Analysis of Model-based Testing methods for Embedded Systems

Mehrdad Bagheri
Abstract

Analysis of Model-based Testing methods for Embedded Systems

Mehrdad Bagheri

The work presented in this master’s thesis is a part of the Artemis-MBAT project. MBAT will provide European industry with a new leading-edge Validation and Verification technology in the form of a Reference Technology Platform (RTP) that will enable the production of high-quality and safe embedded systems at reduced cost in terms of time and money [1]. Model-Based Automated Testing is a new technique which is used for automating the generation of test cases from systems/software requirements. Despite handcrafted tests, the test suite could be derived automatically in this approach by focusing on the model behaviors. The goal of this thesis is to analyze and prototyping a tool where the scope is limited to analyzing the given Timed Automata model as an input, and generating the test suite accordingly.

The output is supposed to be used with Enea Farkle’s Test-bench as an input. Farkle Testbech has been implemented by Enea already and has been integrated into the other Enea tools used for debugging the embedded systems.
Acknowledgements

I would like to thank my thesis supervisor Dr. Kai Lampka for the continuous support of my thesis, for his patience, motivation, enthusiasm, and immense knowledge.

My earnest thanks to Dr. Justin Pearson, my reviewer for undertaking my Masters thesis, and providing his support and valuable comments which led to the betterment of this project.

Finally, I must express my very profound gratitude to my parents and to my lovely wife for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.
# Contents

1 Introduction .................................................. 1
   1.1 Background ................................................. 1
   1.2 Related work ............................................... 1
   1.3 Problem Statement ........................................... 2
       1.3.1 The input of the test generating tool ............... 2
       1.3.2 The testing coverage .................................. 2
       1.3.3 Testing time .......................................... 3
   1.4 Outline .................................................... 3

2 Software testing .............................................. 5
   2.1 What is software testing? ................................. 5
   2.2 Why Testing? ................................................ 5
   2.3 Testing artifacts ............................................ 6
   2.4 Testing terminology ......................................... 7
   2.5 Testing approaches .......................................... 8
   2.6 Testing Levels Based on Software Activity ............... 9
   2.7 Software testing activities ............................... 11
   2.8 Summary .................................................... 13

3 Model-Based Testing ......................................... 15
   3.1 Model-Based Testing Approaches ......................... 15
   3.2 Model-based Testing process ............................. 16
   3.3 Modeling techniques ....................................... 18
       3.3.1 Contract-like Specifications ......................... 18
       3.3.2 Abstract Data Types ................................. 19
       3.3.3 Process Algebras ..................................... 19
       3.3.4 Labeled Transition Systems ......................... 19
       3.3.5 (E)FSM, State Charts ................................. 19
       3.3.6 Hybrid Systems ....................................... 19
       3.3.7 Timed Automata ....................................... 19
   3.4 Benefits of model-based testing ......................... 20
       3.4.1 SUT Fault Detection ................................. 20
       3.4.2 Reduced Testing Cost and Time ..................... 20
CONTENTS

3.4.3 Improved Test Quality ........................................ 20
3.4.4 Requirements Defect Detection ................................. 21
3.4.5 Traceability ......................................................... 21
3.4.6 Requirements Evolution ......................................... 22
3.5 Limitations of model-based testing ............................. 23
  3.5.1 Model-Based Testing Taxonomy ............................... 23
    3.5.1.1 Model Subject ........................................ 23
    3.5.1.2 Model Redundancy level ................................ 24
    3.5.1.3 Model Characteristics .................................. 24
    3.5.1.4 Model Paradigm ......................................... 25
    3.5.1.5 Test Selection Criteria .................................. 25
    3.5.1.6 Test Generation Technology .............................. 26
    3.5.1.7 On-line or Off-line Test Generation .................... 26
3.6 Classification of Approaches and Tools ......................... 26
3.7 Summary ............................................................ 28

4 UPPAAL and Timed automata ........................................ 30
  4.1 Timed Automata .................................................. 30
    4.1.1 Timed Automata Formal Syntax ............................. 30
    4.1.2 UPPAAL Timed Automata ..................................... 31
    4.1.3 UPPAAL Expressions ......................................... 33
    4.1.4 UPPAAL Query Language ..................................... 34
    4.1.5 Time in UPPAAL .............................................. 35
    4.1.5.1 Time languages ......................................... 35
    4.1.5.2 Clock Regions ........................................... 35
    4.1.6 Determinism in Timed Automata ............................ 37

5 Automatic Test Generator Tool .................................... 39
  5.1 Tool Specification .............................................. 39
  5.2 Farkle OSE Test Executer Framework ........................... 41
  5.3 Test generation preliminaries .................................. 41
    5.3.1 Test trace .................................................. 42
    5.3.2 Deterministic input models ................................ 44
    5.3.3 Testing the time requirements .............................. 44
    5.3.4 Test environment model ..................................... 45
    5.3.5 Integrating UPPAAL Verifier ................................ 45
  5.4 Test generator tool ............................................... 46
    5.4.1 Preparing the environment model ........................... 46
    5.4.2 Preparing the test model ................................... 47
    5.4.3 Implementation ............................................... 48
    5.4.3.1 GUI ..................................................... 48
    5.4.3.2 Full node coverage negated query generator ............ 49
    5.4.3.3 Environment model generator ............................ 50
    5.4.3.4 Query executor ........................................... 51
CONTENTS

5.5 Use Case . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51

6 Conclusion and Future Works 55
   6.1 Future work . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
   6.2 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
       6.2.1 Answer for the questions in the problem statement . . 55
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Software development activities and testing levels</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>The activities of a tester for testing a software taken from [1]</td>
<td>11</td>
</tr>
<tr>
<td>3.1</td>
<td>Model-Based testing process</td>
<td>17</td>
</tr>
<tr>
<td>3.2</td>
<td>Traceability among requirements, model, and tests.</td>
<td>22</td>
</tr>
<tr>
<td>3.3</td>
<td>The taxonomy of the Model-Based Testing</td>
<td>24</td>
</tr>
<tr>
<td>3.4</td>
<td>Torx tool as an example of classification</td>
<td>27</td>
</tr>
<tr>
<td>4.1</td>
<td>An example of Timed Automata.[9]</td>
<td>32</td>
</tr>
<tr>
<td>4.2</td>
<td>Syntax of expressions in BNF.[14]</td>
<td>34</td>
</tr>
<tr>
<td>4.3</td>
<td>Example of clock region.[16]</td>
<td>36</td>
</tr>
<tr>
<td>4.4</td>
<td>Example of clock region.[14]</td>
<td>37</td>
</tr>
<tr>
<td>4.5</td>
<td>Non-deterministic TAs : a) w.r.t the time and b) w.r.t the actions</td>
<td>37</td>
</tr>
<tr>
<td>5.1</td>
<td>The tool-chain work-flow</td>
<td>40</td>
</tr>
<tr>
<td>5.2</td>
<td>Farkle framework.[17]</td>
<td>41</td>
</tr>
<tr>
<td>5.3</td>
<td>The paths of a simple graph</td>
<td>42</td>
</tr>
<tr>
<td>5.4</td>
<td>A complex model</td>
<td>43</td>
</tr>
<tr>
<td>5.5</td>
<td>The schematic of the test generator tool</td>
<td>46</td>
</tr>
<tr>
<td>5.6</td>
<td>The test model</td>
<td>48</td>
</tr>
<tr>
<td>5.7</td>
<td>The GUI of the test generator tool</td>
<td>49</td>
</tr>
<tr>
<td>5.8</td>
<td>Simple lamp</td>
<td>52</td>
</tr>
<tr>
<td>5.9</td>
<td>Simple lamp and the relevant environment</td>
<td>52</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Background

This master thesis was performed at ENEA Services AB. The work is part of the ARTEMIS\(^1\)/MBAT\(^2\) project whose purpose is to provide European industries with a new leading-edge verification and validation technology in the form of a Reference Technology Platform (RTP) that will enable the production of safe, high-quality embedded systems at reduced cost in terms of time and money.

A model describes how user actions and system states in a system are related to each other. If the model is exhaustive enough, describing every action and system response, then effective test cases can be derived from the model. This technique is known today as model-based testing.

There are two tools for debugging and testing embedded systems, which are provided by Enea: Optima and Farkle. Of course, there is another tool called Testbench which is a GUI for Farkle. These tools are now integrated and communicate internally. Optima is a suite of powerful system debug, profiling, and tracing tools for the Embedded software which run under Enea OSE RTOS\(^3\). It provides a complete IDE\(^4\) that can cover the whole life cycle of embedded systems. Farkle is another tool for executing test cases on the system.

1.2 Related work

During the last two decades, a lot of research and effort has been done for automating the model-based test generation process e.g. TVEDA [28] is a tool that was developed by research and development centre of France

\(^1\)www.Artemis.com
\(^2\)Project Full Title: Combined Model-based Analysis and Testing of Embedded Systems
\(^3\)Enea OSE is an RTOS optimized for distributed, fault-tolerant systems.
\(^4\)Integrated software development environment.
1.3 Problem Statement

Telecom where the main goal was to support automatic conformance of protocols, the AGEDIS ToolSet [30] is a tools for automated model driven test generation and execution for distributed systems, TGV [29], TestGen [31], and TorX [11] are online, on-the-fly testing tools that implement ioco relations.

