer

GeneralDescriptionImplementationArgumentsRelationsTagsProperties

void show(int m,int s,RiCBoolean b)

Name	Type	Value	direction
m	int		In
s	int		In
b	RiCBoolean		In

Figure 17 - StopWatch: Arguments in show



### 3.2.4 Why $b=FALSE$ is correct in the `evShow(m=0, s=0, b=FALSE)`?

Looking into the statechart of the class `Timer` helps answering it.

Statechart is entered when activating the statechart behavior - via the default transition, state **pre\_off** is entered automatically. After a timeout of 500 ms, state **pre\_off** is exited and state **off** is entered. State **off** has an 'Action on entry': **show**(me, me->min, me->sec, FALSE); **show** is an operation of `Timer`, sending `evShow` with the arguments specified in the action on entry. Here FALSE is used as 4th argument in the state **nocolon**, i.e. the argument  $b$  in `evShow`.

We can find this statechart in Packages → `StopWatchPkg` → Classes → `Timer` → Statechart → `StatechartDiagram`:

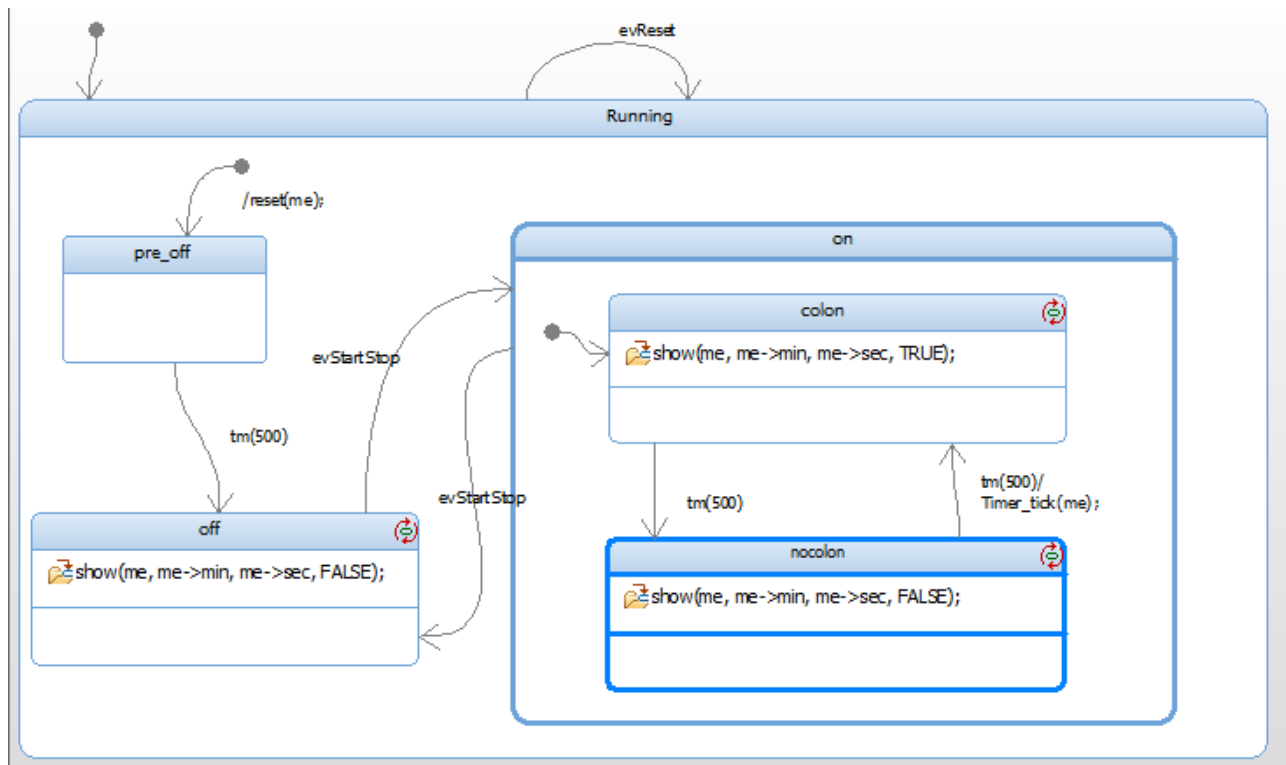


Figure 18 - StopWatch: Statechart of class `Timer` with  $b=FALSE$

Because at the beginning,  $b$  have to be = FALSE like this statechart show. The parameter  $b$  can't initiate as =TRUE. Otherwise, the statechart can't continue to go to the state ON at the right.

To verify this assumption.

We have changed the  $b$  parameter in the state OFF, like bellow in the Statechart of class `Timer`:

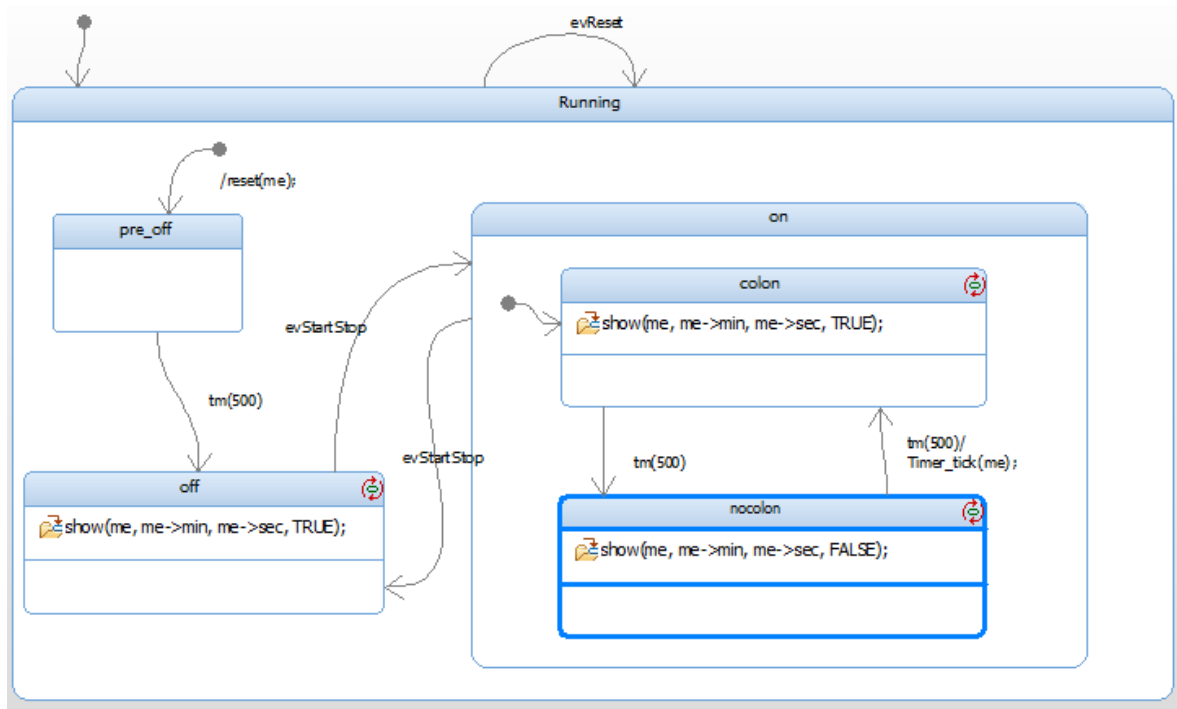


Figure 19 - Stopwatch: Statechart of class Timer with b=TRUE

And we have made the same modification in my SDTestScenario in TestPackages → TPkg\_StopWatch → TestPackages → TCon\_StopWatch\_Architecture → TestContexts → TCon\_StopWatch → TestCases → tc\_check\_init() → TestScenarios

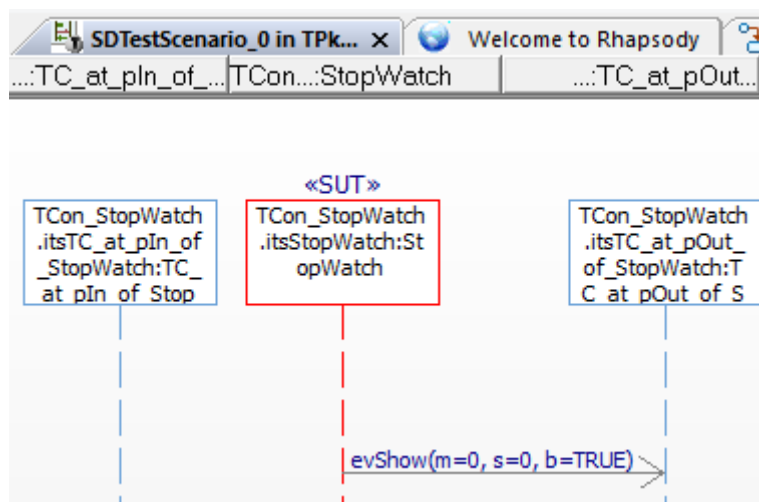


Figure 20 - Stopwatch: SDTestScenario

After this, we have executed TestCase by Click Right on tc\_check\_init() like bellow:

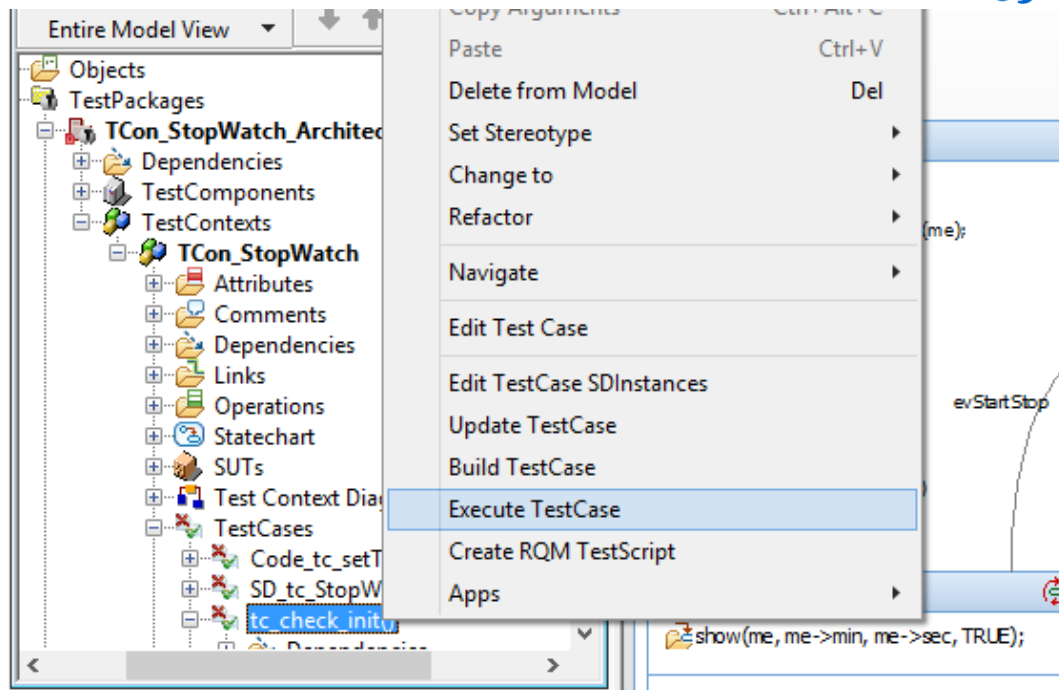


Figure 21 - Stopwatch: Execute TestCase

And we passed all the test we have modified.

Name	Status	File/Iteration	Line/Progress
tc_check_init	✓ PASSED		
SD_tc_0	✓ PASSED	1	100% (2/2)
Detailed Assertion Information			
'evShow': Check of in value of argument m (Actual value: 0)	✓ PASSED	TC_at_pOut_...	126
'evShow': Check of in value of argument s (Actual value: 0)	✓ PASSED	TC_at_pOut_...	131
'evShow': Check of in value of argument b (Actual value: 1)	✓ PASSED	TC_at_pOut_...	136

Figure 22 - Stopwatch: Results of TestCase

The difference when the parameter b =FALSE is that the current value is now 0 instead of 1.

