Automatic Test Generation and Mutation Analysis using UPPAAL SMC

Mälardalens Hägskola

Akademin för Innovation, Design och Teknik

Jonatan Larsson

JLN13010@student.mdh.se

Bachelor of Computer Science

2017-06-04

Examiner: Associate Professor Cristina Seceleanu

Supervisors: Dr. Raluca Marinescu and Dr. Eduard Enoiu
Abstract

Software testing is an important process for ensuring the quality of the software. As the complexity of the software increases, traditional means of manual testing becomes increasingly more complex and time consuming. In most embedded systems, designing software with as few errors as possible is often critical. Resource usage is also of concern for proper behavior because of the very nature of embedded systems. To design reliable and energy-efficient systems, methods are needed to detect hot points of consumption and correct them prior to deployment. To reduce testing effort, Model-based testing can be used which is one testing method that allows for automatic testing of model based systems. Model-based testing has not been investigated extensively for revealing resource usage anomalies in embedded systems.

UPPAAL SMC is a statistical model checking tool which can be used to model the system’s resource usage. Currently UPPAAL SMC lacks the support for performing automatic test generation and test selection. In this thesis we provide this support with a framework for automatic test generation and test selection using mutation analysis, a method for minimizing the generated test suite while maximizing the fault coverage and a tool implementing the framework on top of the UPPAAL SMC tool. The thesis also evaluates the framework on a Brake by Wire industrial system.

Our results show that we could for a Brake-by-wire system, simulated on a consumer processor with five mutants, in best case find a test case that achieved 100% mutation score within one minute and confidently identify at least one test case that achieved full mutation score within five minutes. The evaluation shows that this framework is applicable and relatively efficient on an industrial system for reducing continues resource usage target testing effort.
## Contents

1. Introduction .................................................................................................................. 4

2. Background ...................................................................................................................... 4
   2.1 Model-based Testing ................................................................................................. 5
   2.2 Timed Automata ..................................................................................................... 6
   2.3 UPPAAL SMC ....................................................................................................... 7
   2.4 Mutation Analysis ................................................................................................. 8

3. Related Work .................................................................................................................. 9

4. Problem Formulation .................................................................................................... 9

5. Research Method ......................................................................................................... 10

6. A Framework for Automatic Test Generation using UPPAAL SMC ..................... 11
   6.1 Framework Overview ............................................................................................ 11
   6.2 Automatic Test-Case Generation .......................................................................... 12
   6.3 Mutation Analysis ............................................................................................... 14
   6.4 Test Suite Selection and Minimization ............................................................... 14
   6.5 Tool Support ......................................................................................................... 15

7. Case Study: The BBW Prototype System .................................................................. 18
   7.1 The BBW Prototype ............................................................................................. 18
   7.2 Case Study Results ............................................................................................... 20

8. Discussion ..................................................................................................................... 22

9. Conclusions ................................................................................................................... 22

References ....................................................................................................................... 24
1. Introduction

Software testing is a process of ensuring the quality of the software by executing and observing the actual behavior of software [1, 5]. By uncovering software bugs, testing can be used to improve reliability, usability, efficiency, and maintainability of software. Nowadays, software testing is gaining importance in the development of embedded systems as the complexity and criticality of this kind of software increases dramatically. To ensure the development of reliable systems with less errors, the creation of effective testing methods and tools is required. With manual testing being a labor-intensive process, there is a growing need for efficiently created tests as software complexity is increasing dramatically.

To improve test efficiency, model checkers (i.e., tools used to formally verify finite-state systems) can be used in an attempt to automate the creation of tests. State of the art model-checkers like UPPAAL SMC [4] can be used to model the system’s resource usage, and thus enabling the generation of test cases that target such extra-functional requirements. Currently UPPAAL SMC has no support for performing automated test generation. A couple of recent studies [6, 21] have proposed methods for automatic test generation and a framework to perform automatic mutation analysis in UPPAAL. While these approaches are promising, there is a need to practically implement these methods in a test generation and mutation analysis integrated framework. In this thesis we intend to fill this gap by proposing an integrated framework and implement it in a software tool to be used by engineers for automated test generation. We show how to generate test cases in UPPAAL SMC by using its ability to simulate systems and evaluate the generated tests by measuring each test ability to detect model faults. Based on this method the proposed framework can be used to reduce the test suite by selecting only the tests that are maximizing the mutation coverage (i.e., detecting the injected model faults). To evaluate this solution we apply it on an industrial system: the Brake By Wire industrial prototype system.