The description of specifications in this approach can be in the formal languages, Lotos [19], Promela [20] or Fsp [21], or directly as a transition system in the Aldebaran format [11]. In [12], the tool TTG is proposed for black-box conformance testing of real-time systems. The main characteristic of TTG is its ability for online, on-the-fly testing of the non-deterministic real-time systems. Two separate models are used for the system under test (SUT) and for environment testing. Furthermore because, their timing approach, implemented as both a digital and an analogue clock, TTG can test both discrete and continuous systems. There is another tool called UPPAAL-TRON [13] that has mainly the same characteristics as TTG, in the sense that it uses an online, on-the-fly approach, uses two separate models, but it uses a continuous clock.

1.3 Problem Statement

The existing tools, and Optima work properly now. Like the other typical testing tools, Farkle uses test cases which are produced manually. The effort in this master thesis is to analyze the methods for implementing a prototype tool which can generate effective test cases for Farkle GUI automatically. This thesis work is planned to find the solutions for the questions in the following section.

1.3.1 The input of the test generating tool

The test generating prototype tool will have a timed automata model as the input which can be analyzed for generating test cases.

Q1. What kind of tools, applications, and methods are required to analyze the given model?

1.3.2 The testing coverage

The Uppaal model is a finite state machine, so it is important for the test generator tool to provide the user with different kinds of test coverage, such as node coverage, edge coverage, etc.

Q2. Is it possible to have different kinds of test coverage in the test generator?

---

5 Farkle is a test support framework which is developed by ENEA to target the OSE platform
6 Optima is a debug tool which is developed by ENEA to target the OSE platform
1.3.3 Testing time

Since the purpose of the test generator tool is to generate test cases for applications which work under OSE RTOS, one of the important features of such applications to be tested is the time constraints. However, for testing a continuous range of time, an infinite number of test cases is required.

Q3. How to test a continuous time range?

1.4 Outline

This master’s thesis project will be split into two parts. The first part will consist of an academic report of previous studies on the Optima and Farkle tools, of secondary literature on modeling methods, and of several test case writing methods. The second part will consist of designing and implementing the test generator tool using the knowledge collected in the first part.
Chapter 2

Software testing

Since the main goal of this master’s thesis is to analyze the model based test generation methods, it is necessary to have information about testing concepts and definitions. In the current chapter, the concepts required to have a better understanding of testing are briefly described.

2.1 What is software testing?

In ANSI/IEEE 1059, software testing is defined as follows:

“[T]esting is the process of analyzing a software item to detect the differences between existing and required conditions (that is defects/ errors/ bugs) and to evaluate the features of the software item”. [32]

2.2 Why Testing?

Beizer [2] describes the goals of testing levels based on test process maturity:

- Level 0: There is no difference between testing and debugging.
- Level 1: The purpose of testing is to show that the software works.
- Level 2: The purpose of testing is to show that the software does not work.
- Level 3: The purpose of testing is not to prove anything specific, but to reduce the risk of using the software.
- Level 4: Testing is a mental discipline that helps all IT professionals develop higher quality software.
2.3 Testing artifacts

Level 0 assumes the software testing the same as debugging, therefore the testing process is restricted to tracing the code and trying a few different input values.

But, in the rest of levels where the goal is to check the performance and quality of the software, automating the test generation can play an important role. It can decrease the efforts significantly, and increase the test coverage as well.

2.3 Testing artifacts

The software testing process may produce several artifacts. This section provides a brief definition for each.

Test case In [3], there are several definitions for a test case, which are listed as follows:

IEEE Standard 610 (1990) defines test case as follows

- (1) A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.

- (2) (IEEE Std 829-1983) Documentation specifying inputs, predicted results, and a set of execution conditions for a test item.

According to Ron Patton, “Test cases are the specific inputs that you’ll try and the procedures that you’ll follow when you test the software” (2001, 65).

Boris Beizer defines a test case as “A sequence of one or more sub tests executed as a sequence because the outcome and/or final state of one subtest is the input and/or initial state of the next. The word “test” is used to include subtests, tests proper, and test suites” (1995, 3).

Bob Binder explains that “A test case specifies the pretest state of the IUT\(^1\) and its environment, the test inputs or conditions, and the expected result. The expected result specifies what the IUT should produce from the test inputs. This specification includes messages generated by the IUT, exceptions, returned values, and resultant state of the IUT and its environment. Test cases may also specify initial and resulting conditions for other objects that constitute the IUT and its environment” (1999, 47).

\(^1\)Implementation Under Test
2.4 Testing terminology

As defined above, a test case describes a complete scenario for checking a specific behavior of a test object. It can be presented as a piece of code, a flow-chart, a document file, or other method of information depiction.

In fact, a test case is a purposive set of actions taken to check a specific behavior of a test object. These set of actions includes providing the test object with the required inputs, triggering the test object to behave as expected, and checking the behavior of the test object generally.

**Test script**  A test script aims to create a testing program to test a test object. It usually contains a set of instructions delineating acceptable and unacceptable results.

**Test suite**  A test suite is a set of test cases. It may contain more detailed instructions or aims for each set of test cases. It can also contain the tester description about the required configurations used during the test.

**Test data**  The collection of values are used to test the functionality of a system. It is used to confirm that the given values used as the input(s) to a specific part of the software (e.g. a function) will cause the expected results.

**Test harness**  It is a framework which aims to automate the execution part of a test suite. It involves software and test data for testing a program or smaller parts of program by running it in different environment and monitoring the output(s).

2.4 Testing terminology

For studying software testing, software testing terminology is an important prerequisite. The terminology of the software testing can be found in several standards, such as the IEEE Standard Glossary of Software Engineering Terminology [33], DOD-STD-2167A and MIL-STD-498 from the US Department of Defense [34], and the British Computer Society’s Standard for Software Component Testing[34]. A set of important terms in software testing which are taken from the standards are listed as follows:

- **Validation and Verification** IEEE Standard 610 (1990) defines verification and validation as follows.
  - Validation: The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.
  - Verification: 1) The process of evaluating a system or component to determine whether the products of a given development phase
satisfy the conditions imposed at the start of that phase and e. 2) formal proof of program correctness.

- **Software defect**: A flaw in a system which can cause it to fail.
- **Software fault**: A static defect in the software.
- **Software error**: A human mistake that can lead the system produce the wrong results.
- **Software failure**: Deviation of a system with respect to requirements

- **Testing environment**: A software and hardware setup which provides the tester(s) with required environmental resources. The setup can contain the hardware devices, the operating system(s) and the other softwares necessary to run the SUT.

- **Testing and debugging** According to IEEE 610 (1990):
  
  - **Testing**
    
    * 1) An activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component.
    
    * 2) To conduct an activity as in 1)
  
  - **Debugging**: According to IEEE 610 (1990): "To detect, locate, and correct faults in a computer program. Techniques include use of breakpoints, desk checking, dumps, inspection, reversible execution, single-step operation, and traces."

- **Software observability**: The extent of the possibility to observe the components, outcomes, and behaviors of a system.

- **Software controllability**: The extent of possibility to generate the required inputs for a program.

- **Black-box testing**: Testing a system with respect to its requirements and specifications.

- **White-box testing**: Testing a system with respect to the internal structure of the program.

### 2.5 Testing approaches

In [2], three main approaches for testing are mentioned: structural testing, functional testing, and correctness proofs.
2.6 Testing Levels Based on Software Activity

Structural testing is an approach for testing the internal structure of software, and it is also known as clear box testing, glass box testing, transparent box testing, and white-box testing as well. In this approach, the tester should provide enough test cases to guarantee that all the paths in the routine can be traversed at least once. Due to the infinite loops and number of possible paths in a routine, it is practically impossible to have a complete test of the software by using a purely structural testing that can assure bug-free software.

Functional testing is a kind of black-box testing based on the specification of the software. All programs accept a finite number of inputs and produce related outcomes. In the functional testing approach, the program is tested by giving all the possible inputs and examining the outcomes. By applying the inputs, three possible scenarios may happen for the program:

- It accepts the input(s) and produces the right outcome, meaning, it passed.
- It accepts the input(s) and produces the wrong outcome, that meaning, it failed.
- It rejects the input(s) and produces a relevant message which can be considered as an outcome.

It is practically impossible to have complete functional testing of a software program in the sense that the number of inputs and their compositions are too high for testers and even for input generator tools. However, the structural testing approach may help us to limit the number of test cases. For instance, limitations of the inputs within the structure of the program can limit the number of possible inputs.

Correctness proofs are based on both structural and functional testing concepts. The requirements are expressed in a formal language (e.g., mathematics). Each statement should be tested and utilized in the inductive proof steps. These proofs are used in numerical applications or critical applications (e.g., system’s security kernel or a part of compilers) [2], and they are expensive practically.