First, Test Conductor started on executing the TestCase SC\_tc\_0. And we can see the figure below:

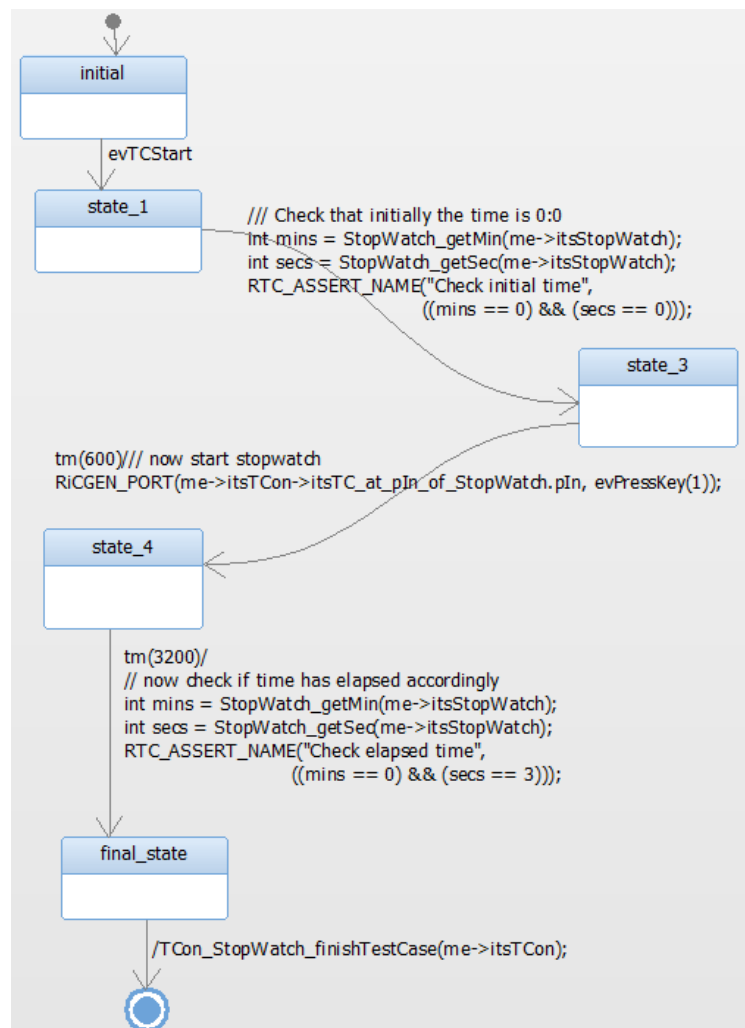


Figure 23 - Stopwatch: Statechart Test Case

This is a Statechart TestCase, named automatically 'TCSC\_tc\_0'.

On updating SC\_tc\_0, Test Conductor instruments the TestArchitecture accordingly and adds the call starting the statechart to the implementation body of SC\_tc\_0.

The next step is that Test Conductor generate automatically the Animated SDTestScenario like bellow for b=TRUE:

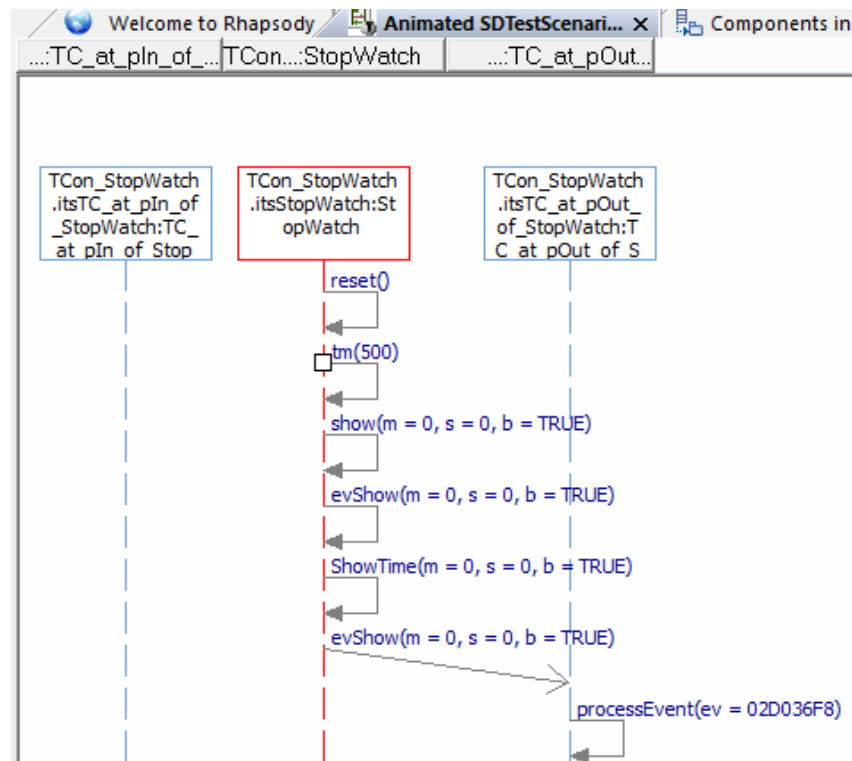


Figure 24 - StopWatch: Animated SDTestScenario with b=TRUE

And for b = FALSE, we have the same sequence but with b = FALSE:

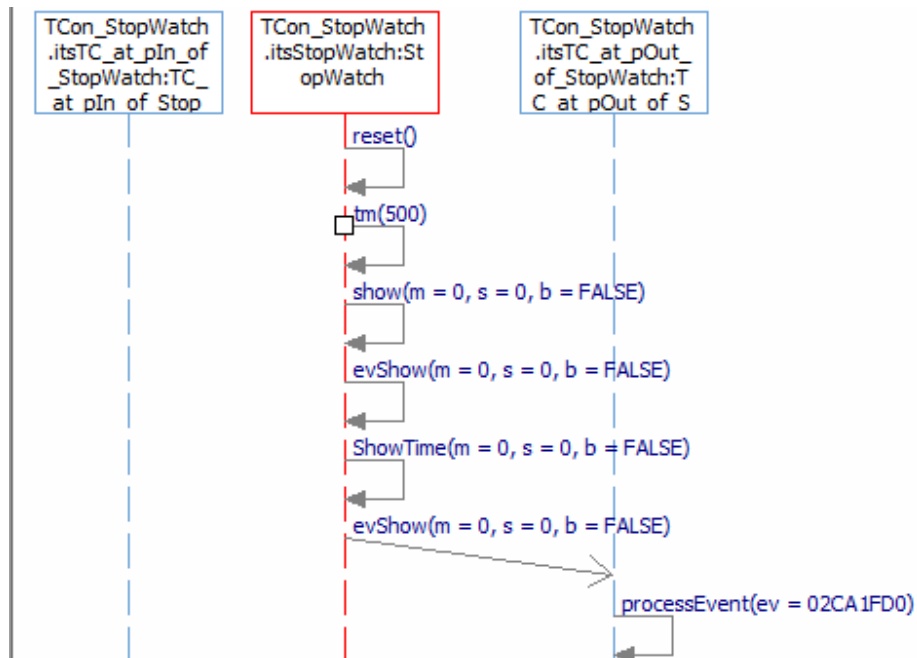


Figure 25 - StopWatch: Animated SDTestScenario with b=FALSE



### 3.2.5 How this TestScenario is generated?

This is exactly what happens when starting stopwatch: Stopwatch is started, Stopwatch has no statechart, but starts all its parts, i.e. Button, Display and Timer. Button and Display do nothing without being triggered, but Timer immediately starts working with:

- Reset() from the Timer Statechart: Transition 1
- tm(500) from the Timer Statechart: Transition 2
- show(\_) in the Timer Statechart: State 3 (off) from Stopwatch Overview in itsTimer
- evShow(\_) from Stopwatch Overview Timer TO Display
- ShowTime (\_\_)from Stopwatch Overview in itsDisplay
- evShow(\_) from Stopwatch Overview Display TO it's port pOut and thus send to pOut of Stopwatch.
- processEvent(\_) is in TestPackages → TestComponents → TC\_at\_pOut\_of\_StopWatch → Operations --> processEvent (RiCEvent\* ev). processEvent is called in TC\_at\_pOut\_of\_StopWatch.

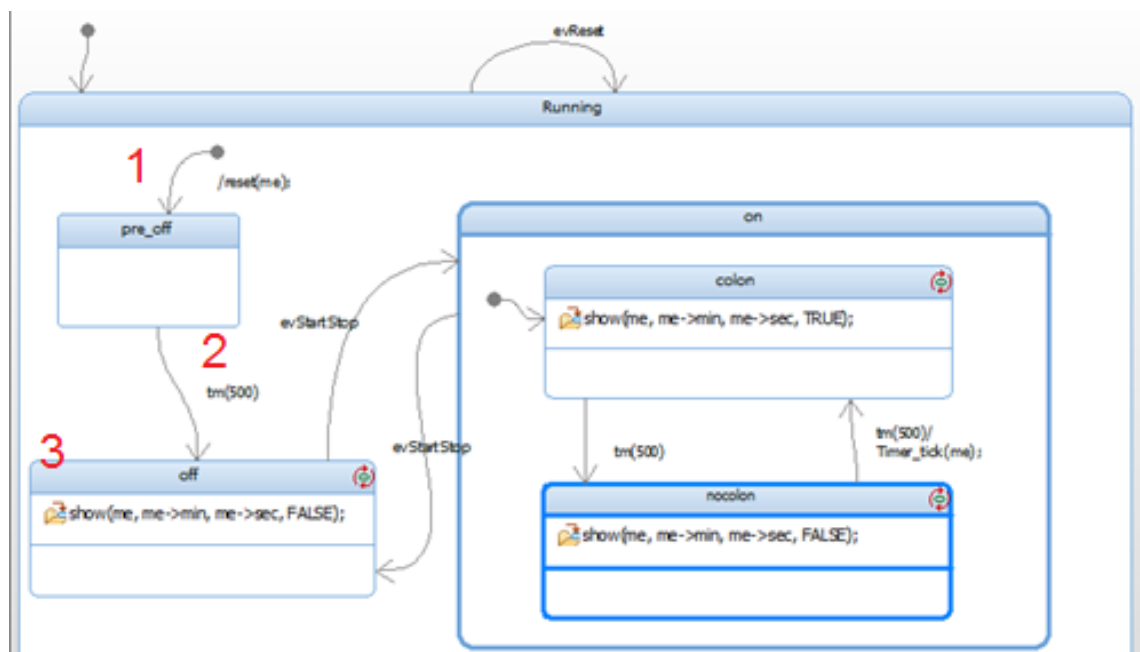


Figure 26- Stopwatch: Statechart of class Timer





Transition 1 (default transition) calls `reset()`. Entering state `pre_off` starts a 500ms timer. When this timer expires, `pre_off` is exited and transition 2 takes us to state `off`. On entering state `off` the 'Action on entry' is executed, i.e. Timer's operation `show()` is invoked, which sends `evShow` via port `pTimer` to `Display`. On reception of `evShow`, `Display` invokes its `ShowTime()` operation, which in turn sends `evShow` to `Display`'s port `pOut` (connected to `TC_at_pOut_of_Stopwatch`).

### 3.3 Link Test Conductor with Rhapsody and Keil $\mu$ Vision

For the explication, we will take a simple exemple which is blinky LEDs. The creation of this example is based on the documentation from Willert. [5]

In this project, we use IBM® Rational® Rhapsody® Developer for C, the version 8.1.5, 64bit. And the MCB1700 Evaluation Board with the debugger ULINK.

#### 3.3.1 Create appropriate profiles

First of all, we need to create the profiles with a demo DVD including IBM Rational Rhapsody, Willert Frameworks and Keil  $\mu$ Vision IDE. This can be download on the website of Willert. [6]

After have installed the software, we have the possibility to choose between two toolchain. The first one is a compiler toolchain Keil  $\mu$ Vision for Cortex M3, used with the MCB1700 evaluation board. The second is Visual Studio 2013 to be used with the simulation on PC. For this, the installation of Microsoft Visual C++ 2013 Express edition is needed.

Then, we can choose between two different modeling tools depending on the code generation language we want: AINSI C or AINSI C++.

On this demo DVD, we can download Keil  $\mu$ Vision. Note that this is an evaluation with a 32k code size limit. And also a link to download the IBM Rational Rhapsody Developer is available. In more, the installation of the Willert RXF is needed. It is in this way that the profiles can be created. The profiles will be installed in a folder named 'RXF-Eval\_Rpy-C-ARM\_V6.09'

Willert RXF for Rhapsody **in C**, Keil  $\mu$ Vision and Cortex M3

Willert RXF for Rhapsody **in C**, Keil  $\mu$ Vision and Cortex M3 on real Hardware or with the Keil Simulator

⬇ Willert RXF for Rhapsody **in C**, Keil  $\mu$ Vision and Cortex M3

Figure 27 - The installation of Willert RXF



After agree with the conditions, we will be asked to specify the paths to several tools we need. This depends on each computer and where we put the different installations. First we need the path of the installation of Willert Software Tools:

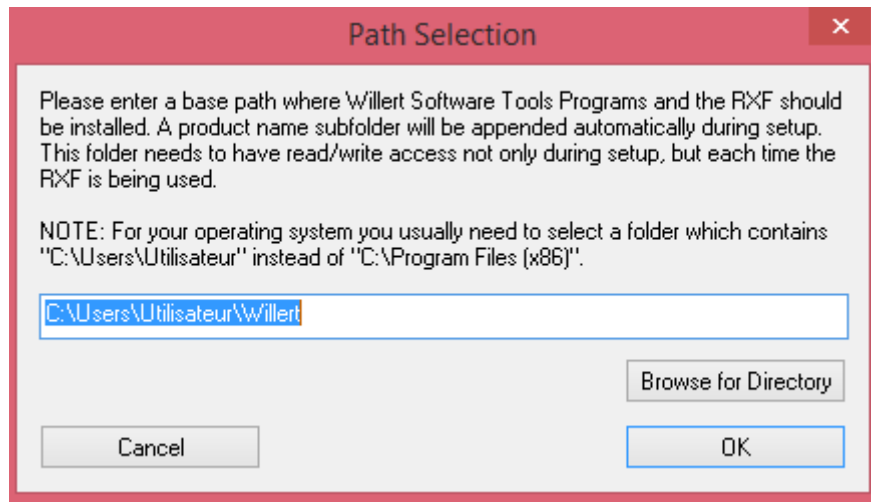


Figure 28 - Select path of Willert Software Tools Program

After, we need to enter the path for the Keil main directory:

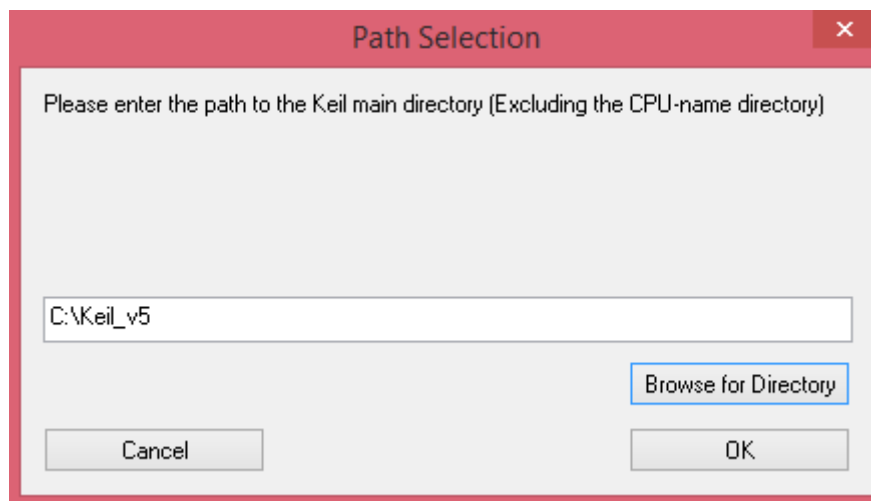


Figure 29 - Select path of Keil main directory



Then, the path of the Rhapsody data folder containing the “Share” directory:

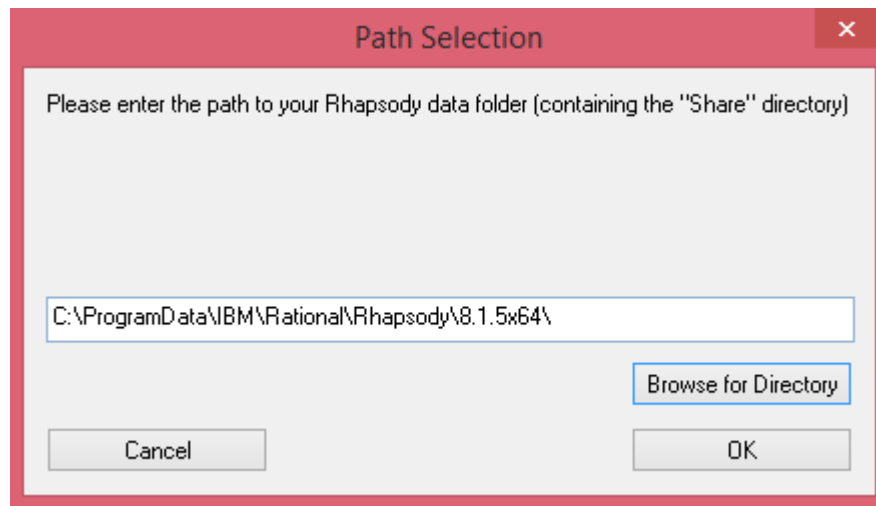


Figure 30 - Select the path of Rhapsody data folder

Finally, the path of the Rhapsody program files folder which contain the “Rhapsody.exe”:

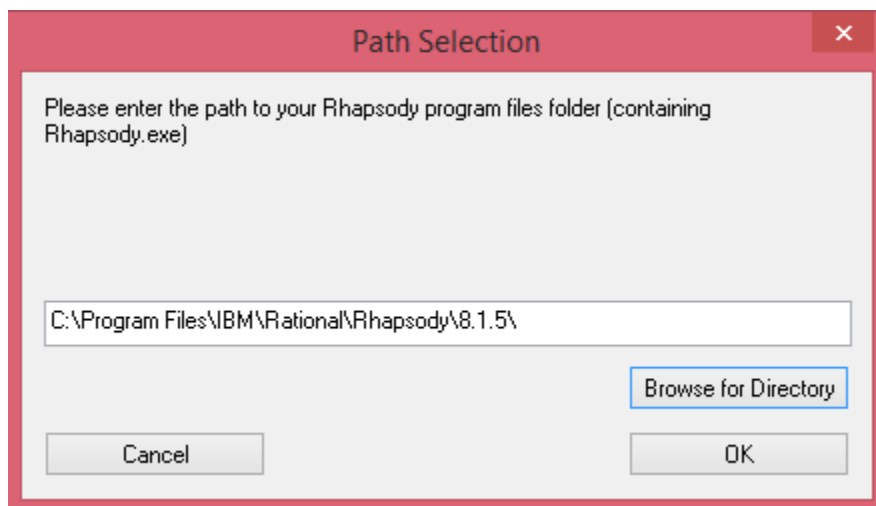


Figure 31 - Select the path of Rhapsody program files folder

Now the profile is created in: C:\Users\Utilisateur\Willert\RXF-Eval\_Rpy-C-ARM\_V6.09. These profiles can be used on a new project.



### 3.3.2 Create a new project

By starting IBM Rational Rhapsody for C 8.1.5, a Welcome-Screen appear and we can click on “File → New” from the menu bar on the top or click “New Project” like the image below:



Figure 32 - Create a new project

After have determinated an appropriate name for the project, we can select the location for the project and continue by ‘OK’.

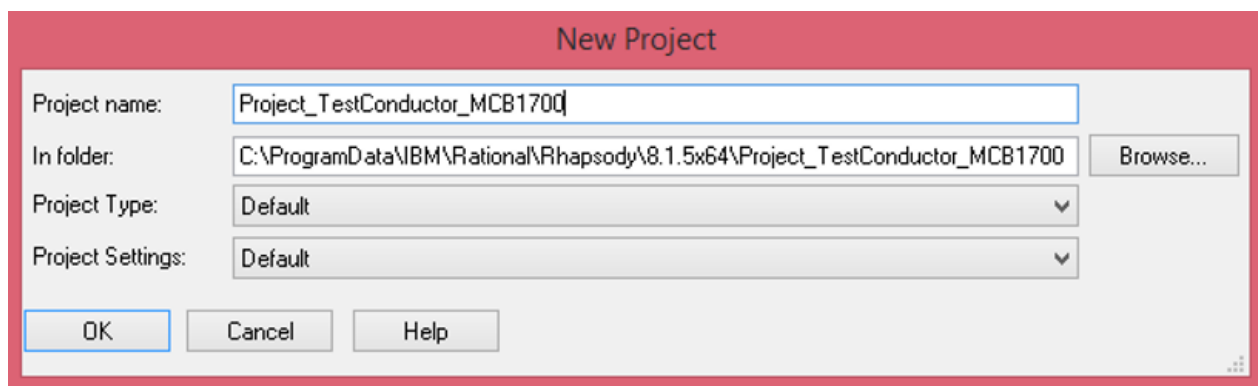


Figure 33 - Name and location for the new project

Then, a message appear to create the directory of the project. Click on ‘Yes’ to continue.

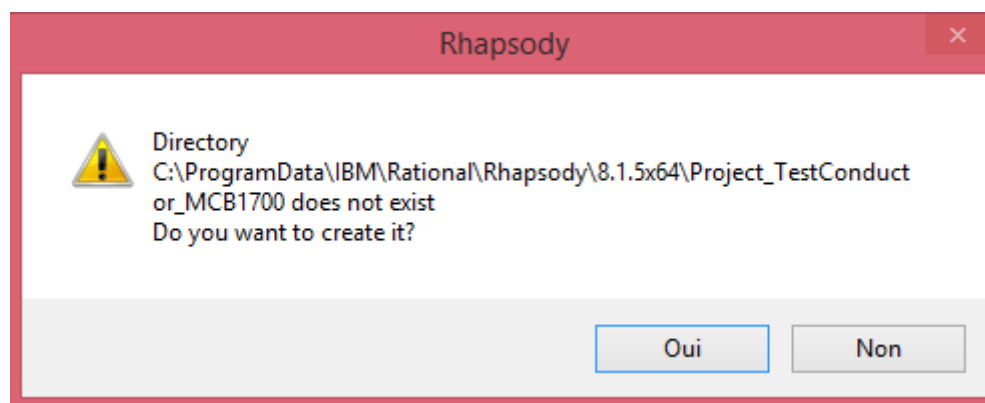


Figure 34 - Create the directory



We can start with renaming some items in the model browser to better understand the meaning of each files:

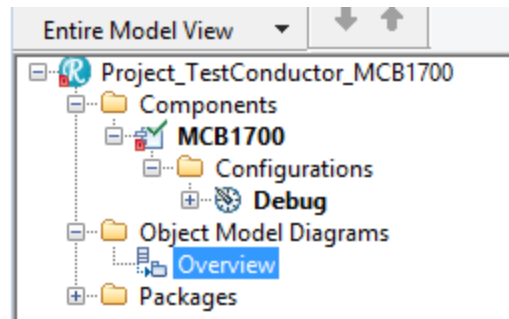


Figure 35 - Rename in the model browser

### 3.3.3 Adding Profiles

We can now add the profiles created before in order to add settings to the Rhapsody project. These profiles are used to adapt the code generation of Rhapsody to work with our Framework.

Click on 'File' → 'Add Profile to Model' and navigate to the folder called "RXF-Eval\_Rpy-C-ARM\_V6.09" containing profiles and select the extension named '.sbs'.

Disque local (C:) > Utilisateurs > Utilisateur > Willert > RXF-Eval_Rpy-C-ARM_V6.09 > Config > UserSettings > RXF-Eval_Rpy-C-ARM_Profile_rpy			
Nom	Modifié le	Type	Taille
RXF-Eval_Rpy-C-ARM_Profile.hep	17/01/2017 15:53	Fichier HEP	3 Ko
RXF-Eval_Rpy-C-ARM_Profile.sbs	18/12/2016 16:03	Fichier SBS	43 Ko
RXF-Eval_Rpy-C-ARM_Profile.txt	18/12/2016 16:03	Document texte	1 Ko
WST_CG_Profile.sbs	18/12/2016 16:03	Fichier SBS	10 Ko
WST_CG_Profile.txt	18/12/2016 16:03	Document texte	1 Ko
WSTProfile.sbs	18/12/2016 16:03	Fichier SBS	7 Ko
WSTProfile.txt	18/12/2016 16:03	Document texte	1 Ko

Figure 36 - Location for the profiles

The first Profile belongs to the Willert RXF "RXF-Eval\_Rpy-C-ARM\_V6.09" and needs to be included in all models containing components which generate code for the Realtime eXecution Framework. It allows the user to select the required Stereotypes.

The second Profile 'WST\_CG\_Profile.sbs' includes all target independent Stereotypes which modify the codegeneration.

And the last profile named 'WSTProfile.sbs' can optionally be loaded to override a set of mostly just look and feel properties to the Willert Software Tools recommended style.



In the model browser, the three profiles appear under the folder 'Profiles' with the indication 'REF' in bold. That means the profiles are not copied to the project directory but only referenced.

### 3.3.4 Project Settings, Methods and Attributes

Then, by double-click on the MCB1700 component and select the following stereotype in the pop-up window: 'RXFComponent in RXF-Eval\_Rpy-C-ARM\_Profile'. In more, in the general-tab, we enter under 'Standard Headers' the name of the header file "<lpc17xx.h>" that we want to include for the hardware specific component. With this parameter, the file LED.h will contains automatically the "#include <lpc17xx.h>".

Figure 37 - Enter the stereotype and Standard Headers

In the model browser → in Object Model Diagrams (OMD), we can add the class LED in 'Overview' with selecting a class in the elementpool for Diagrams on the right. This class can be also found in: Packages → Default → Classes → LED.

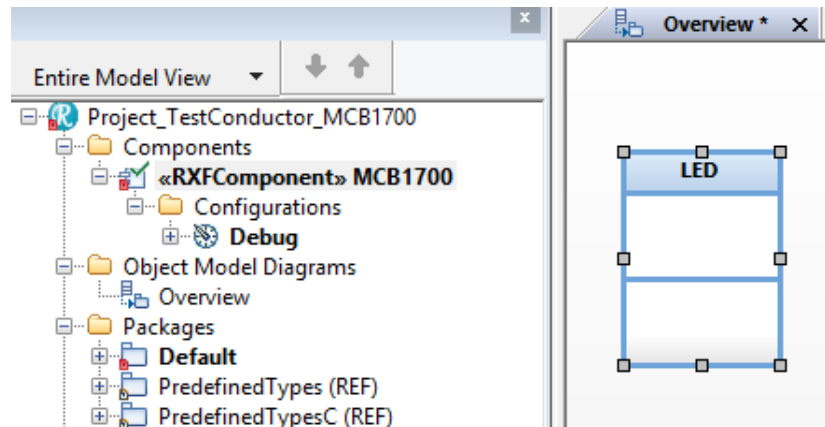


Figure 38 - Class LED and the model browser

In this class LED, we add two operations and one initializer. If the initializer don't work, the other way is to click right on 'Operations' in the Model-Browser and select 'Add New' → 'Initializer' and finish by only click on 'OK' in order to don't put arguments in the initializer. In this way, we add the initializer called 'Init' and two operations, one named 'on' and the second 'off'.