The thesis makes the following contributions:

- A method for minimizing the generated test suite while maximizing the fault coverage.
- A tool implementing the framework on top of the UPPAAL SMC tool.
- An evaluation of the framework on the Brake By Wire industrial system

The results from the evaluation show that tests can be generated to achieve full mutation coverage on the system could be reached in 277 seconds. These encouraging results show that the framework is applicable on an industrial system.

In Section 2 we describe the concepts of model-based testing, Timed Automata, UPPAAL SMC and mutation analysis. Section 3 presents the related work to this thesis. In Section 4 we formulate our problem formulation and in Section 5 we present our research method. In addition, in Section 6 we present the proposed framework and its tool implementation. Section 7 contains the result of our evaluation case study. In Section 8 we discuss the results and the details of the tool implementation. Section 9 concludes this thesis and we suggest possible future work.

2. Background

In this section, we describe some of the concepts and tools that are relevant to the contribution of this thesis, which include (i) model-based testing (see Section 2.1), (ii) timed automata and UPPAAL SMC (see Section 2.2 and 2.3) and (iii) mutation analysis (see Section 2.4).
2.1 Model-based Testing

Model-based testing (MBT) is a testing method that depends on explicit behavior models that represent the intended behavior of a System Under Test (SUT) [15]. Test cases are generated from models and then executed on the SUT to determine differences between the implemented system and the intended behavior. The generic process of MBT is divided in several steps mirrored in Figure 1.

![Diagram of MBT process]

**Figure 1.** A generic model-based testing process.

The **Model** is an abstract representation of the SUT and it is built from the **Requirements** of the intended behavior and represents the part of the system that needs to be tested. An abstract model cannot be used to directly create tests that can be executed on the SUT because the **Model** is on a different level of abstraction containing less details than the SUT. However, the **Model** should precisely represent the SUT and represent the real system behavior. The **Model** can have different characteristics and it is possible to create timed, untimed, deterministic and non-deterministic behaviors. The characteristics of the **Model** are important for the needed abstraction and it impacts the choice of model paradigm. Some common paradigms to describe the model are transition-based, functional, stochastic and data-flow.

**Test Selection Criteria** are also derived from the **Requirements** and are used to produce relevant and comprehensive test suites. The criteria define what type of test model will be used to produce an effective test suite. Such criteria include: structural model coverage, data coverage, and random & stochastic criteria (e.g., environment models where the environment determines the usage patterns). In addition, it defines in which way the tests will be generated by using either random generation, model-checking or constraint solving.

**Abstract Test Cases** are test cases generated to satisfy the test case specification and represent the actual tests generated based on the **Test Selection Criteria** from the **Model**. Sometimes, a large number of test cases will be generated to satisfy a given **Test Selection Criteria**. Thus, there is a need to minimize the set of test cases (e.g., redundant test cases, minimizing test execution). The test generator can choose to minimize the amount of test cases that fulfill the test case specification to speed up testing on the SUT later.

The **Adapter** is used to convert the **Abstract Test Cases** into **Executable Test Cases** that will be executed against the SUT. That means that the **Adapter** needs to fill in the missing information in the abstracted test cases created from the model. The **Adapter** is usually an important part of the
process and requires that the generated tests are to be executed automatically. The **Executable Test Cases** are the tests that are composed to be ran on the SUT, which is the real system. The result of running the **Executable Test Cases** will result in a pass or fail **Verdict**.

### 2.2 Timed Automata

Timed Automata (TA), first proposed by Alur and Dill [18], are a finite-state automata extended with timing constraints. TA introduces a finite set of real valued clocks, where a clock is a continuous variable that measures the elapsed time and progresses continuously. This mechanism allows for modeling of time and timing properties, which allows one to model the timing behavior of embedded and real-time systems. Priced timed automata (PTA) is an extension of timed automata with cost variables that are allowed to progress at rates different from 1, which is useful for encoding resource consumption of a continuous nature, such as energy consumption.

An automaton is modeled as a set of locations and a set of edges that connect different locations. Locations can be annotated with an invariant, which is a conjunction of clock constraints that defines when the TA must leave the location. Edges can be annotated with a guard, an assignment and an action. A guard is a Boolean expression that defines the constraints on discrete or clock variables. When the guard is satisfied the edge can be taken. The assignment allows one to manipulate the variables (e.g., resetting the clock). Action refers to the synchronization with other timed automata in the network through a communication channel (handshake synchronization - one to one, or broadcast synchronization - one to many). Assuming a channel named "chan", this mechanism allows the synchronization of the sender "chan!" with the receiver "chan?".