2.6 Testing Levels Based on Software Activity

Testing activities can be classified into five levels based on software activities. Each level of testing in this classification has its relevant software development activity, in which test cases can be derived [1], and are as follows:
2.6 Testing Levels Based on Software Activity

- Acceptance Testing: Test the software regarding the requirements.
- System Testing: Test the software regarding the architectural design.
- Integration Testing: Test the software regarding the subsystem design.
- Module Testing: Test the software regarding the detailed design.
- Unit Testing: Test the software regarding the implementation.

Figure 2.1: Software development activities and testing levels

Figure 2.1 shows the V-Model depicting a typical procedure for developing and testing software. It illustrates how the various software development and testing levels are related. In fact, it is recommended to have testing concurrently with each software development level, since the early detection of defects can reduce the final cost. Each development level and its relevant test level are described briefly as follows:

The Requirement Analysis part of a software development plan defines the customer’s needs, and Acceptance Test checks whether the completed software meets the respective requirements.

The Architectural Design The Architectural Design part of a software development plan selects the components and connectors that together form a system which can meet the requirements, and System Test checks whether the assembled system meets the respective requirements.
The *Subsystem Design* part of a software development plan specifies the subsystems, and *Integration Test* checks the subsystem relations.

The *Detailed Design* part of a software development plan defines the structure and behavior of each module, and *Module Test* checks each module individually.

The *Implementation* is the final part of a software development plan, which produces the code, and *Unit Test* assesses the units produced in this part [1].

### 2.7 Software testing activities

In [1], an IT professional who is responsible for the activities of the testing is called a Test engineer. The test activities for a test engineer are defined thus: design the required inputs, generate test values, execute the test scripts, examine the results, and report the result to developers and managers. A test manager’s responsibility is to set test strategies, interact with the other managers, or help the test engineers.

![Diagram of software testing activities](image)

*Figure 2.2: The activities of a tester for testing a software taken from [1].*

Figure 2.2 shows the above mentioned activities. Such activities can be performed by one or more test engineers, who are supervised by a test manager. An executable test should be derived from requirements. The
2.7 Software testing activities

Software testing tests, which are denoted as P in the figure, will be executed against the implementation to evaluate the corresponding results [1].
2.8 Summery

Due to possible mistakes in the development of any product, testing can play an important role in any development process. The complexity of software development makes it an especially error prone process.

In each level of software development, there is a relevant test level, which detects the faults of the corresponding development level. The Acceptance Testing level is a related test for requirement analysis of the software development level. System Testing is for architectural design, Integration Testing for subsystem design, Module Testing for detailed design, and Unit Testing, which the lowest level, is performed for the implementation level of software development.

Software testing, however, falls into two main groups: 1) Black-box testing, which tests the functionality of the software, and 2) White box testing, which points to the internal structure of software parts.
Chapter 3

Model-Based Testing

Software testing is truly important in the software development process, and it generally takes between 30 and 60 percent of the overall development effort. Nowadays, companies are already using different tools to automate the test execution to save a big portion of the time, which is normally spent on setup and executing the manual tests.

Model-based testing pushes the level of automation even further by automating the test design. In addition, the test data, test cases, and test scripts are derived from a model in this type of testing. In this chapter, different model-based testing approaches and techniques are briefly described. Moreover, at the end of chapter, the benefits and limitations of using model-based testing are listed.

3.1 Model-Based Testing Approaches

There are four main testing approaches which are known as model-based testing. The approaches are described as follows [4].

- Approach 1 provides the test input data from a domain model\(^1\). The test data should be derived from the model using a smart algorithm to select a set of data and their compositions to cover all the possible inputs. For instance, a pairwise algorithm\(^2\) for a procedure which accepts three inputs: A:1, 2, 3, B:'a', 'b', 'c' and C:0.25 0.5, 0.75, will generate nine different combinations. This approach can only provide the tester with automatic test data generation, and it can’t be used to design a complete test, since it can not determine the result of the test.

\(^1\)Domain model is a model which contains the information about the domains of the input values.

\(^2\)http://www.pairwise.org/
3.2 Model-based Testing process

- Approach 2 provides test cases from an environment model. The environment model describes the expected environment of a SUT, for example, a temperature control system [25]. Like the previous approach, since the expected behavior of a SUT is not defined, it is not possible to know whether the test has passed or failed.

- Approach 3 provides executable test cases based on the expected behaviour of the SUT. Test cases consist of oracle information\(^3\), like the expected outcomes. It provides the tester with the possibility of comparing the test results with the oracle information to decide whether the test passed or failed.

- Approach 4 provides the test scripts from abstract tests\(^4\). The main focus of this approach is to convert a given a test case abstraction into a low-level test script. The required information for converting a high-level call into the test scripts.

The approach of this thesis is to follow the third approach of model-based testing, since it’s supposed to implement a tool prototype to generate suitable and exhaustive test cases for a given model as its input. In [4], model-based testing is considered as the automation of the design of black box tests, but it differs from usual black box testing. The difference is that usual black box test is based on the requirements and model-based testing is based on the model of the SUT which contains the requirements. Then, test case can be derived from the model automatically using model-based testing tools.

### 3.2 Model-based Testing process

In model-based testing, the test design process can be performed automatically. The test designer can design an abstract model of the SUT and then, test cases can automatically be derived from the model. There are five main steps to generate test cases using model-based testing [4].

1. Design a model of the SUT or the environment.
2. Generate an abstract test.
3. Make the test executable by concretizing the abstract test.
4. Run the test and give the verdict.

---

\(^3\)An oracle is a mechanism used by software testers and software engineers for determining whether a test has been passed or failed.

\(^4\)A model which describes the SUT is an abstract model and test cases derived from this model are on the same level of abstraction as the model. These test cases are known as abstract tests[22].
5. Check the test results.

Figure 3.1 shows the five main steps of model-based testing. In the first step, the main task is to prepare an abstract SUT. The model should contain the main features of the SUT without the details. In order to test the generated model to ensure that it has the expected behaviors, it is recommended to use the available tools which are usually provided by the modeling notations (e.g., type-checkers and static analysis tools as the automated verification tools and animators as interactive ones).

In the second step, the abstract tests should be derived from the abstract model. The main task is to define test selection criteria to reduce the number of possible test cases. Additionally, the model coverage criterion should be specified. This step can have several outputs. The main output is the abstract test, which is derived from an abstract model and is not executable. The other outputs include the requirements traceability matrix or the coverage reports. The requirements traceability matrix [4] determines the functional requirements and produces test case relations. These relations are many to many i.e. a requirement can be tested by more than one test cases, and a test case may cover several requirements. The coverage reports measure the test coverage of the model (not the SUT), and they can be used for statistical feedback. The reports can help to indicate the part of the program which is not well tested and the reason for this problem.
3.3 Modeling techniques

The main task of the third step is to convert the abstract tests into executable concrete tests. The first method for this conversion is to use transform tools, but it can also be done using adaptors. The second and third steps can be done in one step as well. The advantage of this separation, however, is to have the test cases language independent, so they can easily be translated to any scripting language by changing the adaptor or converting tool.

In the fourth step, the concrete executable test scripts should be run. Mainly, there are two different ways of executing the test scripts, online and offline, which will be described later. Both online and offline methods can be done in this step. In online model-based testing, the test cases will be run as they are generated. For non-deterministic systems, it is easier to use online testing, and in some cases it is necessary, because the current state and path in the SUT can be recognized. With offline testing, the generated test scripts can be executed many times in any environment, and the results can be recorded.

Finally, the last step is to compare the test results with the expected outputs provided by the test scripts and provide the verdicts.

3.3 Modeling techniques

There are several different types of modeling techniques. Different modeling techniques result in the various models in which the system is described differently. This section gives a brief overview of different techniques for modeling the SUT.

3.3.1 Contract-like Specifications

Contract-like Specifications [5] are also known as state-based specification languages. The main idea is to have an explicit specification of the behaviour of each component. A contract is a set of statements that describe precisely the aspects of each component. There are three main types of contract which are described as follows:

- **Invariant**, is a guard, which should be assigned to a type for all instances of the type. For instance, it maybe a non-zero condition for an attribute.

- **Precondition**, states that the requirements of any operation call, e.g. the, must not overflow condition for a sum operation.

- **Postcondition**, is a condition that must be true after an operation. For instance, the result of a factorial operation should be equal or greater than one.
3.3 Modeling techniques

3.3.2 Abstract Data Types

Abstract data types (ADTs) are mathematical specification methods for indicating the relationships of the operations. The ADTs are theoretical and are used to abridge abstract algorithms, to categorize and assess the data structures, and to express the types of programming languages. There is no information about the internal behaviours of the object [5].

3.3.3 Process Algebras

Process Algebras (Process calculi) are a set of approaches for studying distributed and concurrent systems. In these approaches, equational and inequational reasoning are used to analyze the behaviour of such systems. A process algebra is a mathematical structure. It has been used in automata theory development, in which a process is modeled as an automaton [36].

3.3.4 Labeled Transition Systems

A labeled transition system is essentially a directed state machine with labels on the edges. The labels are the actions between the states, and the states represent the system states. Labeled transition systems form a powerful semantic model for arguing about the processes, such as specifications, implementations, and tests [7].