Class : LED in Default

Name	Visibility	Return Type
on	Public	void
off	Public	void
Init	Public	

Figure 39 - Operations in the class LED

Then, we add two attributes in the same way named 'bitNr' and 'delay'.

Class : LED in Default

Name	Visibility	Type
bitNr	Public	int
delay	Public	int

Figure 40 – Attributes of the class LED

In the initializer 'Init' we add two arguments which are 'aBitNr' and 'aDelay'.

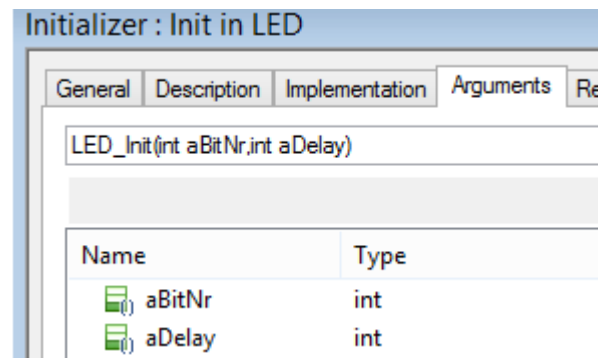


Figure 41 - Two arguments of Init

In the implementation Tab in the Init in LED, we add the following ANSI 'C' -code.

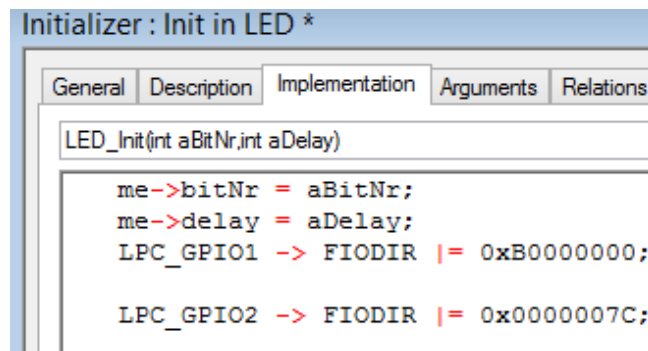


Figure 42 - Implementation Init in LED

In the same way, we add the appropriate implementation for the operations on() and off().

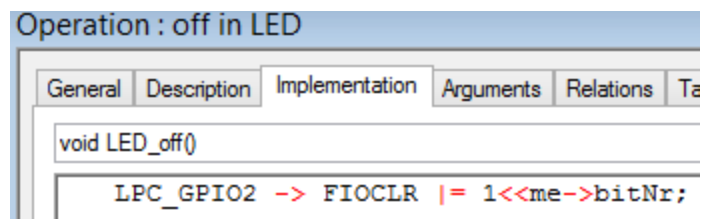


Figure 43 - Implementation off in LED

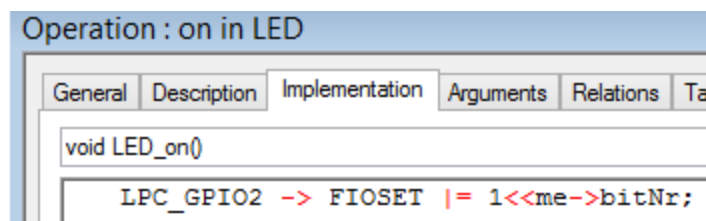


Figure 44 - Implementation on in LED





### 3.3.5 Statechart

In the model browser, when we right click on the LED class, we can 'Add New' → 'Diagram' → 'Statechart'. This state diagram will call the two operations "off" and "on" by this structure.

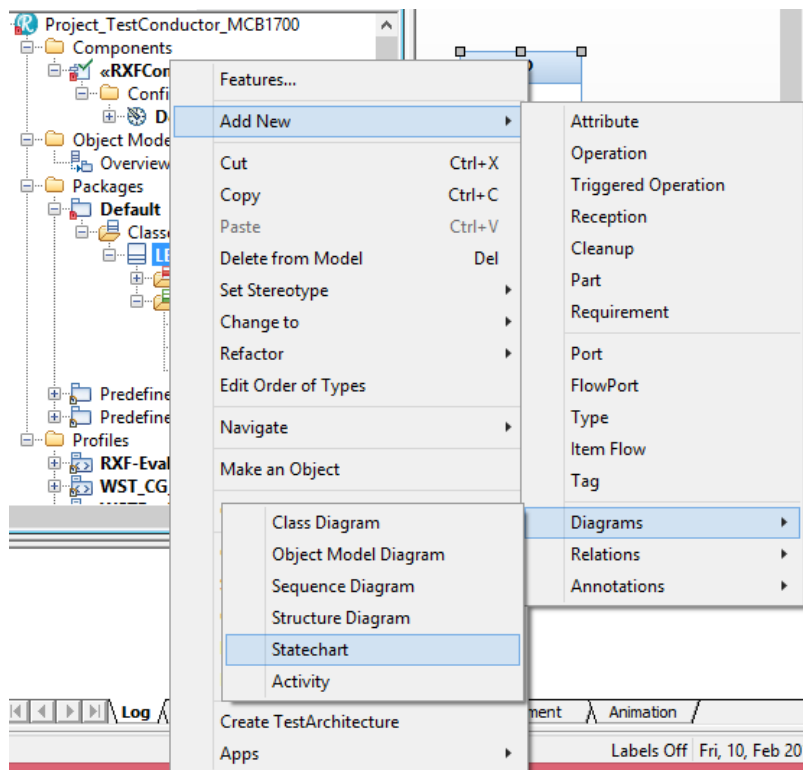


Figure 45 - Add new statechart on the LED

We create two states called 'State\_off' and 'State\_on' with one default transition and two transistion between the two states.

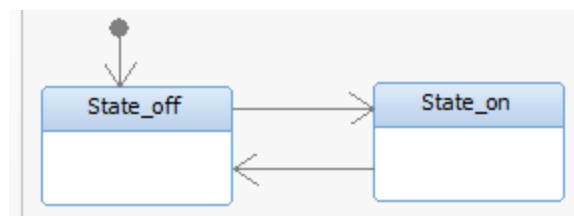


Figure 46 - Build the statechart



We enter the call of the 'off' operation under 'Action on entry'.

State : State\_off in statechart\_4

General Description Relations Tags

Name: State\_off

Stereotype:

Action on entry

LED\_off(me);

Figure 47 - Action on entry 'off'

Do the same for the 'on' operation.

State : State\_on in statechart\_4

General Description Relations Tags

Name: State\_on

Stereotype:

Action on entry

LED\_on(me);

Figure 48 - Action on entry 'on'

We also enter a delay to wait a specified time to go from one state to another. For this, we double-click on the first transition and add under 'Trigger' this below:

Transition : 0 in statechart\_4 \*

General Description Tags Properties

Name :

Stereotype:

Target State\_on

Trigger : tm(me->delay)

Guard :

Figure 49 - Add a trigger

Same for the second transition.

After all this, we have our final statechart which look like the figure below.

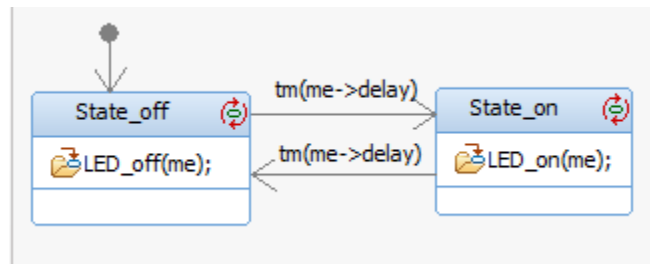


Figure 50 - Final statechart

### 3.3.6 Instances of a Class

Click right on 'Object Model Diagrams' in the Model-Browser → click on 'Add New Object Model Diagrams' and call it 'Runtime'.

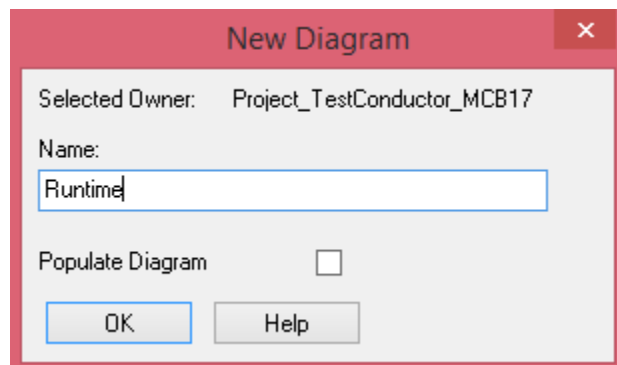


Figure 51 - Add new OMD called 'Runtime'

Now we can drag the 'LED' class from the model browser in the 'Runtime' OMD.

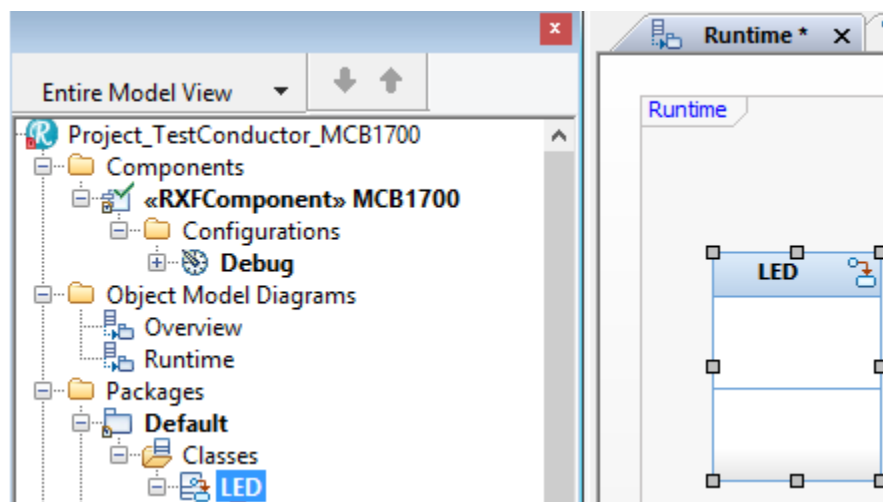


Figure 52 - Drag 'LED' class in Runtime OMD



Make an object by right click on the LED class. It is now called 'itsLED'.

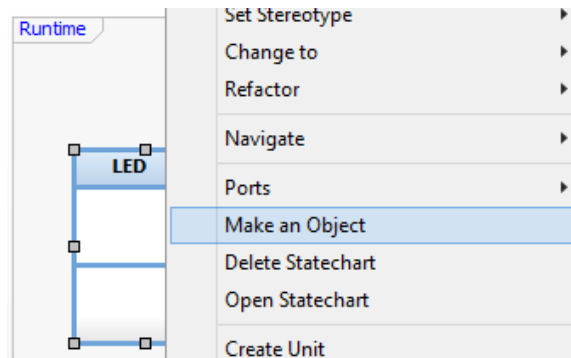


Figure 53 - Make an object

Then, we need to enter a suitable 'BitNr' and a value for the 'Delay' in our statechart. By double-click on the object 'itsLED', in the General Tab, we can add this by clicking on the extend button at the end of 'Initialization'

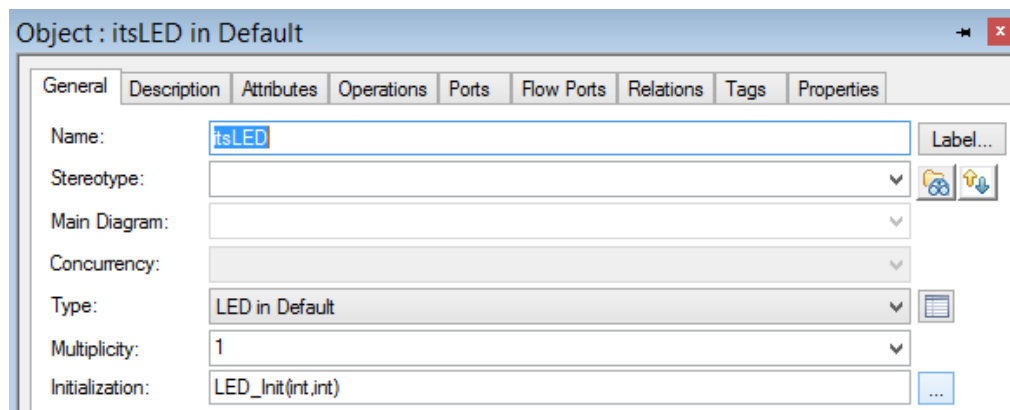


Figure 54 - Initialization of itsLED

By click on 'Set Value' we can enter "2" for 'aBitNr' corresponding to the LED P2.2 on the Keil MCB1700 and we enter "100" corresponding to milliseconds.