The semantics of a TA are defined as labeled transition systems. The two types of transitions are **Delay transitions** and **Action transitions**, where a **Delay transition** happens when none of the invariants are violated in the current location and clocks can progress without changing location, and an **Action transition** happens when an explicit edge is taken.

In Figure 2 and 3 we show a network of PTA of a small coffee machine example. In this model, a user asks for coffee every 2 to 100 seconds, and the machine is making coffee for a duration between 4 and 5 seconds. When the coffee machine is idle, it consumes little energy, but this changes when coffee is produced. To achieve this, we define a continuous variable called “energy” for which we explicitly model the rate of consumption, noted with "energy' == value" on each location of the coffee machine automaton.

![Figure 2. User automaton.](image)
Each PTA starts at their initial positions, the machine starts increasing the energy with 0.1 for every time unit/second spent in this location while waiting for the user to order coffee. The User waits a duration between 2 and 100 before jumping to Ask_for_Coffee location and while taking the edge sends a signal to the coffee machine to start making coffee. When the machine gets Make_Coffee_start signal it follows the edge and resets clock \( x \). When the coffee machine jumps the next edge, it signals the user that coffee is done and the user resets its clock to prepare for the next loop.

### 2.3 UPPAAL SMC

UPPAAL [3] is a toolbox used to model and analyze real-time systems modeled as timed automata, developed jointly by Uppsala and Aalborg University. UPPAAL SMC (Statistical Model Checker) [4] is an extension of UPPAAL that allows a user to simulate the system over many runs and has the ability to give approximate answers using statistical analysis and an estimate of the correctness of the property to be checked with a given confidence level. By employing such techniques, UPPAAL SMC avoids the exhaustive state-space search of usual symbolic model checking (e.g., UPPAAL), hence providing benefits in terms of improved scalability. The SMC extension also supports priced timed automata models which improves the ability of modeling continuous resources such as energy. UPPAAL can compose Timed automata in parallel to form a network of TA to model complex systems.

In UPPAAL SMC queries are used to control the evaluation process and the query language UPPAAL SMC uses is Weighted Metric Temporal Logic (WMTL) [16]. The following properties can be verified with UPPAAL SMC:

- Probability Estimation: What is the probability of an event (i.e., a conjunction of predicates over the states of the TA network) to occur?
- Hypothesis Testing: Is the probability of an event greater or equal to a certain threshold?
- Probability Comparison: Is the probability of one event greater than the probability of another event?

UPPAAL SMC also supports the ability to visualize a simulation of a system (or a set of overlapping simulations).

In the Coffee machine example from Figure 2 and 3 we use the query that generates random simulations,

\[
\text{Simulate } n \{<=\text{bound}\} \{E_1, E_2, \ldots E_n\}
\]  

where \( n \) is number of simulations, \( \text{bound} \) is how many time units each simulation will run for and \( E_1, E_2, \ldots E_n \) are expressions that are monitored and visualized.

In Figure 4, we show how to use this simulation query in practice by using the following property:
This is used to simulate three system instances for 300 time units and monitor the value of energy. The result of the simulation can be displayed as a graph, where the x-axis is time and y-axis is the values of tracked variables (energy) defined in the query.

![Simulation result for the coffee machine example in UPPAAL SMC.](image)

**Figure 4.** Simulation result for the coffee machine example in UPPAAL SMC.

### 2.4 Mutation Analysis

Mutation testing (also known as mutation analysis and illustrated in Figure 5) is a method used to analyze tests by evaluating the ability of these tests to detect small changes in the software (e.g., to evaluate the effectiveness of a test suite) [2]. These small changes in the software (also called mutants) can be manually or automatically created using a mutation tool based on a set of mutation operators or manually injected faults [2].

![A generic mutation analysis process.](image)

**Figure 5.** A generic mutation analysis process.
The created test cases are then executed on these mutated programs, to check if the error (or mutation) is detected. The effectiveness rate of a test suite is measured as a mutation score which is the ratio of killed (detected) mutants out of the total number of mutants. The purpose of mutation testing is to produce test suites that are sensitive to small changes in the software in order to ensure that tests are properly testing the behavior of the program while maximizing the mutation score.

3. Related Work

There has been extensive work and research on automatic test generation that has produced tools such as EvoSuite [9], Pex [10], Randoop [11]. These tools try to provide comprehensive unit testing for different programming languages using different techniques to achieve high code coverage. While these tools and techniques are focusing on the code level, UPPAAL TRON [12] and UPPAAL CoVer [13] are tools intended to be used on a higher abstraction level than code for automated testing of systems modeled in timed automata. UPPAAL TRON is not only an automated test generation tool, but allows a user to run tests on a running software system in real-time. UPPAAL CoVer [14] is able to generate test cases using a novel global reachability algorithm and stored to be used for offline test execution.