3.3.5 (E)FSM, State Charts

FSM is a finite number of states which represent the internal status of the machine and are connected using the transitions. Transitions allow the system to move from one state to another in reaction to the defined input(s) or event(s). Moreover, transitions can generate the output(s) or event(s). The FSMs have been previously used in model-based testing [23].

3.3.6 Hybrid Systems

Hybrid systems is a study of systems which include the relationship between discrete events and continuous time dynamics. Today, the use of computer control systems is becoming ubiquitous, and hybrid systems have become prominent, especially in safety-critical applications [24].

3.3.7 Timed Automata

Timed automata is a finite number of states connected with a finite number of labeled edges (finite automaton) which is extended with a real valued time. The time variable can be reset to zero and increased during the related action. In timed automata, time constraints can be represented by guards on edges to straighten the behaviours [16].
3.4 Benefits of model-based testing

The abovementioned modeling techniques model a system differently. Contract-like specifications model the effects of the individual system operations of an API. The abstract data-type defines an API via the algebraic properties of its operations in a more abstract way. Process algebras label the transition systems, and EFSMs, State Charts, hybrid systems, and timed automata model the intended behaviors of the system using mathematical expressions or graphical state charts.

In this thesis, timed automata are used to model the SUT, which is a real-time embedded software. The model consists of a network of timed automata in which the desired behaviors of the SUT and test data are described.

3.4 Benefits of model-based testing

Using model-based testing as a testing technique has several benefits which are described as follows:

3.4.1 SUT Fault Detection

Fault detection in SUT is the main purpose of testing. The number of faults that manual testing can detect depends on the test designer. However, modelling a system can give a better understanding of the system, which of course is also helpful for testing the system, but it is not possible to say that the tests which are generated from a model can detect more faults than manual test cases, since the model designer’s experience and knowledge greatly effects the tests derived from the model [4].

3.4.2 Reduced Testing Cost and Time

The total time and effort, which is usually spent generating tests from a model, is less than the time and effort spent designing and maintaining tests. Also helpful in the analyzing phase, there are some model-based testing tools which can make failure analysis easier by finding the shortest path which caused the failure [4]. Because of the nature of model-based testing, which involves the tester earlier in the development process, it can detect faults earlier, and early fault detection reduces the time spent.

3.4.3 Improved Test Quality

Model-based testing improves some of the usual problems in manual testing. Here is a list of the improvements of model-based testing as follows:

- It is based on algorithms and is systematic, which makes the process repeatable.
- It is possible to generate many more test cases than manual testing.
3.4 Benefits of model-based testing

- To generate more test cases, it is necessary to change the selection criteria.
- The lower cost for generating more executable tests is the computing time of generating the input data and oracles from the model.

3.4.4 Requirements Defect Detection

Model-based testing should derive the tests from a model of the system and detect defects in the SUT. One of the advantages of model-based testing is to detect the defects in the informal requirements, which contain unclear parts. Models should have accurate semantics, and during modeling, the issues of the requirements should be revealed. Detecting these problems is the main advantage of the model-based testing approach [4].

3.4.5 Traceability

The IEEE Standard 610 (1990) defines traceability as:

- 1) The degree to which a relationship can be established between, two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; for example, the degree to which the requirements and design of a given software component match.

- 2) The degree to which each element in a software development product establishes its reason for existing; for example, the degree to which each element in a bubble chart references the requirement that it satisfies.

Traceability makes it possible to link all the test cases to the model, selection criteria, and also to informal requirements. It is useful to describe the test cases and the reasons for generating them. Moreover, it can be helpful to test the part of the model, which is affected after changing the model.

Figure 3.2 shows the three features of traceability, model-tests, the reqs-model, and reqs-tests. The model-test describes the communication between model elements and test cases by recording the contribution of each part of the model to each test case [4].

The more challenging feature is the reqs-model, which is a matrix showing how the model and the informal requirements are related to each other. It can answer the following requests:

- Find the missing requirements of the model.
- Find the affect of a particular requirement.
3.4 Benefits of model-based testing

Model-Based Testing

Find the reason for the given transition or prediction in the model and the related requirements.

The last feature of traceability, Reqs-Tests, is a combination of the previous two features and reveals the relationship between informal requirements and the derived test cases with the purpose of performing the following tasks:

- Realize the untested requirements.
- Illustrate all the related tests of the given requirements.
- Illustrate all the related requirements of the given test.

3.4.6 Requirements Evolution

In manual testing after any change in the requirements of SUT, it takes a lot of effort to change the test cases with respect to the changes of the requirements. In the model-based approach, however, it is enough to update the model. Deriving the cases again takes less effort, because the model of the SUT is usually smaller than the test suite. It is also usually a good idea to provide incremental traceability, which is applicable in all three features. For instance, the incremental reqs-test traceability can be used to realize changes between the old model and the new (changed) model and to report the test cases which are no longer useful and consequently, it can classify test cases into four groups: deleted, unchanged, changed, and added.
3.5 Limitations of model-based testing

Like the other kinds of testing, the basic limitation of the model-based testing approach is that there is no guarantee about covering all the possible differences between the model and the IUT even though it tries an infinite number of test cases. The next limitation is the need for designers with some manual testing knowledge, along with the ability to abstract and design a model, in addition to expertise in application areas that may require training costs. Furthermore, the usage of model-based testing is limited to functional testing. Moreover, the need to have a good level of test maturity due to the complexity of model-based testing is another limitation. At the starting point of model-based testing, there are number of possible “pain” factors which are listed as follows: [4]

- **The requirements are not updated**: The out-of-date informal requirements cause this problem. This can lead to generating the wrong model based on the wrong requirements.

- **The usage of model-based testing is not suitable**: Modeling some part of the SUT is sometimes more complex and difficult, in which case, it is more logical and quicker to use manual testing. Sometimes, however, model-based testing needs to be tried to determine its best use.

- **Required time to analyse the failed tests**: Analyzing the cause of the failed tests is a part of testing. The test failure may be caused by the SUT, the adaptor code, or the model. Due to the complexity of model-based testing, it is not easy to analyze the failed tests.

- **Choosing the inappropriate metrics**: A common measure for manual tests is the number of test cases, but there are several possible measurements like code coverage, requirements coverage, and model coverage.

3.5.1 Model-Based Testing Taxonomy

In [10], model-based testing is classified into seven different dimensions, where each dimension can have some possible instantiations. Figure 2.5 shows the taxonomy of model-based testing [10]. This section describes each dimension in brief.

3.5.1.1 Model Subject

In this category, the model subject is the expected behaviour of the SUT or the conceivable behaviour of the environment. It follows two main purposes. The first goal is to be an oracle for the SUT for encoding the expected behaviours, and the second goal is for test case generation. It can be also used as a test selection criterion by restricting the possible behaviours of the
3.5 Limitations of model-based testing

Model-Based Testing

Figure 3.3: The taxonomy of the Model-Based Testing

SUT. Moreover, stimuli can define the environment model. In model-based testing, a combination of the SUT and its relevant environment is usually used [10].

3.5.1.2 Model Redundancy level

There are different possible scenarios for using model-based testing, where the difference usually lies in the level of model redundancy between the test model and/or the IUT. For instance, one possible scenario is to have one common model for both testing and the IUT, which means that the tests and the executable codes are derived from the same model. It is also possible to have just one model for testing and to generate the IUT manually [10].

3.5.1.3 Model Characteristics

This dimension specifies the characteristics of the model, such as non-determinism, time inclusion, and continuous/discrete or hybrid types of the model [10]. In real time embedded systems, it is common to have all of the possible characteristics mentioned above.
3.5 Limitations of model-based testing

3.5.1.4 Model Paradigm

The model paradigm describes the notation and paradigm that is used in system modeling. In [10], the model is categorized into six main groups as follows:

- **State-Based (or Pre/Post) Notations** is a set of variables which describes the internal states of the system, like JML and VDM.

- **Transition-based Notations** are the graphical descriptions of state transitions, such as FSM, statement state charts, UML state machine, Simulink Stateflow Charts, Labeled Transition System, and I/O automata.

- **History-based Notations** is a description of the possible traces of the system behaviour over time.

- **Functional Notations** model a system using a set of mathematical functions.

- **Operational Notations** model a system using a set of parallel executable processes.

- **Data-Flow Notations** model the system by focusing on the data.

3.5.1.5 Test Selection Criteria

Test selection criteria is a tool for controlling the test generation. The test selection criteria can be a parameter for classifying the testing tools and approaches. In Figure 3.3, there are some frequent criteria, which are described as follows: [10]

- **Structural Model Coverage Criteria** are for covering the whole structure of the model, such as nodes, transitions, conditions, etc.

- **Data Coverage Criteria** are for covering all the possible test values by selecting a limited number of data from a wide range of data.

- **Requirements-Based Coverage Criteria** are for covering all the model elements by mapping to the SUT requirements.

- **Ad-hoc Test Case Specifications** can be written by the test engineer to control the test process manually.

- **Random and Stochastic Criteria** are used for modeling the probable actions, mostly for environment models.