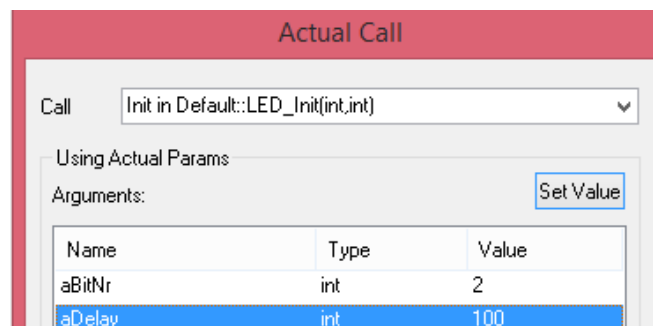


Figure 55 – First set value in Init



Repeat the same procedure with an other LED class that we drag. However for the second object, we add “3” for ‘aBitNr’ and “50” for ‘aDelay’.

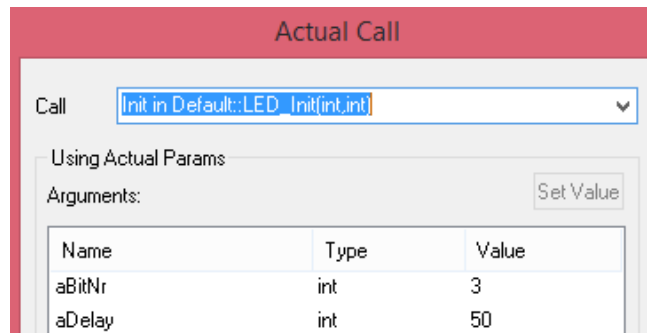


Figure 56 - Second set value in Init

### 3.3.8 Generate / Make / Run

When we click on the Generate / Make / run button on Rhapsody, a message appear to create the ‘Debug’ directory. Say ‘Yes’ to continue.

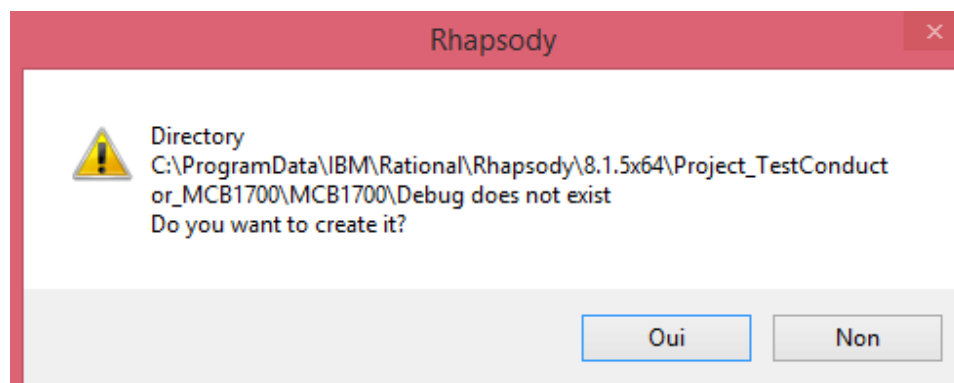


Figure 57 - Create the debug directory

This automatically start the deployer configuration when it is the first time running a project. We can put the destination of the project. This is genraly in : ‘<product installation path> \Willert \RXF-Eval\_Rpy-C-ARM\_V6.09 \Samples \Code \GettingStarted \GettingStarted.uvprojx’. Take the project with the extension ‘.uvprojx’.

And then, we just have to click on save. Normally, we don’t have to enter this again if it is the same project.

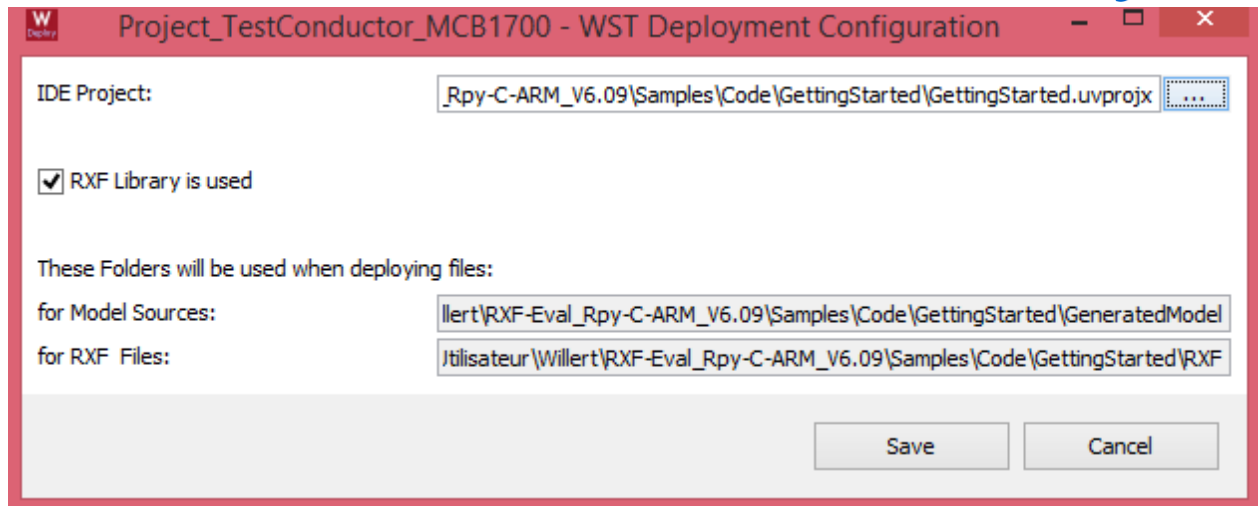


Figure 58 - The deployer configuration

### 3.3.9 Keil $\mu$ Vision Development Environment

When the build is done on Rhapsody, we can open the project on Keil  $\mu$ Vision by click on 'Tools'  
→ 'RXF-Eval\_Rpy-C-ARM\_V6.09 Open uVision5 Project'

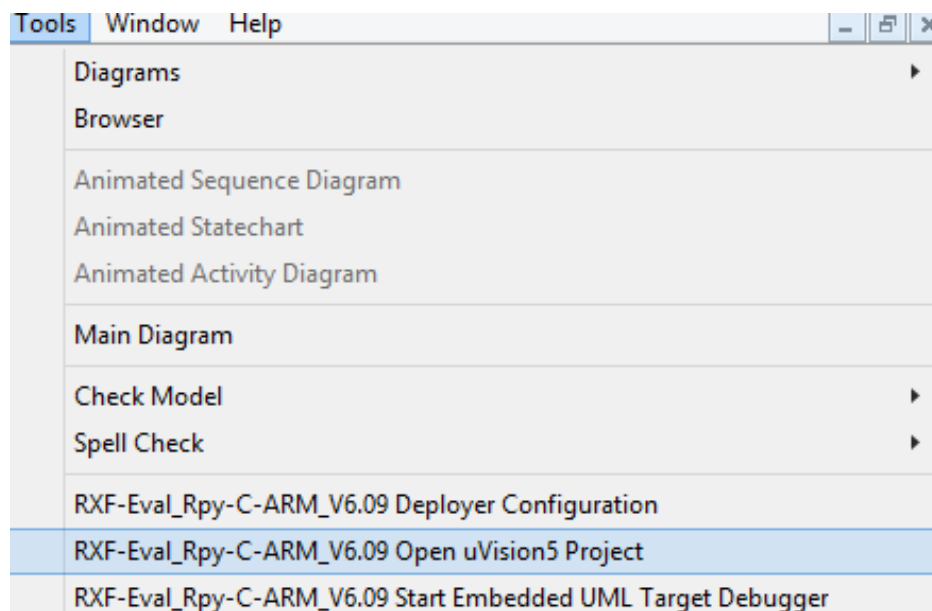


Figure 59 - Open Keil  $\mu$ Vision

Then, the Keil software is open.

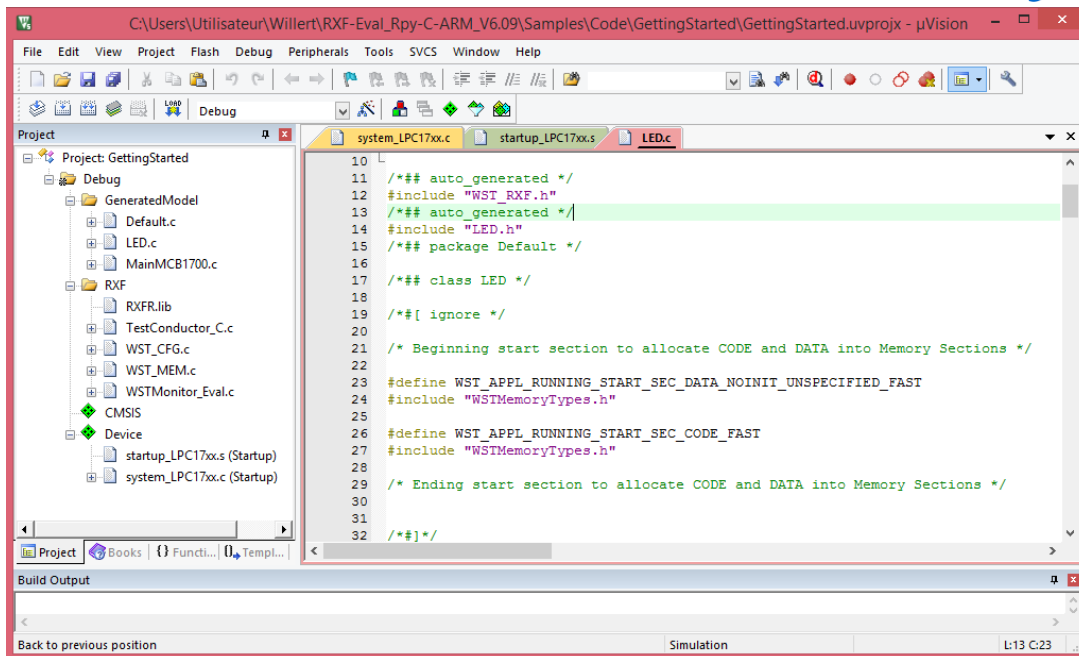


Figure 60 - Keil μVision environment

It is important to switch the target from Simulator to the Hardware in the 'Debug' tab.

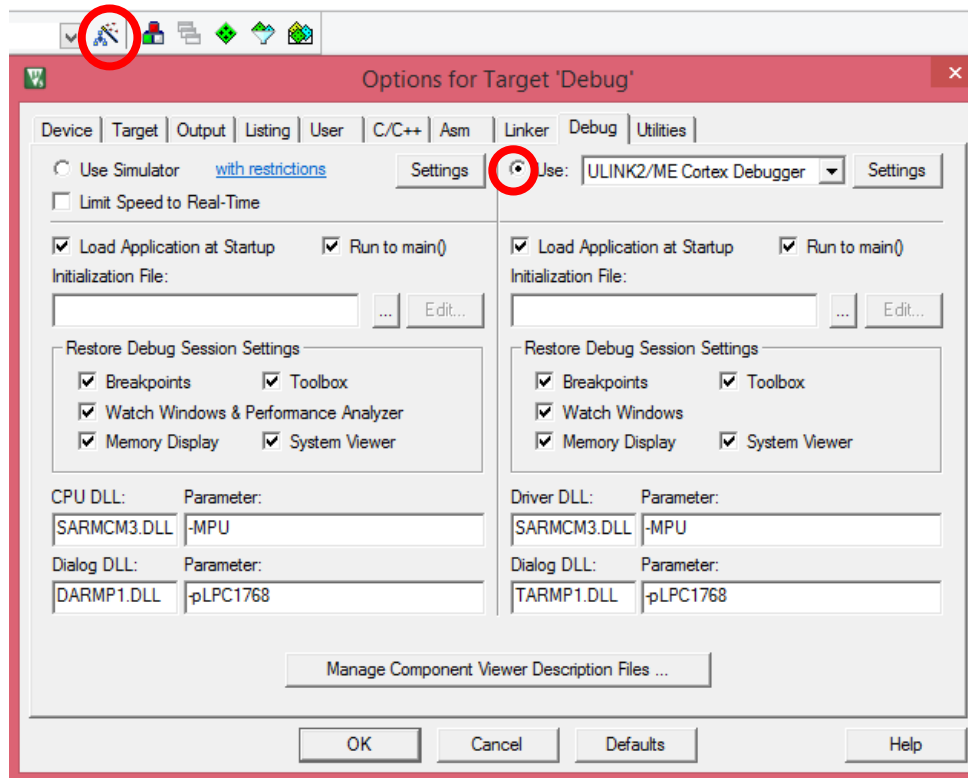


Figure 61 - Switch Target from Simulator to Hardware



To fix the problem with TCP/IP connexion, we have to open the installation folder of the Keil RXF. Enter the folder Tools / TargetDebugger and start the Batch file called 'Embedded\_Uml\_Target\_Debugger.bat'.