Even if there is a relatively large body of knowledge on the use of model checking (e.g., UPPAAL) for testing functional and extra-functional requirements, few approaches and tools have used statistical model checking for automatically generating tests and selecting tests using mutation analysis. One such approach is implemented in MaTeLo [7], a commercial tool that combines statistical usage testing based on a MCUM (Markov Chain Usage Model) and specification based testing using an FDT (Formal Description Technique). UPPAAL SMC is lacking such testing capabilities (e.g., test generation, mutation analysis) that could be useful for both research and industrial practice in the development of reliable embedded and real-time systems.

There has been some research [21] on test generation and mutation analysis using UPPAAL model checker. Compared to this work, we are focusing on UPPAAL SMC and its ability to export simulation runs which makes our work novel. A recent study [6] on test case generation for energy consumption in embedded systems using UPPAAL SMC has already been conducted with promising results. The study proposes an approach to generate test suites for testing the energy consumption using an EAST-ADL [19] system architectural model. This result motivated us to propose a framework for test case generation and mutation analysis using UPPAAL SMC. In comparison to previous related work [19], our thesis proposes a framework for automatic test generation and selection of the minimum set of tests maximizing mutation score.

4. Problem Formulation

The growing complexity of embedded systems requires new techniques and tools that are able to support testing of extra-functional properties like resource usage (e.g., energy, bandwidth, memory). Currently there exists a research gap for automated test generation and selection for this kind of resource models using statistical model checking (as explained in our related work section).

UPPAAL SMC is a tool that can handle statistical verification of continuous resource properties and simulation of models, but lacks the ability to generate tests and perform fault detection analysis. A recent study [6] has shown that UPPAAL SMC can be used for test generation and manual fault detection analysis. This method lacks a way of automatically extract test suites that can be used for test execution and evaluation. One way to evaluate test cases is to perform mutation analysis. Mutation analysis has been shown experimentally to be a good proxy measure
for real fault detection [22].

Our research goal is to investigate how to automatically generate and select tests (based on mutation analysis) by using UPPAAL SMC, and evaluate the applicability of our proposed solution on an industrial case study. To tackle this, we propose the following research questions, which will be answered in this thesis:

1. How to generate test cases for testing resource consumption in embedded systems by using UPPAAL SMC?
2. Using mutation analysis as the underlying technique, is it possible to select the test cases that are maximizing the fault detection in terms of mutation score?
3. Are the proposed solutions applicable on an industrial system?

5. Research Method

A research method is a structured and systematic process that is applied to answer certain research questions (or confirm/refute a research hypothesis) [8]. The thesis starts with the formulation of the research questions that are investigated in this thesis, based on a short investigation of existing related literature. To address the research questions listed in Section 4, we start with finding the existing limitations of using statistical model checking (i.e., UPPAAL SMC) for test generation and mutation analysis. The thesis ends with performing an evaluation of this solution. These research questions are answered by the following steps (illustrated in Figure 6):

1. A literature search and review used to study the state-of-the-art in using statistical model checking for resource-aware automated test generation (i.e., data handling, tool architecture and framework capabilities).
2. Refinement of the initial thesis goals and research questions based on the previous step.
3. Proposal of a solution by determining the appropriate method to extract and generate tests as well as how mutation analysis can be used using UPPAAL SMC.
4. Development of the solution and implementation of the tool support on top of UPPAAL SMC.
5. Validation of the solution on a relevant industrial system. The method will be evaluated on a Brake-by-Wire (BBW) system. The results will be analyzed and discussed in terms of method applicability. If the results are not satisfactory we will go back to step 4 and the solution is redeveloped.

Figure 6. The research method process.
6. A Framework for Automatic Test Generation using UPPAAL SMC

In this section, we provide a detailed description of the testing framework proposed in the thesis and its tool support.