- **Fault-based Criteria** are used in the SUT to find the faults (mostly to cover and find the mutants).
3.6 Classification of Approaches and Tools

3.5.1.6 Test Generation Technology

This dimension describes the technology which is used in model-based testing. The common technology is to generate a test case from a model of the SUT. Additionally, in model-based testing, there is a possibility to automate the process of test case generation. In the automated process, a test case can be generated manually or by means of model checking, symbolic execution, or deductive theorem proving [10].

3.5.1.7 On-line or Off-line Test Generation

The approach of online testing is to test the SUT based on the actual output of the SUT. It is easier to use online testing for non-deterministic models [10]. In this case, the model-based testing tools should connect to the SUT directly; give the possible inputs or wait for possible delays, observe the outputs, and then depending on the observed output, observe the next path, inputs, or delays, and/or the verdict should be given. Furthermore, the test abstract is a concept and will not be an explicit artifact. In the offline case, the model-based process should be applied to generate the tests and then give the verdict. After generating test cases, it is possible to run them many times, which is one of the advantages of offline testing.

3.6 Classification of Approaches and Tools

It is a good idea to use the taxonomy in [10] to classify the existing approaches and tools. Figure 3.4 shows the Torx [11] tool specification as an example of the classification.
### 3.6 Classification of Approaches and Tools

*Model-Based Testing*

<table>
<thead>
<tr>
<th>Model subject</th>
<th>The model is a behaviour model of the SUT. Some environmental aspects can be taken into account at the level of the model, but also at the level of test selection criteria within the test purposes.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model redundancy level</td>
<td>Dedicated model for test generation was used.</td>
</tr>
<tr>
<td>Model Characteristics</td>
<td>TorX manages non-deterministic, untimed, discrete models.</td>
</tr>
<tr>
<td>Modelling Paradigm</td>
<td>The underlying paradigm used by TorX is that of LTS; compilers from the LOTOS and SPIN modelling languages have been developed.</td>
</tr>
<tr>
<td>Test Selection Criteria</td>
<td>The TorX test generation algorithm is based on a walk through the state space of the specification. This walk can be done randomly or controlled by the test purpose.</td>
</tr>
<tr>
<td>Test Generation Technology</td>
<td>Automated test case generation using on-the-fly state space exploration techniques.</td>
</tr>
<tr>
<td>On line/Off line</td>
<td>Both</td>
</tr>
</tbody>
</table>

Figure 3.4: Torx tool as an example of classification
3.7 Summery

For testing an object, it is possible to prepare a model of the object in which the requirements of the system are specified and then, derive a number of test cases from the model; this approach is called model-based testing. In this approach, the first possible way is to use modeling tools to provide the model and generate test cases manually considering the model, or to use tools to provide the model and use automatic test case generator tools. There are number of modeling techniques which are used for different purposes, and testing with each of them imposes different considerations.

Basically, there are five steps for generating test cases from a model: Design the model, generate an abstract test, generate a test script, execute the tests, and finally, observe the results to give the verdict.

Although, it can differ in online testing in the sense that online testing does not contain abstract test generation and it is just a concept, since each execution of the test can lead to a different path than the test case.
Chapter 4

UPPAAL and Timed automata

In model-based testing, the system specification is a model; therefore, the model plays an important role. The model should have the capability to specify all the requirements. For instance, to test a real-time system, it’s required to time constraints in the model.

Since the goal in this thesis project is to test a real-time application for embedded systems, UPPAAL timed automata (TA) has been chosen because includes the required features for the purpose of the thesis. In this chapter, the definition and features of timed automata will be briefly described, and then UPPAAL timed automata will be discussed.

4.1 Timed Automata

One of the theories used for modeling and verification of real-time systems is timed automata [26]. It is essentially a finite automaton\(^1\) which is extended with a finite number of real-valued clocks. When it starts, the clock variables, which are non-negative values, will be set to zero, and they will be increased during the run with the same speed. There may also be clock constraints, like guards on the edges, and a transaction can be taken when the constraints are satisfied by the relevant values [9].

4.1.1 Timed Automata Formal Syntax

Assume that \(C\) is a set of clocks and \(G(C)\) is the set of conjunctions over the conditions of \(x\geq c\) or \(x-y\geq c\), where \(x, y\in C, c\in \mathbb{N}\) and \(\geq\in \{<,\leq, =, \geq, >\}.

\(^1\)A finite automaton is a mathematical model with a finite number of states and labeled edges which is used to design and describe a system.
4.1 Timed Automata

**Definition 1** A timed automaton is a tuple \((S, s_0, C, P, D, I)\), where:

- \(S\) is a set of states
- \(s_0\) is the initial state, \(s_0 \in S\)
- \(C\) is the set of clocks
- \(P\) is the set of actions
- \(D \subseteq S \times P \times G(C) \times 2^C \times S\) is a set of edges between the states with an action.
- \(I : S \rightarrow G(C)\) assigns invariants to locations

The semantics of a timed automaton A clock valuation is a function in the form of \(u : C \rightarrow \mathbb{R}_{\geq 0}\). Let \(\mathbb{R}^C\) be the set of all clock valuations and \(u_0(x) = 0\) for all \(x \in C\). We will abuse the notation by considering guards and invariants as sets of clock valuations, writing \(u \in I(s)\) to mean that \(u\) satisfies \(I(s)\).

For a timed automaton \((S, s_0, C, P, D, I)\) the semantics are defined as labelled transition system \(\langle L, l_0, \rightarrow \rangle\) where:

- \(L \subseteq S \times \mathbb{R}^C\) is a set of states.
- \(l_0 = (s_0, u_0)\) is the initial state.
- \(\rightarrow \subseteq L \times (\mathbb{R}_{\geq 0} \cup \{A\}) \times L\) is the transition relationship,

such that:

- \((s, u) \xrightarrow{d} (s, u + d)\) if \(\forall d' : 0 \leq d' \leq d \implies u + d' \in I(s)\), and
- \((s, u) \xrightarrow{a} (s', u')\) if there exists \(e = (s, a, g, r, s') \in E\) s.t. \(u \in g, u' = [r \mapsto 0]u, \text{ and } u' \in I(s)\).

where \(d \in \mathbb{R}_{\geq 0} u + d\) maps every clock \(x \in C\) to value \(u(x) + d\), and \([r \mapsto 0]u\) denotes the valuation of the clock which maps every clock in \(r\) to 0 and agrees with \(u\) over \(C \setminus r\).

Figure 4.1 shows an example of timed automata which is designed in UPPAAL. There are two clock variables: \(x, y\), which are used to control the behavior of the timed automata. Clock \(x\) controls the self loop.

4.1.2 UPPAAL Timed Automata

UPPAAL as a modeling language which has extended the timed automata with several features as follows:
4.1 Timed Automata

Templates: To have a complete system, templates are defined for the processes which are instantiated. They include a set of parameters of any built-in types [14].

Constants: Constants: To have a integer value which can not be changed, they should be declared as: **const name value** [14].

Bounded integer variables: These are used to declare an integer value ranging from a minimum value to a maximum value as **int[min.max]**, where min and max are the bounds of the range. They can be used in guards, invariants, and assignments. The bounds will be checked in verification, and any violation of a bound will cause an invalid state. The default bounds range from -32767 to 32768 [14].

Binary synchronization: This is implemented using channels. The channels should be declared as **chan c**. When an edge is labeled with **c!** it synchronizes with the channel labeled **c?** [14].

Broadcast channels: These should be declared as **broadcast chan c**. In this type of synchronization, a sender **c!** can synchronize with more than one receiver **c?**. Any receiver which is able to synchronize in any state, must do so. When there is no receiver, the sender continues to execute the **c!** action [14].

Figure 4.1: An example of Timed Automata.[9]
4.1 Timed Automata

Urgent synchronization: This is declared using the urgent prefix in channel declaration. A delay is disallowed when a synchronization transition on an urgent channel is enabled. Nor is it possible to have any time constraints on an edge which uses an urgent channel [14].

Urgent locations: These are the locations in which time is not allowed to pass, i.e., a location which has clock x. The clock is reset when there is an incoming edge and the location has an invariant \( x_t = 0 \) [14].

Committed locations: These are like urgent locations, no delay is acceptable but more strict. In an urgent location, time will not be passed but inter-leavings with normal states are possible. A Committed location must be left immediately and the only possible transition is *Going out from the committed location* [14].

Arrays: These can be defined for clocks, integer variables, constants and channels. They should be defined by attaching a size to the name of the variable like \( \text{clock } x[8]; \text{int}[1,3] m[5]; \) [14].

Initializers: These are used for initializing the variables of the type integer and the arrays of integers, for example, \( \text{int } m := 0; \text{int } m[5] := 0,1,2,3,4 \) [14].

4.1.3 UPPAAL Expressions

UPPAAL expressions range over integer variables and clocks. Figure 4.2 shows the syntax of an expression in BNF. The following labels use expressions:

Guard: This is a special expression which satisfies the following conditions: No side-effects; evaluation to a boolean; clocks, integer variables, and the constants which are referenced (or arrays of them); clocks and clock differences, which are compared to the integer expressions; and guards over clocks [14].