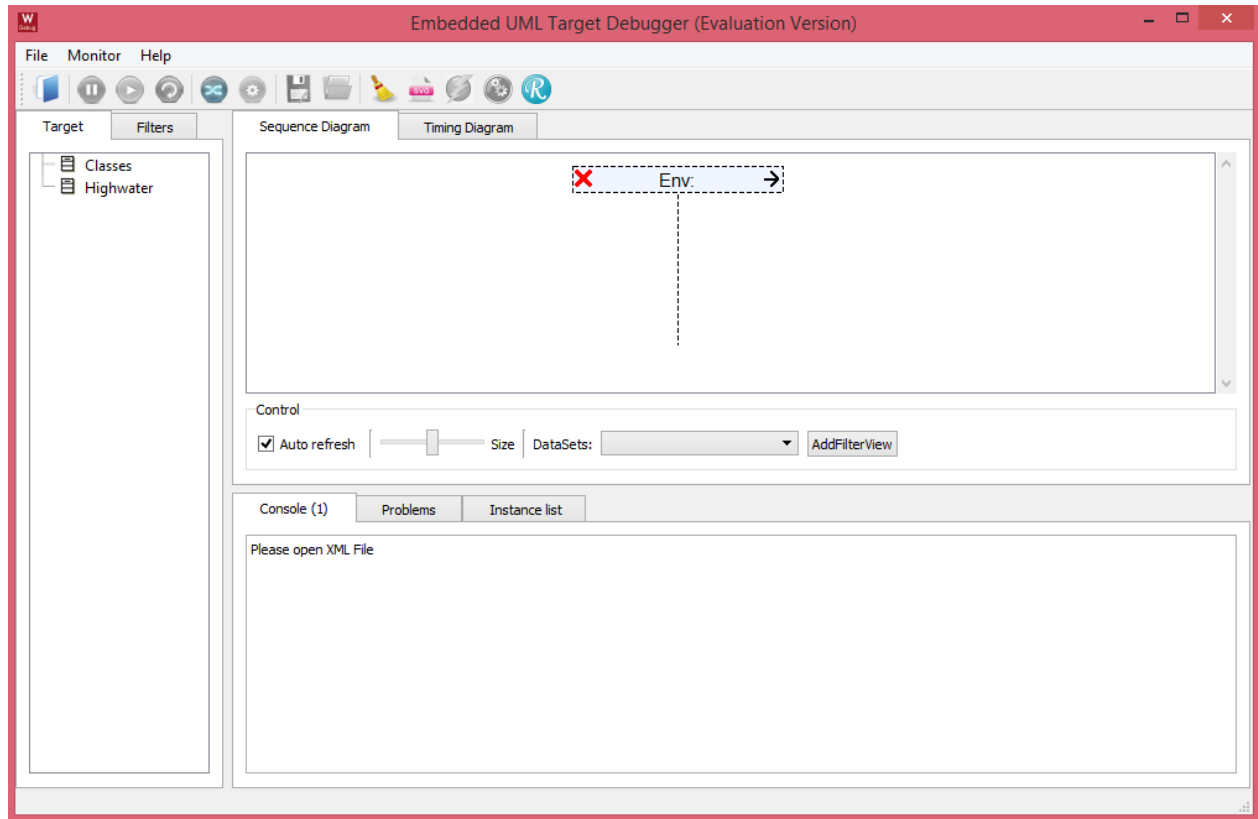


Figure 62 - 'Embedded\_Uml\_Target\_Debugger.bat'

Now we can click in the menu bar 'Monitor' → 'Select Communication Plugin'. And choose Keil Plugin named 'Keil JTAG' and finish by click on 'Ok'.

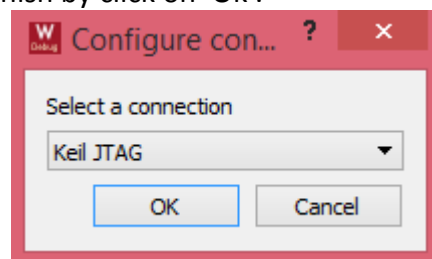
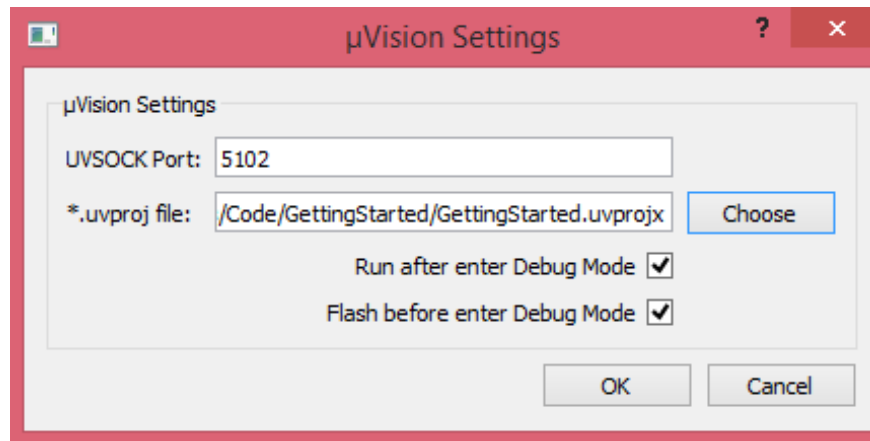


Figure 63 - Configure communication plugin

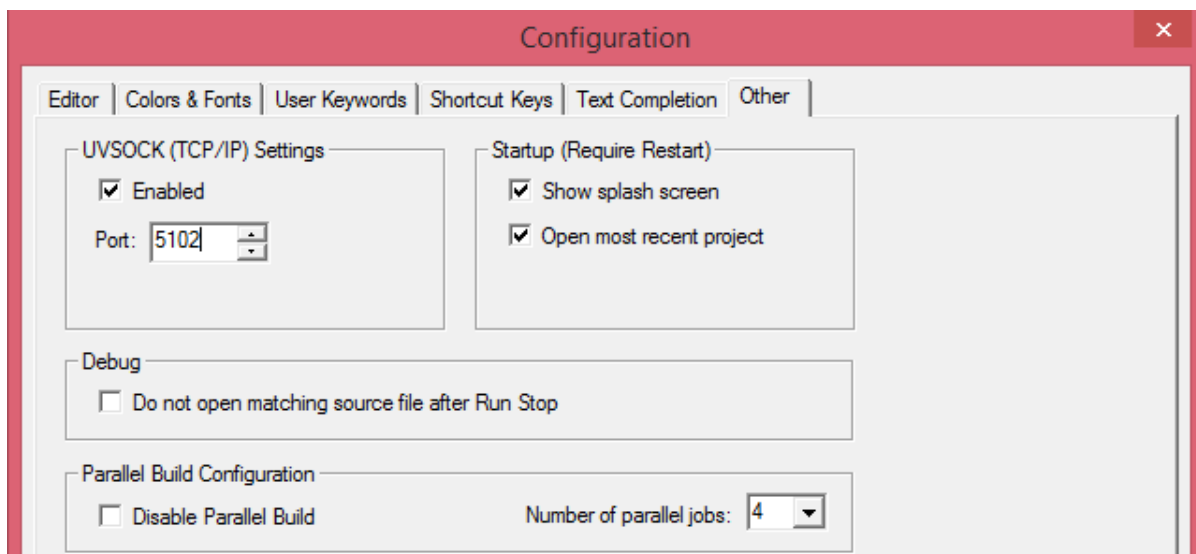
Then the following figure should appear:



*Figure 64 - µVision Settings*

Enter the path where is the file with the extension '.uvprojx' and click on 'OK'.

Then in the Keil µVision, in the 'MenuBar Edit' → 'Configuration Tab' → 'Other' we can find the Port, click on 'Enable' if this does not. And inside the UVSOCK Port we should input the same port number of the Keil UVSOCK.

*Figure 65 - Configuration in Keil µVision*

From this point we should be able to run tests on the target like we do it on the host.

We may now build and download the example project to the evaluation board using the µVision commands below:



- **Project** -> **Build target (F7)**
- **Flash** -> **Download (F8)**
- **Debug** -> **Start/Stop Debug Session (Ctrl + F5)**

At this point, a message appear to remind the Keil  $\mu$ Vision is a evaluation mode. Click on 'OK'.

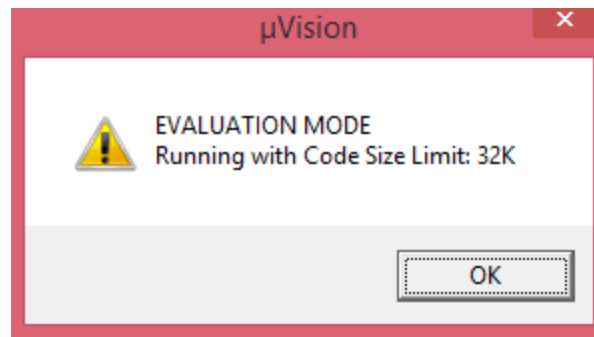


Figure 66 - Keil Message evaluation mode

- **Debug** -> **Run (F5)**

The project work well in the target.

### 3.3.10 Using Test Conductor

Now, we use Rhapsody Test Conductor in combination with the target.

First we create a Test Architecture on click right on the class LED like the figure below:

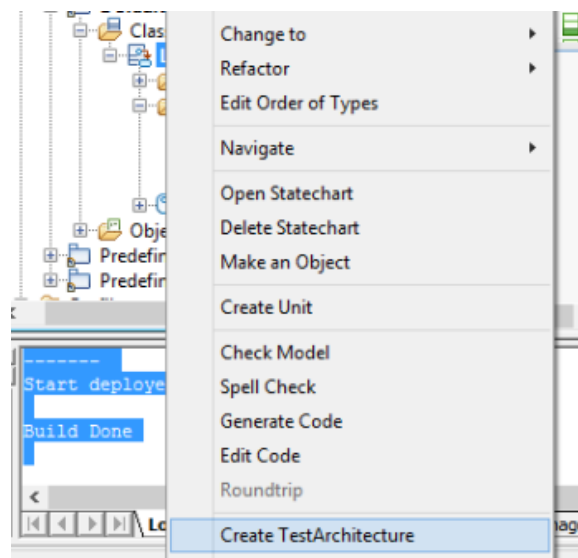


Figure 67 - Create Test Architecture on LED



The add of the Rhapsody TestingProfile to the project allow the access the functionalities of Test Conductor. Click on 'Yes'.

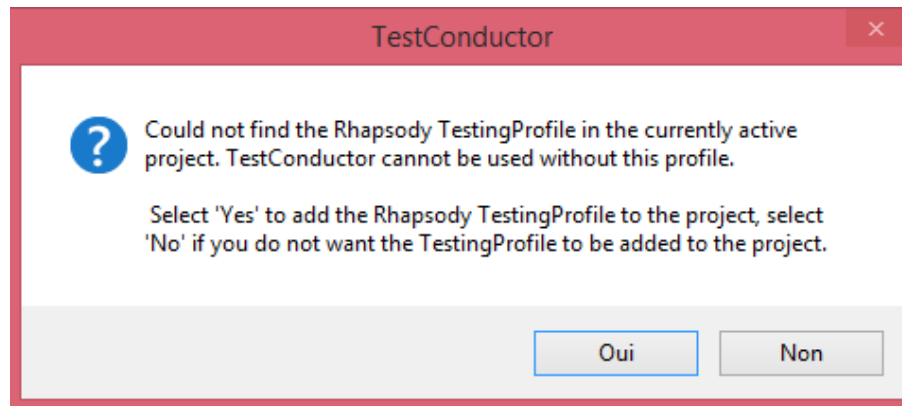


Figure 68 - Add TestingProfiles

Open the component 'Features' on right click on 'TPkg\_LED\_Comp' and set the stereotype by choosing 'MonitorForTC...' and 'RXFComponentet...'.

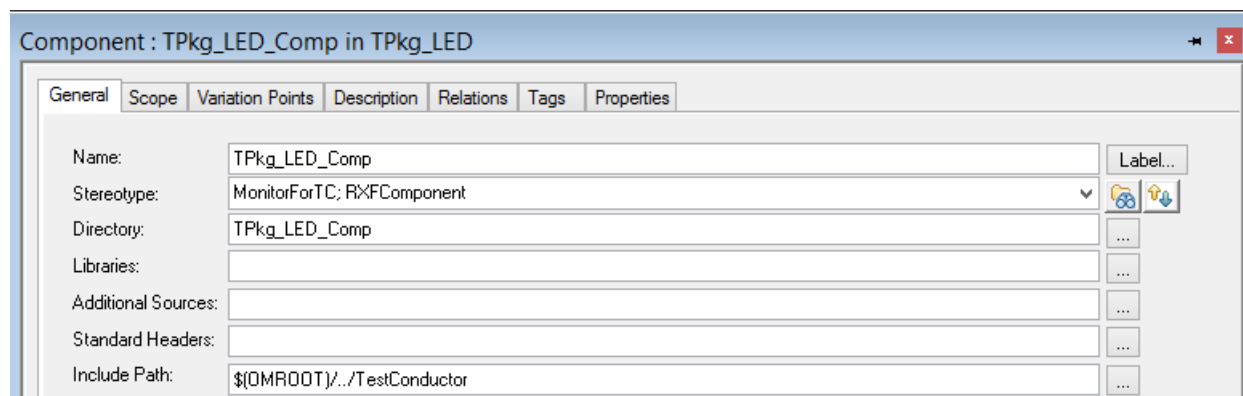


Figure 69 - Set the stereotype in 'TPkg\_LED\_Comp'

Then, by right click on 'TestingConfiguration DefaultConfig' open 'Features' and deselect all stereotype and select only 'WSTTargetTestingConfiguration in TestArchitecture' in the general tab.