6.1 Framework Overview

To answer the research questions in Section 4, a software framework has been developed based on UPPAAL SMC for automatic test case generation. An overview of the framework is presented in Figure 7, where the numbered steps are as follows:

- **Step 1**: The engineer creates a TA model, an abstraction of the SUT that is evaluated.
- **Step 2**: The engineer also formulates the query to guide the test case generation.
- **Step 3**: With the provided model and query, UPPAAL SMC randomly generates simulation traces, which represents our test cases and contain the set of inputs and the expected output.
- **Step 4**: From this the simulations test cases (i.e., inputs and output values) are extracted and used later in Step 8.
- **Step 5**: The generated input values are extracted and used for mutation analysis.
- **Step 6**: The sequence of inputs in each test case are automatically inserted in a set of mutated models that the engineer has created or generated.
- **Step 7**: The mutated models are simulated with the extracted inputs using UPPAAL SMC to generate new sets of actual outputs from the mutated models in order to measure the difference between the original and the mutated model.
- **Step 8**: The actual outputs extracted from the mutated models for each test case will be compared to the expected outputs in order to determine the test case ability to kill (detect) any difference between the mutated models and the original (assumed correct) model.
- **Step 9**: From the analysis result in Step 8, test cases are selected based on the mutation score achieved. This steps removes the tests that are not contributing to a larger mutation score for the entire test suite.

![Figure 7. The proposed framework.](image)
6.2 Automatic Test-Case Generation

While UPPAAL SMC is a tool for statistical model checking, it is not directly tailored to test case generation. We demonstrate how to work around this by automatically generating simulation runs for a model described in timed automata extended with energy consumption and how we transform these simulations to actual test cases. A simulation run produced by the model checker for a given random run defines the set of inputs in time executed on the model which in our case is considered the system model. Test cases are obtained by extracting from the test path the observable input values at any given time.

The automatic test generation step in the framework is implemented using the UPPAAL SMC command-line called Verifyta. This step is used to automatically generate simulation traces. To be able to use the simulation data, it needs to be automatically analyzed. An example output from Verifyta of this extracted data is given in Figure 8. The following information is extracted: the parameter name (line 4), simulation number (line 5) and the points pairs (the rest of line 5) representing the simulation trace as \((X,Y)\) points in a vector graph.

![Figure 8. An output simulation trace example from Verifyta.](image)

To parse the simulation data, we use the algorithm illustrated in Figure 9. The algorithm will retrieve the parameter’s name (e.g. `pBrakePedelLDM_ELSignalIn` in Figure 8) and all the points from each simulation and store it internally. As the points from Verifyta represent a vector graph, we need to sample them at predefined data points to retrieve all the needed information for test execution.
For each simulation, a test case is generated which contains the values of all the user-defined input, used to trigger the SUT and an expected output (e.g., energy consumption) generated from the formal model. These are retrieved with the parser and stored for use later.

To extract the test case inputs from the simulation data the sampling algorithm shown in Figure 10 is used. The sampler is based on the line equation:

$$y = mx + b$$

where $y$ is the value at the sample point, $m$ is the gradient, $x$ is the relative sample point and $b$ is the y intercept.

```
public List<PointF> SampleTraces(decimal stepSize, decimal simTime, decimal offset)
{
        Create List of list of PointF 'sampledTraces'
        ADD list of PointF to 'sampledTraces'
        SET stepCount to offset
        FOR all points 'j' in simulation 'i' except first point
                GET point1[i][j-1]
                GET point2[i][j]
        WHILE stepCount is less then point2.X and stepCount is less or equal simTime
                SET diff_X = endPoint.X - startPoint.X
                SET diff_Y = endPoint.Y - startPoint.Y
                SET gradient = diff_Y / diff_X
                SET Y = gradient * (pointSize - StartPoint.X) + StartPoint.Y
                ADD new point(PointSize, Y) to 'sampledTraces[i]'
                StepCount += stepSize
        return sampledTrace
}
```

Figure 9. Simulation Data Parsing Algorithm.

Figure 10. Sampling algorithm.

After the simulation data has been sampled at each data point, the test case inputs are extracted and stored as an array ready to be used for mutation analysis.
6.3 Mutation Analysis

To evaluate the generated test cases, we integrate mutation analysis in our framework. The set of selected mutants, created by adding small errors (mutations) in the original model (e.g., changes in rate of energy consumption for different components). These mutants are simulated using the same inputs generated by the reference model in each test case. The inputs of each test case are given as a predefined set of inputs for the mutated model. For example, for 5 mutants and 10 test cases a total of 50 models (e.g., mutants X test cases) must be created. At this point the mutated models with the predefined inputs are ready to be simulated in Verifyta. The result of these simulations represents the actual output which is stored and compared with the expected output generated initially from the original model.