Synchronization: This is label on the form \( Expression!, \) \( Expression? \), or an empty label. The expression must not have any side-effects, must be evaluated to a channel, and can only be referred to integers, constants and channels [14].

Assignment: The assignment label is a set of expressions which are separated by comma with a side-effect; the expressions can only refer to clocks,
integer variables, and constants. Furthermore, they can only assign integer values to clocks [14].

**Invariant** An expression which is side-effect free; refer to clock, integer variables, and constants. It is a conjunction over the conditions of $x \ll c$, where $x$ is a clock, $\ll \in \{<, \leq\}$, and $c$ evaluates to an integer. [14]

### 4.1.4 UPPAAL Query Language

For the sake of verifying a model, UPPAAL, like the other model checkers, is equipped with a query language which is a simplified version of CTL$^2$[27]. The UPPAAL query language includes formulae and state formulae but unlike CTL, it will not support nested path formulae. The state formulae can describe the states of the model, and path formulae are used to examine the paths or traces. There are three main types of path formulae: *reachability, safety and liveness* [14].

**Reachability properties:** These test whether there is any possibility of a given state formulae, $\varphi$, being satisfied by any reachable state, i.e., if there is any path which starts from the initial state in the way that $\varphi$ is satisfied along it. The properties are written in the form of $E <> \varphi$ in UPPAAL. [14]

---

$^2$Computational Tree Logic: The model in this logic is a tree-like structure.
4.1 Timed Automata

**Safety properties:** These test the condition that “something bad never happens.” For example, in a heating system, a safety property might be that the temperature must never reach some dangerous point. Another form of condition in these properties is to test can be ”something bad will possibly never happen.” For instance, a system which is supposed to search for a particular point in a plane, would possibly never miss that point. These properties in UPPAAL are formulated positively in the form of $A[[\varphi]$ which means that with the path formulae in all the reachable states the state formulae $\varphi$ should be true, and the path formulae $E[[\varphi]$ which means there exists a maximal path in which $\varphi$ is always true.[14]

**Liveness properties:** These test the condition that “something will eventually happen.” For example, by pressing a reset button in a given system, that system will eventually be reset. Liveness properties are written in the form of $A_{\leq}\varphi$ in UPPAAL which means the state formulae $\varphi$ is eventually satisfied. Another form of writing these properties is $\varphi \rightarrow_{>\psi}$, which can be read as “when $\varphi$ is satisfied, $\psi$ will be satisfied”. [14]

4.1.5 Time in UPPAAL

Upaal models a system as a finite state machine with clocks, and it uses the clocks to handle time. Uppaal uses a continuous time model, which is implemented as regions and therefore, the states are symbolic, i.e. there is no concrete value for the time and they are differences. The clocks check the progress of the time against the progress of the system as a whole, and it is possible to check or reset their values [9].

4.1.5.1 Time languages

The timed words are defined in a way such that a behavior of a real-time system corresponds to a timed word over the alphabet of events, and in the case of dense-time models, the time domain, $\mathbb{R}$, includes the set of non-negative real numbers [16].

**Definition 2** A clock interpretation for a set $C$ of clocks, assigns a real value to each clock, i.e. it maps $C$ to $\mathbb{R}$. A clock interpretation $i$ satisfies the clock constraint $\delta$ over $C$ if and only if $\delta$ is evaluated to true when using the given by $i$.

4.1.5.2 Clock Regions

A state and the values of the its clocks determine the future behavior of a timed transition table at each point of time.
### 4.1 Timed Automata

**Definition 3** Let \( \langle \Sigma, L, l_0, C, E \rangle \) be a timed transition table \( A \). For each \( x \in C \), let \( c_x \) be the largest integer \( c \) in the way that \( (x \geq c) \) or \( (x \leq c) \) is a sub-formula of some clock constraints. The equivalence relation \( \sim \) is defined over all Clock interpretation for \( C \); \( i_1 \sim i_2 \) if and only if all the following requirements are met:[16]

- For all \( x \in C \), \([i_1(x)]\) and \([i_2(x)]\) are the same, or \( i_1(x) \) and \( i_2(x) \) are greater than \( c_x \).
- For all \( x, y \in C \) with \( i_1(x) \leq c_x \) and \( i_1(y) \leq c_x \), \( \text{frac}(i_1(x)) \leq \text{frac}(i_2(x)) \), if and only if \( \text{frac}(i_1(y)) \leq \text{frac}(i_2(y)) \).
- For all \( x \in C \) with \( i_1(x) \leq c_x \), \( \text{frac}(i_1(x)) = 0 \), if and only if \( \text{frac}(i_2(x)) = 0 \).

A **clock region** for timed transition table \( A \) is defined as an equivalence class of a clock interpretations which is induced by \( \sim \). [16]

A **region graph** or **region automaton** is a finite state model which can be constructed using the equivalence classes induced by the un-timed bisimulation as symbolic states. In this model, the transition relationship between the states is defined using following formulas:[14]

- \( (l, [u]) \Rightarrow (l, [v]) \) if \( (l, u) \xrightarrow{d} (l, [v]) \) for a positive number \( d \).
- \( (l, [u]) \Rightarrow (l', [v]) \) if \( (l, u) \xrightarrow{a} (l', [v]) \) for an action \( a \).

There is more efficient way to represent the state-space of timed automato, which is based on the concept of **zone** and **zone-graphs**. In practice, thanks to usage of zones instead of regions in the graph, the representation of the state space is more compact.

Figure 4.4 shows a timed automaton and its zone graph as an example. There are 8 states in the zone graph, while the region graph for the same automaton has more than 50 states.
4.1 Timed Automata

4.1.6 Determinism in Timed Automata

A timed automata is deterministic when for each timed word \( w \), there is just one initial run. Time and Actions in a timed automata, are two parameters that can affect determinism. A timed automata is deterministic when for each timed word \( w \), there is just one initial run. Time and Actions in a timed automata, are two parameters that can affect on the determinism.

![Figure 4.4: Example of clock region.[14]](image)

![Figure 4.5: Non-deterministic TAs : a) w.r.t the time and b) w.r.t the actions](image)

Figure 4.5 shows the non-deterministic behavior of the time automata. Part A is a timed automaton in which there are two edges that are labeled by two time words. The timed words are not disjoint, and at the time \( x=c \), there are two possible next states. Part B shows a non-deterministic TA as well, but with respect to the actions. There are two distinct edges which are labeled by the actions.

37
Chapter 5

Automatic Test Generator Tool

This thesis work proposes an optimized off-line method for generating test cases for the UPPAAL timed automata model of the embedded software. The current chapter is a description of the method and the algorithm which is used in the proposed prototype tool.

5.1 Tool Specification

As mentioned above, the process of model testing should be started by preparing a model of the software to be tested. In this state, the model should be developed according to the requirements of the software and should be able to describe the entire system requirements (e.g. timing). Afterwards, test cases should be generated from the model and then run on the system using an adapter and a test executer.

The Artemis/MBAT project follows the same procedure. Several institutes and companies are contributing to this large project, and the contribution of Enea is to prepare an automatic model-based testing tool-chain, which will be integrated with the tools provided by the other partners. The tool-chain will include an automatic model-based test generator which receives a model from the partners’ tool(s), a test executer, which sends the test data to the system, receives the test results from the system and an adapter, which provides the connections between the system and the test executer. A test executer and an adapter for the Embedded software, which are running under Enea OSE RTOS\textsuperscript{1}, have been developed by Enea already called "Farkle." A test bench has also been developed and integrated into Farkle as an interface to receive the python test cases, run them on the

\textsuperscript{1}Enea OSE is an RTOS optimized for distributed, fault-tolerant systems.
5.1 Tool Specification

Automatic Test Generator Tool

target, and give the verdicts. The main idea is to develop an automatic test generator and integrate it with the test executer and the adapter.

Figure 5.1: The tool-chain work-flow

Figure 5.1 shows the tool-chain workflow. As the figure shows, in the first step, the model will be received by the test generator tool as an input. Since the project has just started, and the Meta model\(^2\) is not yet specified, we have agreed to start with the UPPAAL timed automata model as an input model, because UPPAAL can handle the real time restrictions, which is necessary for embedded systems in most cases.

On the other hand, according to the specifications of Farkle, the test generator tool should prepare test cases in python unit test formats. Moreover, another python module is required in which the required signals\(^3\)(input, output, and functions) are defined, and the module will be generated from a .sig file of the OSE application using a Ruby application called sigpa\(^4\).

---

\(^2\)A meta-model typically defines the language and processes from which to form a model

\(^3\)They are required to send the objects to the running application and receive the results from IUT

\(^4\)This application is a parser which is already implemented and is integrated to Farkle.
5.2 Farkle OSE Test Executer Framework

As mentioned in the previous section, Farkle is a framework for executing test cases in python format on a system which is running Enea OSE. It has been developed using Rubby and python. The Farkle Python classes includes OSEGateway and Linx communication types and classes to convert the signals in C language format into the Python classes.