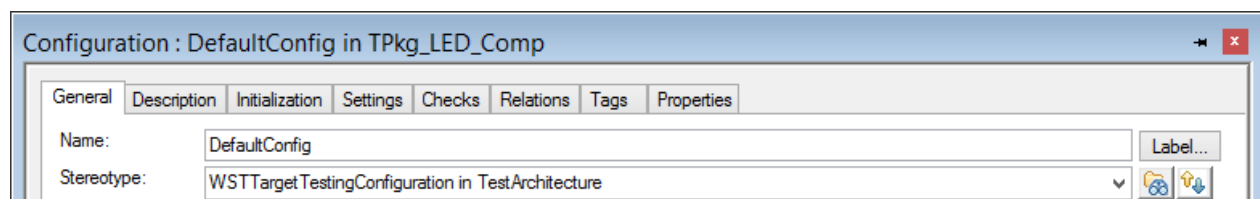


Figure 70 - Set the stereotype in 'TestingConfiguration DefaultConfig'



In the same configuration, in 'Settings', modify the 'Instrumentation Mode' to 'None' and add <lpc17xx.h> in 'Standard Headers'.

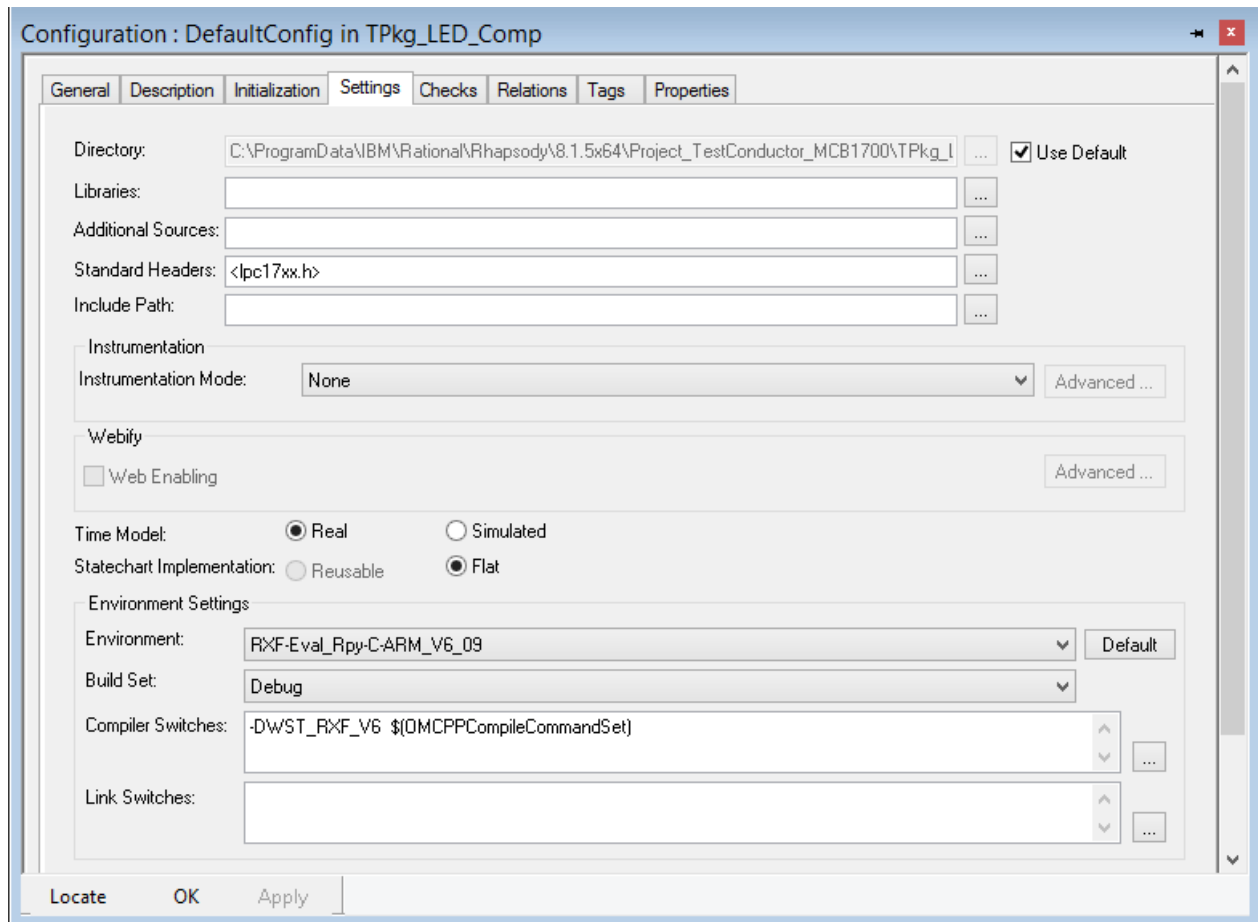


Figure 71 - Set the settings in 'TestingConfiguration DefaultConfig'

To finish with the configuration in 'TestingConfiguration DefaultConfig' in the Tags tab, we enter the path of the project with the extension '.uvprojx' in 'TargetProxyIDEProject' and select 'user\_defined' in 'rtc\_log\_kind' like the figure show below:

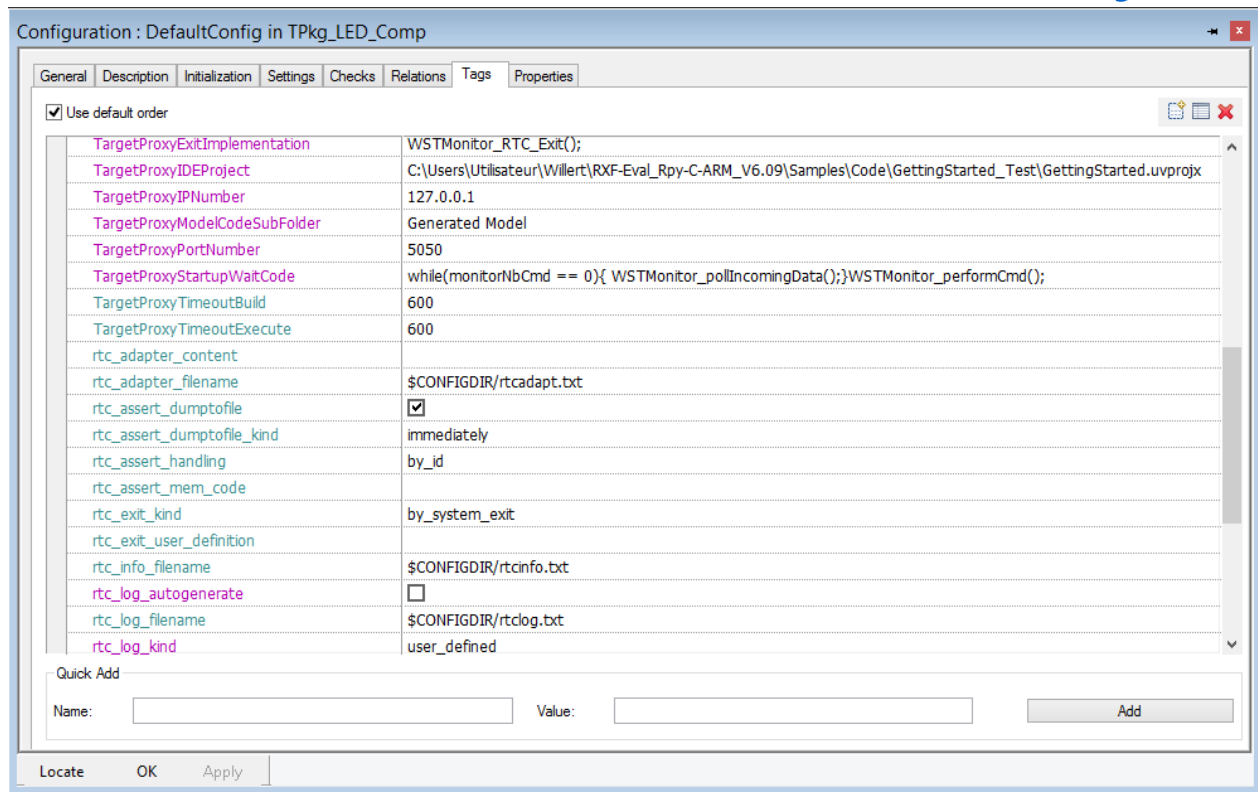


Figure 72 - Set the tags in 'TestingConfiguration DefaultConfig'

Open the configuration of 'itsLED' in the SUT. Then open the 'Initialization' dialog with the extend button at the right. We can add arguments for the LED 'Init'.

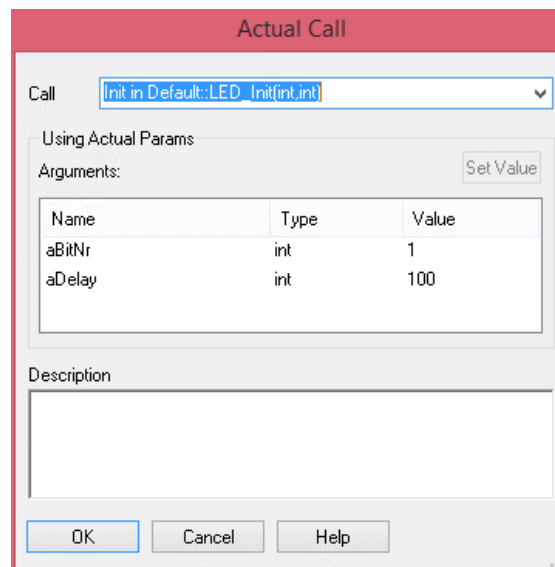


Figure 73 - Add arguments in itsLED



Then, we need to create Test Cases because otherwise we can not test without test cases.

First we create a code test case because it is the simplest test case we can create. The advantage of this test case is that we have to do nothing in the test.



Figure 74 - Code test case in the Model Browser

Have a look in the 'Initialization' tab of 'DefaultConfig in TPkg\_LED\_Comp', to be sure that we have a initialization code, like the figure show below:

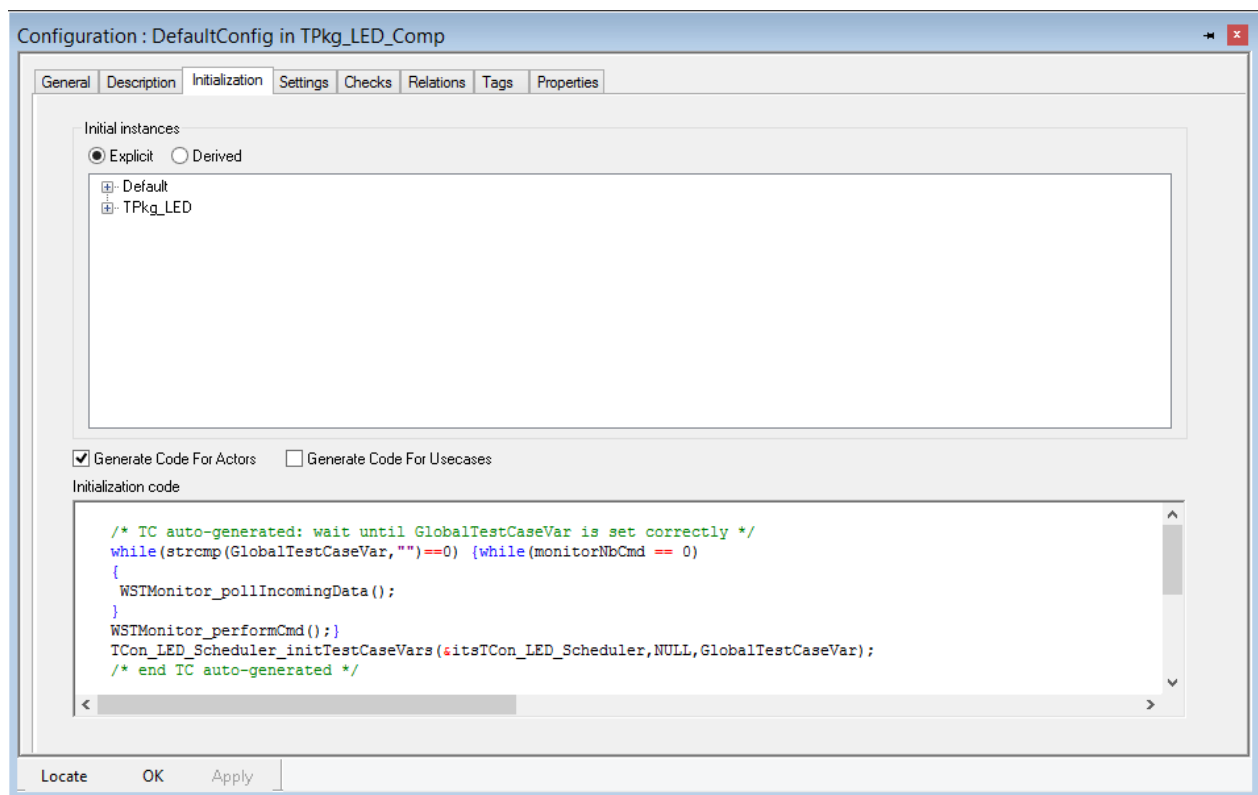


Figure 75 - Initialization code in 'DefaultConfig'

Now we can 'update TestContext' by right click on 'TCon\_LED'.