To evaluate the mutation result each output is sampled using a user defined granularity and then compared at each time point with the expected output to detect any significant differences larger than a user defined $\Delta$-value (as illustrated in Figure 11 by the algorithm for signal difference detection). This framework relies on the expertise of the engineers responsible for testing such systems. An experienced engineer should define what is an acceptable deviation. We note here that small deviations between the outputs are to be expected and the $\Delta$-value can vary from one system to another.

```
1 public int CompareOutput(List<PointF> first, List<PointF> second, float delta) {
2     if first.Count is not equal second.Count
3         return -1
4     for all points 'i' in 'first'
5         if |first[i].Y - second[i].Y| >= delta
6             return 1
7         return 0
8 }
```

**Figure 11.** An algorithm for measuring output signal difference detection.

The result of this mutation analysis is collected in a tabular form and can be used by the test suite selection and minimization module. The table is a $m \times n$ matrix, where $m$ is the number of mutations and $n$ is the number of test cases. Each entry is marked with either 0 or 1, where 1 represents a killed mutant and 0 represents an undetected mutant (as shown in Table 1). The mutation score can be calculated based on this matrix. For example, Test case 1 and 2 have a mutation score of 66% since these tests kill two mutants. However, since this value is not directly needed for test suite selection and minimization, we do not explicitly calculate it in the tool.

<table>
<thead>
<tr>
<th></th>
<th>Mutant 1</th>
<th>Mutant 2</th>
<th>Mutant 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test case 1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Test case 2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 1.** Example of a mutation analysis result.

6.4 Test Suite Selection and Minimization

Once the mutation analysis result is available, the test suite selection module can process these results and prepare the final test suite. The goal of this process is to highlight which test case or combination of test cases is enough to maximize the mutation score while removing the test cases
that are not improving this score. The algorithm used in test selection is presented in Figure 12, by using mutation analysis result to determine the maximum coverage provided by the set of test cases. The algorithm starts by adding the first test case in the list of selected test cases (Line 4 in Figure 12), then it compares all test cases with the ones in the list to see if it is a better test case or one that kills different mutants (Lines 5-17 in Figure 12). If there is a better test case in the test suite (i.e., a test case that improves the mutation coverage) then it replaces the existing one from the list. If there is a test case that kills other mutants then this test case is saved in the list. This step is repeated until every test case has been evaluated in this way. After the evaluation of all test cases, the duplicated tests in the list are removed (Lines 19-27 in Figure 12) and the list of selected test cases is shown.

```csharp
1. Public List<int> SelectBestTestCase(List<List<int>> resultTable)
2. {
3.     Create List of int 'testCase'
4.     ADD 0 to 'testCase'
5.     FOR each entry 'i' in 'resultTable'
6.         SET bool 'add' to true
7.         FOR each entry 'j' in 'testCase'
8.             SET first = 'resultTable[testCase[j]]'
9.             SET second = 'resultTable[i]'
10.            Compare 'first' with 'second'
11.            IF second contains same and more
12.                replace 'testCase[j]' with 'resultTable[i]'
13.                SET 'add' to false
14.            ELSE if second covers same or less
15.                SET 'add' to false
16.            IF 'add' is true
17.                ADD 'resultTable[i]' to 'testCase'
18.         FOR each entry 'j' in 'testCase'
19.             SET 'same' to false
20.             FOR each entry 'k' in 'testCase' starting at j + 1
21.                 IF 'testCase[j]' is equal to 'testCase[k]'
22.                     SET 'same' to true
23.                     break
24.             IF 'same' is true
25.                 remove 'testCase[j]'
26.                 decrease 'j' with 1
27.             return 'testCase'
```

Figure 12. Test selection algorithm based on mutation coverage.

### 6.5 Tool Support

To support and automate the framework described in the previous section, we developed the MATS tool. The current implementation is developed in C# and is intended to be used on computers using Microsoft Windows OS using .NET 4.5.2. Some of the tool technicalities are presented in this section.

To get the outputs from UPPAAL SMC's command-line, Veriflyta is practically executed with the arguments "-q -s 'filepath_to_model' 'filepath_to_query'". The first two options are used to reduce the amount of unnecessary information output from the execution, -q is to hide Veriflyta's option summary and -s is to hide the progress indicator. The query is constructed by retrieving the Simulation Runs and the Input Query Paramaters and Output Query Parameter from their
In order to store and easily retrieve the output data, the data structure in the MATS tool is organised in a tree structure where each model is stored as a `TraceManager` object which contains a list of `Parameter` objects representing each simulated parameter. A `Parameter` contains the name of the parameter, and a list of list of points, where the first list represents each simulation and the second list is the list of points containing the actual data. Figure 13 illustrates a model which has been simulated two times and has three input and output parameters.