![Farkle framework](image)

**Farkle** The figure shows the components which are forming Farkle and their relations. For using the signals in Python an off-line tool called Sigpa should be used on an Eneca OSE signal file for generating Python classes into a Python Class file. The Python class is extending the Python class signal.py in which the logic for converting the C language data structures into Python variables.

As it is described in previous chapter, UPPAAL timed Automata is a FSM graph essentially which is labeled by actions and time constraints. The constraints can lead the system to take different paths and producing the different outcome consequently.

5.3 Test generation preliminaries

In previous sections, the input and output of the test generator were described. The input which is an UPPAAL model should and it is representing the requirement of the system should be analyzed to obtain test cases. To this end, the first step would be obtaining the test traces from the model ac-
5.3 Test generation preliminaries

According to the test plan, and then test cases could be generated by following the extracted test traces.

### 5.3.1 Test trace

As described in the UPPAAL chapter, an UPPAAL model is an FSM graph which consists of a set of states connected via some labeled transitions. For generating test traces from such a model, based on the test plan, a test trace should be defined by following a specific path of the model.

![Figure 5.3: The paths of a simple graph](image)

Figure 5.3 shows a simple graph and all the possible paths which are starting from the initial state 1. According to the paths, for instance the following trace could be written for the first path as simply:

\[ 1 \rightarrow 3 \rightarrow 6 \]

The test traces will be generated by verifying the test models and a negated UPPAAL query. In this case, the UPPAAL verifier will generate a counter-example which can be used as a test trace. The test traces will be in the form of:

\[ l_0 \xrightarrow{c,u,v} l_1 \xrightarrow{c,u,v} \ldots l_n \]

where

- \( l_0..l_n \) are the locations.
- \( c \) is the clock constraints.
- \( u \) is the clock reset which may happen.
- \( v \) is the test data assignment(s).
- \( a \) is the required synchronization(action).
Challenges for finding the paths In the real world, complex systems contain not only few states and straights paths, they include inner loops inside. Furthermore, the communication between the timed automata could be another challenge.

- **Inner loops**: For the model in Figure ??, it is possible to use methods which are used for traversing the trees, e.g. pre-order traversing, but for the general cases, these methods are not applicable. For instance, the model in Figure ?? includes several loops, and the loops could be traversed infinite times.

The model which is presented in Figure 5.4 is one in which an infinite number of paths can be extracted by using the tree traversing methods. In the model, the loops will generate an infinite number of iterations and derive an infinite number of possible test traces. For instance:

\[ 1 \rightarrow 3 \rightarrow 6 \]
\[ 1 \rightarrow 3 \rightarrow 6 \rightarrow 1 \rightarrow 3 \rightarrow 6 \]
\[ 1 \rightarrow 3 \rightarrow 6 \rightarrow 1 \rightarrow 3 \rightarrow 6 \rightarrow 1 \rightarrow 3 \rightarrow 6 \ldots \]

There are some ways to solve this problem. One possible solution is to consider a maximum number of iterations for each loop, and it can work for such a model shown in Figure ??, but there would be other problems, like nested loops, which can be handled by this solution.

- **Timed automata communications**: As described in previous chapter, it is possible to have a network of timed automata. It is really difficult to traverse a specific path in a network of timed automata, since the path may include the states and transitions of the different timed automata.
5.3 Test generation preliminaries

**Solution:** In this thesis work, the general case is considered. Therefore, the UPPAAL model checker is used, since it is powerful enough and includes optimized solutions for generating the traces of a timed automata.

5.3.2 Deterministic input models

Another important concept which should be considered in model-based testing is to define whether the model is deterministic or non-deterministic already. In timed automata, as described in the previous chapter, actions (signals) and time may have either deterministic or non-deterministic behaviors. In non-deterministic cases, either with actions or time, the next behavior of the system is not predictable.

In off-line model based testing methods, a test case includes a finite structure of inputs and desired outputs. Test cases are generated and stored before running. They can be executed as many times as needed, and on each run the same inputs are provided and the same outputs are expected. It means that the SUT behaves the same when it receives the same sequence of events. Therefore, it is required that the behavior of the SUT is predictable.

This requirement makes it almost impossible (or at least much more sophisticated) to use off-line model-based testing methods to test the non-deterministic model. Of course, there are some approaches to changing the non-deterministic model to deterministic. The game approach [18], in which a game-based algorithm is proposed for the timed automata, tries to produce a language-equivalent deterministic timed automaton, otherwise a deterministic over-approximation.

Additionally, for non-deterministic cases, it is possible to use on-line model-based testing methods. In online testing, the test generation algorithms should react to the actual outputs of the SUT. The test generator can see which path in the model shall be taken. These methods are very efficient when testing non-deterministic models, but test cases cannot be stored and executed repeatedly.

The proposed model-based testing method in this thesis project is an off-line method, since it is required to provide Farkle with recorded Python test cases. As mentioned in earlier in this section, it is very complex to use test non-deterministic models using off-line methods, and thanks to time limitation of the project, it is assumed that the input model will have a deterministic behavior in time and action.

5.3.3 Testing the time requirements

One of the important aspects of a timed automata which is describes the behavior of a real-time system, is the timing requirement, and it is important to test the time deviations. In the on-line test generation methods, for
5.3 Test generation preliminaries

Automatic Test Generator Tool

testing a timed action, it is usual to have an auxiliary clock for observing and measuring the time which has elapsed by the IUT.

The auxiliary clock can work as an analogue clock, and it is easy to measure the dense value of time. In off-line methods, however, it is totally different, since it is dense, and it could be problematic, since for testing even a short range of time, it is required to specify an infinite number of test cases. The proposed method in this thesis work is off-line and the effort was to find a solution to overcome this problem. The solution is to use an auxiliary clock in the environment model and assign it to the input test data set when appropriate. In the following section, the algorithm of implementing this solution will be described.

5.3.4 Test environment model

It is common in on-line model-based testing methods to extract the test data from the SUT as the IUT is running and give the required data to the IUT when appropriate. However, in off-line methods, once a model is traversed, the test data will be extracted. Then, in each test case, the test data should be given to the IUT when appropriate.

In this work, the environment which contains all the test data and actions, will be modeled as a timed automaton and together with the system model will form a network of timed automata which is the test model. It may also contain an auxiliary clock, which will be used to construct the required guards in the environment of a timed automaton.

5.3.5 Integrating UPPAAL Verifier

This thesis work uses the UPPAAL model checker. The UPPAAL model checker is capable of providing a set of test traces. To this end, the following steps will be taken:

1. According to the test metric and test plan a negated UPPAAL query should be prepared.

2. Using the UPPAAL verifier as an executable command and the prepared negated query to generate the counterexamples and will be saved in a "xtr" file.

3. The counterexample is generated in a specific format that is readable for UPPAAL model checker to represent the traces in its GUI graphically. Therefore, the generated "xtr" file should be parsed into an understandable format for test generator tool.
5.4 Test generator tool

In this thesis work, I focused on conformance testing, i.e. the generated test cases will check whether the behavior of some black box implementation conforms to that of its specification. The first important thing is to define a test purpose. In a test purpose the goal of testing and the test environment should be defined. In this thesis work, the test purpose and test metrics can be defined using the UPPAAL query language. For instance, it is possible to check the reachability the internal states of the application which are represented as the locations in the Uppaal model, the constraints (e.g. guards, invariants, ...) which are defined on the transitions or locations, etc.

Figure 5.5: The schematic of the test generator tool

Figure ?? describes the test generator tool which is started by reading the system model to prepare the environment model and test model, and finally generating the test traces and test cases.

5.4.1 Preparing the environment model

For generating test cases, the first step is to generate the environment model. To this end, the model graph should be traversed, and all the required information for forming the environment model should be collected. Pragmatically, for performing this task, it is enough to parse the UPPAAL model of the system, since it is an XML file. In this XML file, all the required
5.4 Test generator tool

information, like locations, guards, assignments, clocks, etc., should be extracted using the parser and should be located in memory for preparing the test data.

Additionally, an auxiliary clock should be defined in the environment model to keep track of time and to specify the time for giving the test data to the IUT. For any transition which has a time constraint, the auxiliary clock should be defined based on the extracted time constraint value, and it should be assigned to the test data set of this specific transition.

For generating the test data set, the extracted information from the model should be processed as follows:

- **Updates:** Using the boundary value testing method, the test data for this assignment could be generated as follows:
  - A number which is less than the extracted value
  - The extracted value
  - A number which is greater than the extracted value

- **Actions:** There are formally two types of actions (synchronizations): Receive which is indicated by "!" and Send which is indicated by "?". If the extracted action is a Receive, then a Send should be added to the data set, and vice versa.

- **Guards and Invariants:** The same as the extracted value

**Auxiliary clock:** Since the method is off-line testing, the auxiliary clock can help to give the test data and the actions at the proper time. The auxiliary clock will be reset, as soon as any action in the environment is taken.

5.4.2 Preparing the test model

For constructing the test model, the environment model will be attached to the SUT timed automata and together, they will be verified using the UPPAAL verifier.