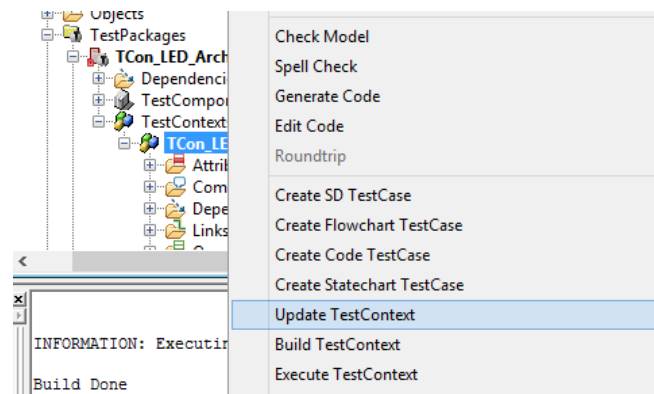


Figure 76 - Update TestContext

Then, we can 'Build TestContext'

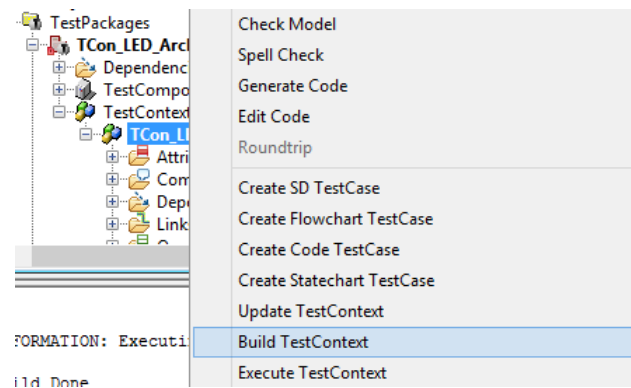


Figure 77 - Build TestContext

And finally, 'Execute TestContext' by clicking right on 'TCon\_LED' like the two step above.

Go back on Keil uVision and Reload → Rebuild → Run.

After the time needed the test on Rhapsody are successfully validated. When you execute test cases, it's much more quick on a real target than with the simulator on uVision Keil.

A html page is available to see the result of the test case in a summary. This result can be found in the Model Browser in Rhapsody, in TestPackages → TestContexts:

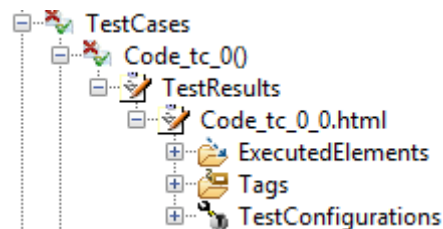


Figure 78 - Location of html result of TestCases



## TestCase Result

TestCase: Code\_tc\_0

Friday, February 10, 2017 10:36:44

Environment Information	
Test executed on machine:	DELL
Test executed by user:	Isabelle
Used operating system version:	Windows 8 / Windows 8.1
Used Rhapsody version:	8.1.5, build 9728113
Used TestConductor version:	2.6.5, build 4550

Tested Project	
Project:	Project_TestConductor_MCB1700
Active Code Generation Component:	TPkg_LED_Comp
Active Code Generation Configuration:	DefaultConfig

Results	Summary: PASSED
Initial	PASSED

Result Verification
No result verification for Code TestCases

Figure 79 - Result of Code Test Case

Now we want to test with Sequence Diagram.

So with the modification below in the statechart of LED, that are renamed the trigger on the two transitions by 'evOn' and 'evOff'.

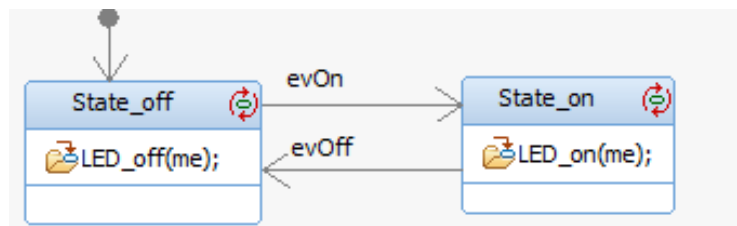


Figure 80 - Statechart for SD test case





By right click on 'TCon\_LED' → select 'Create SD TestCase'.

The dummy driver on the right is a test component which can be use for diriving the LEDs. In order to check the result of the execution, we can add TestAction to the dummy and a Time interval between the event send and the test action to wait the time needed.

In the diagram tools, we need:



We can draw the Sequence Diagram Test Scenario which specify the test case like the figure below show:

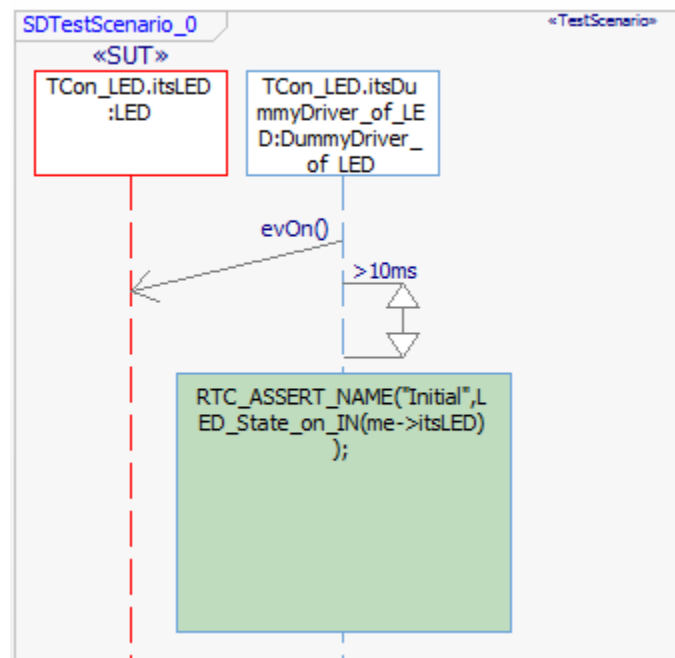


Figure 81 - SD Test Scenario

After sending the event 'evOn()' to the SUT, we want to check if the LED is switch to the other state, that is mean from the state 'on' to the state 'off'. For this, we need to add a delay who can allow to wait the time we need to change from one state to another.

We need the delay because when you send the event, this event can't be consumed immediately, it will take some time.

With the test action, we want to test in the LED.h : 'int LED\_State\_on\_IN(const LED\* const me);'.



We can and execute the SD\_tc\_0(). The software propose automatically to update and build the SD Test Case before executed because he recognize that we have modified the test case.

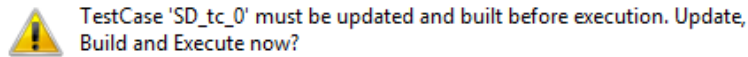


Figure 82 - Message to update and built before execute test case

We can see in Rhapsody that the SD test case are successfully passed.

Name	Status	File/Iteration	Line/Progress
SD_tc_0	PASSED		
SD_tc_0	PASSED	1	100% (3/3)
Detailed Assertion Information			
Initial	PASSED	DummyDriver_of_LED.c	78

Figure 83 - SD test case passed

We can also show as SD by right click to see if every step in the Scenario are green meaning that all is successfully passed.

Name	Status	File/Iteration	Line/Progress
SD_tc_0	PASSED		
SD_tc_0	PASSED	1	100% (3/3)
Detailed Assertion			

Figure 84 - Show as SD

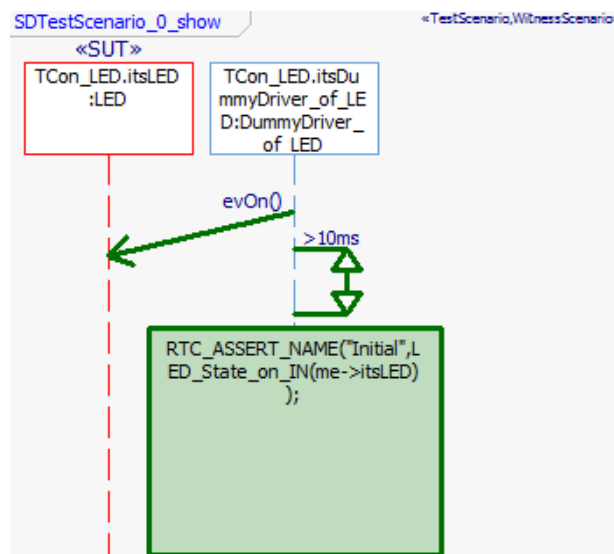


Figure 85 - Verification when Show as SD



First, the event has been sent, we wait the specific time interval and then the assertion has been executing successfully, which means that the expression, and the second argument became TRUE.

If the scenario of test execution shows color blue, it means that the test step has not been performed, and if it is red, test step is failed.

Like the code test case, we can have the TestCase result on a html format page.

## TestCase Result

TestCase: SD\_tc\_0

Friday, February 10, 2017 11:38:57

Environment Information	
Test executed on machine:	DELL
Test executed by user:	Isabelle
Used operating system version:	Windows 8 / Windows 8.1
Used Rhapsody version:	8.1.5, build 9728113
Used TestConductor version:	2.6.5, build 4550

Tested Project	
Project:	Project_TestConductor_MCB1700
Active Code Generation Component:	TPkg_LED_Comp
Active Code Generation Configuration:	DefaultConfig

SequenceDiagram used in TestCase	
TPkg_LED::TCon_LED_Architecture::TCon_LED.SD_tc_0::SDTestScenario_0	

Results	
Status:	PASSED
Progress:	100% (3/3)

Detailed Assertion Information	
Initial	PASSED

Result Verification	
Result verification successful	

Figure 86 - Result of SD Test Case



## Conclusion

The advantage of the use of Test Conductor in a project is to make sure that the model of the project work correctly if we test this. Each project have to be tested either by manual execution but we can have the problem on how we can test regulary and periodically and how document the results of the testing or use a test tool like Test Conductor with we can do test with the formal execution and we can do this periodically on every changes on the model and we can get the result repot as part of the model.

Test Conductor is one possibily to test a project on the software Rhapsody. It is a modelling based testing tool that is part as a plugin in Rhapsody and can only be used for modelling base testing of the Rhapsody models.



## Bibliography

- [1] BTC Embedded Systems AG: IBM® Rational® Rhapsody® Test Conductor Add On – User Guide [online], Release 2.6.2, 2014:  
[http://www.ibm.com/support/knowledgecenter/en/SSB2MU\\_8.1.2/com.btc.tcatg.user.doc/pdf/RTC\\_User\\_Guide.pdf?view=kc](http://www.ibm.com/support/knowledgecenter/en/SSB2MU_8.1.2/com.btc.tcatg.user.doc/pdf/RTC_User_Guide.pdf?view=kc)
- [2] Webinar: BTC Embedded Systems AG [video], the 08.11.2016:  
<http://www.willert.de/produkte/test-debugging-und-qualitaetssicherung/ibm-rational-testconductor/>
- [3] Test Conductor Testing Cookbook: IBM Knowledge Center [online], Release 2.6.5, 2016:  
[http://www.ibm.com/support/knowledgecenter/SSB2MU\\_8.1.5/com.btc.tcatg.user.doc/topics/com.btc.tcatg.tccook.doc.html](http://www.ibm.com/support/knowledgecenter/SSB2MU_8.1.5/com.btc.tcatg.user.doc/topics/com.btc.tcatg.tccook.doc.html)
- [4] Software Testing fundamentals: [online], 2016:  
<http://softwaretestingfundamentals.com/>
- [5] UML Getting started IBM Rational Rhapsody: Marco Matuschek, Willert Software Tools GMBH, v11[online]: <http://www.willert.de/assets/Download/UML-Getting-Started-Rhp-V11.0-en.pdf>
- [6] UML Getting started: Demo DVD, Willert Software Tools GMBH [online]:  
<http://www.willert.de/service/uml-getting-starteds/>
- [7] BTC Embedded Systems AG: Rhapsody in C Tutorial for IBM® Rational® Rhapsody® TestConductor Add On [online], 2015:  
[https://www.ibm.com/support/knowledgecenter/SSB2MU\\_8.1.3/com.btc.tcatg.user.doc/pdf/TestConductor\\_Tutorial\\_C.pdf](https://www.ibm.com/support/knowledgecenter/SSB2MU_8.1.3/com.btc.tcatg.user.doc/pdf/TestConductor_Tutorial_C.pdf)



## Glossary of definitions and acronyms used

**ANSI:** American National Standards Institute

**ARM:** Advanced RISC Machine (**RISC:** Reduced Instruction Set Computer)

**ATG:** Automatic Test Generator

**BB:** Black Box

**e.g.:** exemplum gratia (Latin) = for example

**GB:** Grey Box

**GUI:** Graphical User Interface

**i.e.:** id est (Latin) = that is

**IBM:** International Business Machines

**IDE:** Integrated Development Environment

**MDK:** Microcontroller Development Kit

**OMG:** Object Management Group - <http://www.omg.org/>

**OMD:** Object Model Diagrams

**RQM:** Rational Quality Manager

**RTC:** Rhapsody Test Conductor

**RXF:** Realtime eXecution Framework

**SD:** Sequence Diagram

**SUT:** System Under Test (from a Unit Testing)

**TC:** Test Conductor

**UI:** User Interface

**UML:** Unified Modeling Language

**WB:** White Box

**WST:** Willert Software Tools