![Figure 13](image)

**Figure 13.** Overview of the TraceManager in the MATS tool used to store the simulated model output.

The tool takes advantage of multithreading as each Verifyta simulation is independent of another and can thus be executed on separate threads which improves the tool’s scalability for higher number of test cases and multiple mutations. Currently the MATS tool is used to simulate the Reference Model before any other models are simulated. The reason for this step is that the mutants need to be modified based on the result from the original model and then executed with the right inputs. We mention here that currently only the simulations of mutations can be more efficient by using additional threads.
Figure 14 displays the user interface of the MATS tool. This interface contains several adjustable parameters. The Select button is used for linking the reference model. Using the second Select button, the user needs to choose multiple mutants based on the reference model. The query parameters are adjusted with the numericUpDown controls from the Simulation Runs item, Period Count item, Period Length item. These items are explained as follows.

- **Simulation runs** is used to define how many simulations will be generated on the reference model which corresponds to the number of generated test cases,
- **Period Length** is used to define how long each execution period is. This is represented in time units and is based on the system model's execution period to be simulated by UPPAAL SMC,
- **Period Count** stands for how many periods the simulation will execute. Both Period Length and Period Count are used to generate the Simulation Time. This is calculated based on the following equation: Period Length × Period Count + (Period Length -1),
- **Sample Size** allows the user to choose the granularity of the mutation analysis,
- **Output Delta** sets the delta limit for the output comparison based on the difference between the actual and mutated output signals at each time unit,
- The **Input Query Parameters** item allows the user to define which variables to be monitored as inputs,
- **Output Query Parameter** item allows the user to define which variables to be monitored as outputs.

Both Sample Size and Output Delta values are used to calculate a quantitative measure of fault detection. A mutant is considered to be detected by a test suite if the values differ drastically at certain time points. Sample Size decides how often this detection should be checked while the Output Delta defines how large the output difference can be.

Figure 15 is an example of a mutation result. When the result of the mutation analysis has been generated the final result will be shown. The selected tests to be used on the real system are highlighted in green. This set of tests is the minimum test case needed to maximize the mutation score. The Mutation Result in the MATS tool can be saved to a text file for later use.
7. Case Study: The BBW Prototype System

In this section, we will present the use of our framework and the MATS tool on the BBW prototype system and the results from this case study.

7.1 The BBW Prototype

The BBW prototype is a braking system equipped with an anti-lock brake system (ABS) and is controlled electronically instead of mechanically. A sensor is attached to the brake pedal to read its position which is used to compute the desired force applied to the brakes. With additional sensors to measure each wheel's speed, the ABS algorithm and the desired brake force the actual brake torque is calculated and applied. This prototype system was originally described in EAST-ADL [19], an architectural language dedicated to automotive systems with support for resource annotations. A transformation from EAST-ADL to UPPAAL SMC was proposed by Marinescu et al. [20] in 2014.

The BBW Prototype used in this thesis is a result of that transformation and is then manually extended with energy information. The transformation generates two TA for each EAST-ADL component, one based on the component’s interface and one based on the component’s behavior. For more details about this transformation, we refer the reader to the original paper [20]. We show in Figure 16 and 17 an example of a pair of TA for the pedal sensor that demonstrates this annotation, where pBrakePedalSensor_e is a (clock) variable that stores the energy consumed by the TA pair.
To perform mutation analysis, we currently create each mutant manually as a modified version of the original model and each fault is inserted into respective mutants by a human. An example of a mutation is illustrated in Figure 18 where the pBrakePedalSensor_e’ rate of consumption is increased by an additional 0.5 units from the original value of 2 while being in Exec location.

Each mutant is also altered to follow a predefined set of inputs. Figure 19 displays how the brake pedal is controlled by a predefined set of values in this particular model.
Once the models are provided and modified with the necessary modifications the framework and the MATS tool will work automatically.

### 7.2 Case Study Results

To perform the validation of our framework, we use the MATS tool to generate test suites, measure the time needed to generate tests and the mutation coverage reached with different parameters. For example, as shown in Figure 20, for a set of parameters (*Simulation runs: 25, Simulation Time 64, Sample Size 0.050, Output delta 4.0*), a test suite will be automatically generated and suggested to the engineer. The test suite generated for the BBW system is a single test case that kills all mutants which results in full mutation coverage. In order to fine tune these parameters and show the applicability of this approach we repeated this process for different parameters. The reference model selected in our case study is the BBW model which contains a network of 50 timed automata divided into 25 pairs where 16 of these are computational blocks. This model represents an industrial brake by wire system developed by industrial engineers. Five manually created mutants of the BBW model (modified as described in Section 7.1) are used in this case study.