Figure ?? shows the main idea of the thesis in which the environment model and the system model form the test model as a network of timed automata. The environment which will be created automatically according to the test purpose, provides the SUT with the required test data when appropriate.
5.4 Test generator tool

The idea of generating the test model is not to construct a model formed by all the possible test traces. In the test model, the system model will be provided by all the possible test data. Pragmatically, a system usually requires some test data from the real environment, such as users, sensors, actuator, etc. Here, the environment timed automata plays this role. Via the UPPAAL communication channels, the environment can give data to the system timed automata, and when UPPAAL is asked by an UPPAAL query to check, for instance, the reachability of a location or a transition, it will start from the initial location of the system automata to traverse the queried path and through this traverse, it can obtain all the required input data from the environment timed automata.

5.4.3 Implementation

In this thesis, a prototype of the test generator is implemented. The implementation consist of a simple GUI, a full node coverage negated query generator, an environment model generator, a query executor, and a parser to parse the result of the query execution.

5.4.3.1 GUI

A simple GUI to facilitate the process of giving inputs and receiving the test traces. It needs the UPPAAL model and a negated query as inputs. As it is shown in figure ??, there are three possibilities to specify the negated query:

- It is possible to browse a file which contains the query.
- It is possible to write the query in the specified text field directly.
- It is possible to ask the test generator tool to generate the required negated queries to generate full node coverage test(s).
5.4 Test generator tool

5.4.3.2 Full node coverage negated query generator

In this work, the negated query is important. A test case is generated based on the specified negated query. The negated query can lead to generate different types of test cases e.g. node coverage, edge coverage, etc. Algorithm 1 shows the algorithm of generating the required negated queries for generating full node coverage test(s).

Algorithm 1 The algorithm for generating the negated queries for full node coverage test.

```plaintext
1: procedure GENNEGQUEFULLNODECOVERAGE
2:    uppaalModelFile ← UPPAAL model as an XML file
3:    processInModel ← Extract processes from uppaalModelFile
4:    i ← 0
5:    negQueries ← []
6:    for process in processInModel do
7:        for node in process.nodes do
8:            negQueries[i] ← \{verifier uppaalModelFile "A[] not $node"\}
9:        i ← i + 1
```

As a proof of concept, the test generator tool is generating negated
queries for a full node coverage test automatically, but it is simply possible to extend the implementation to generate the other types of negated queries. To this end, it iterates over all nodes and it generates a query for each node in the form of:

$$A[.] \text{ not } \{Node\}$$

In fact, the negated queries which are generated by test generator tool automatically for full node coverage tests claim that it is not possible to reach the specified node. When UPPAAL verifier receives the query, it searches for a counterexample. It returns the first path to reach the specified node.

### 5.4.3.3 Environment model generator

This module defines and creates the required environment model. As it was described in this chapter, the environment model is attached to the test model to provide the required test data. For the sake of simplicity, the current test generator prototype assumes that the test model can include only one clock, but in reality an UPPAAL model can have more than one clock.

**Algorithm 2** The algorithm of generating the environment model.

1: procedure GENENVMODEL  
2: uppaalModelFile $\leftarrow$ UPPAAL model as an XML file  
3: processInModel $\leftarrow$ Extract processes from uppaalModelFile  
4: envModel $\leftarrow$ []  
5: for process in processInModel do  
6: for item in process.items do  
7: if item == clock then  
8: envModel.extend("clock")  
9: else if item == sync.answer then  
10: envModel.extend("Sync.answer")  
11: else if item == sync.request then  
12: envModel.extend("Sync.answer")  
13: else if item == condition then  
14: envModel.extend(Assignments for boundary values)

As Algorithm 2 shows, the test generator tool walks through the test model, collects the clock, guard(s) and synchronization(s). Then, based on the collected information it adds a transition to the environment model with or without guard(s), synchronization(s) or assignment(s). For instance, if there is a synchronization receive in the test model, it creates a transition with a synchronization send in the environment model.
5.4.3.4 Query executor

This component is responsible for executing negated queries using UPPAAL verifier. As it was described in earlier sections, the process of executing negated queries is more than executing UPPAAL verifier as an external executable file. It needs to be done in several steps. Algorithm 3 shows the process of executing the negated query.

**Algorithm 3** The algorithm of executing negated queries.

```
procedure execNegQuery
  uppaalModelFile ← UPPAAL model as an XML file
  // The .xtr files contain the counterexample(s).
  xtrFile ← GetCountexampleFromUppaalVerifier(uppaalModelFile)
  // The .if files contain the model information which is required
  // to parse .xtr file.
  ifFile ← GetModelInfoFromUppaalVerifier(uppaalModelFile)
  stringTestTraces ← ParseXtrToStringTestTraces(ifFile, xtrFile)
  pythonTestCases ← GenPythonTestCases(stringTestTraces)
```

UPPAAL verifier describes the counterexample as a group of IDs in an .xtr file. These IDs which are generated by UPPAAL for a specific model, represent different objects of that model e.g. Location, Transition, Edge, etc. The .if which can be generated by UPPAAL verifier for a specific model includes the list of IDs of that specific model. Therefore, in order to parse the counterexample and create the Python test case, it is required to generate both .xtr and .if files.

5.5 Use Case

In this thesis, one of the simple UPPAAL models, called “simple lamp”, is used as a use case, and its OSE application will play the role of IUT.

Figure ?? shows the “simple lamp” model, and Figure ?? shows the lamp model combined with the generated environment model. It is a model of a smart switch for a lamp. If user presses the button once, it will cause the lamp to turn at a low level. When the switch receives a second press, and if the second press occurs less than 5 seconds after the first press, the lamp will become brighter, otherwise, the lamp will be turned off. A test purpose was designed for this case to check the possibility of reaching the state bright.

The environment in Figure 5.8 is generated for the following negated Up- paal query: A // not lamp.bright.
5.5 Use Case

Automatic Test Generator Tool

Figure 5.8: Simple lamp

Figure 5.9: Simple lamp and the relevant environment
The outcome of using Uppaal Verifyta for this case, is to receive a counter example as follows:

\[
\text{off } \xrightarrow{\text{press!}, \text{reset}(y), \text{toLow?!}} \text{low} \xrightarrow{\text{press!}, y<5, \text{toBright?!}} \text{bright}
\]

The auxiliary clock $x$ which is defined in environment model plays an important role in this example. There is a synchronization $\text{press}$ which based on the time can lead to two different outcomes.

When defining the environment model, the test generator tool searches for the clock in the model of the smart lamp and finds $y$. Then, it searches through the model and check if there is any guard which includes this clock. In this example, it finds two expressions $y < 5$ and $y \geq 5$ which are combined with $\text{press}$ synchronization. Therefore, the environment contains two transition with the synchronization but with different time conditions $x < 5$ and $x \geq 5$ to cover both possible transitions.
Chapter 6

Conclusion and Future Works

6.1 Future work

The developed tool prototype can accept UPPAAL model and the UPPAAL parser and verifier is developed and a good idea is to implement the parsers and verifiers. Another area for future work is to implement an on-the-fly approach, especially for the sake of the Enea/Optima debugger which can help to find failures online.

6.2 Conclusion

In this thesis, the main focus has been to study the methods for automating the process of model-based testing for embedded systems. Therefore, the thesis started by studying related works and collecting pertinent ideas. After studying model-based testing methods and related works, it seemed apparent that the first implementation could be started by implementing a prototype to accept a UPPAAL timed automata as the input model and by doing the test generation process automatically, since the IUT in this thesis work is an embedded application, which was developed to work under a real-time operating system, and UPPAAL timed automata are powerful enough to handle all requirements for such an application. Moreover, the UPPAAL verifier used in the prototype is a powerful verification tool which can be useful to find traces.

6.2.1 Answer for the questions in the problem statement

After studying and investigating the solutions for the problem statements, the following answers were found for the first question:
Q1. What kind of tools or applications are required to analyze the given model?

Uppaal as a powerful model checker provided the possibility to analyze the given model. Uppaal is capable of giving counterexamples as a negative reply to a given Uppaal query. The counterexample could be used as a test trace or test sequence. It determines a trace which contains all the transitions to be traversed and the required test data and actions.

Q2. Is it possible to have different kinds of test coverage in the test generator?

Yes, the prototype tool is capable of generating different test cases with respect to the test coverage type. Since, the environment and test sequence generation is based on the given Uppaal query, it is possible for a user to define different test cases with respect to the test coverage, and even the test case can be generated to test a very specific constraint.

Q3. How to test a continuous time range?

As discussed in Chapter 4, the time ranges in Uppaal can be divided into time zones. Therefore, for testing a continuous range of time, it is possible to have a finite number of test cases for a finite number of time zones instead of having an infinite test case for each specific time.
Bibliography


[5] Bernhard Aichernig, Willibald Krenn, Henrik Eriksson, and Jonny Vinter. D 1.2 - state of the art survey - part a: D 1.2 - State of the Art Survey - Part a: Model-based Test Case Generation. Technical report, 06/2008 2008. For more information, please contact one of the authors: Bernhard Aichernig: aichernig@ist.tugraz.at Willibald Krenn: wkrenn@ist.tugraz.at Jonny Vinter: jonny.vinter@sp.se Henrik Eriksson: henrik.eriksson@sp.se.