![Figure 20. Example of a test selection result.](image)
The following tool parameters were used to obtain the measurement results:

- **Simulation Runs:** 25, 50, 100, 200, 400. This values should show how the number of simulation runs can influence the number of tests and how easy is to achieve full mutation coverage for the BBW system.
- **Simulation Time:** 64. This value was chosen based on the BBW model and its full system execution. Practically this represents the calculated end-to-end deadline for our model.
- **Sample Size:** 0.05. This value was chosen to detect differences in the energy signal at small intervals.
- **Output Δ:** 5.2. This delta was selected based on our experience with verifying and analyzing the BBW system and should show if the energy noticeably diverges from the expected result.
- **Inputs:** `pBrakePedalLDM_ElSignalIn`, `pLDM_Sensor_FL_TicksIn`, `pLDM_Sensor_FR_TicksIn`, `pLDM_Sensor_RL_TicksIn`. These are the controllable variables in the BBW model which models the input received from the brake pedal sensor and the wheel sensors.
- **Output:** energy. This is the output signal modeled in the BBW system.

Each set of simulation runs was executed five times in order to prevent random results that could skew the achieved mutation score. Because the test case generation is using a random algorithm used by UPPAAL SMC to simulate the system, there is a chance that the generated test cases could be better or worse in obtaining full coverage depending on how many times we run the generation process. In the best-case scenario generating one test case could be sufficient to obtain full mutation coverage.

The results of our measurements are shown in Figure 21. This plot shows that the average mutation coverage keeps increasing until it reaches 100% when generating 400 test cases. This means that in our case when using 400 runs we can obtain a set of test cases that would detect all of our mutants. Generating these 400 test cases would take about 277 seconds on average when using the MATS tool on an Intel i5 series 4.5Ghz processor with 8GB of RAM. This demonstrates that this framework for test generation is relatively efficient and easy to use, and thus answering our third research question. We note here that the generation time increases linearly with the number of simulations for our particular case. In conclusion, the proposed method is applicable on an industrial system of similar size as the BBW system.

![Figure 21. The test generation results on the BBW system; X-axis represents the number of test cases generated by the MATS tool (i.e., 25, 50, 100, 200, 400).](image)
8. Discussion

The obtained results could be useful for both industrial engineer, test generation tool developers and researchers. To further explore the results of our case study we consider the results of our thesis and our experiences. By proposing an integrated framework and tool for automated test generation using UPPAAL SMC and performing a case study we have been able to answer our research questions. While we investigated the first research question we discovered that sometimes the output data generated by UPPAAL SMC is not updated instantaneous. In Figure 7 we can observe this behavior for the first parameter in both simulations. After the time has elapsed 20 time units, the value for the parameter is supposed to change to value 2 instantaneously, instead it changes over a span of 0.0152 time units. This affects the input extraction because the model is not always changing the input values at the predefined time without any delays. This prevented us from having the same sample size as the period length. To overcome this, we sample at an offset to get clean and accurate values. We chose to set the offset to half of the period length to be sure that we extract the correct input values.

Our framework does not rely on UPPAAL SMC's built in simulation sampler. We found out that this sampler does not support resampling without simulating the model again. To overcome this, we extract the raw data output and we manipulate the data later using our own MATS tool.

The results from this thesis indicate that this framework of test generation is very efficient given the complexity of the industrial model used to validate the MATS tool. The generation time measurements were obtained using a four-threaded processor. Additional cores and threads would be needed to decrease the generation time and enable the generation of more test simulations in less time. The fact that automated test generation using MATS tool is efficient in terms of generation time stands as a significant progress in aiding engineers performing testing of this kind of systems. It is also important to note that our framework can handle continuous resources which sets our work apart from the previous works.

There are still some limitations in the MATS tool that the framework is not limited to. The most obvious limitation is the result interface which can be improved to get a better overview of the selected test cases. The most useful addition would be the ability to sort the test cases based on different properties and show alternative test selection recommendations. Another important limitation for the framework is related to the model flexibility. In this thesis we assume that all the inputs have the same period when MATS tool generates the inputs for test cases. This may not always be the case for other models. Currently the tool only supports one output parameter, which limits the tool to only evaluate one continuous resource at a time. The framework does have support for evaluating several continuous resources at a time but was omitted from the tool because of time constraints. Another useful function would be to adjust the delta value on the already simulated test cases. This would enable the engineer to better determine which test cases have the largest confidence levels or which test cases are close to the original delta value. Another addition would be to also adjust the sample size on the fly to see how it affects the result.

9. Conclusions

UPPAAL SMC is a tool used to model and statistically evaluate systems modelled in timed automata. Currently, this tool is lacking the ability to automatically generate and select tests using mutation analysis. In this thesis we proposed a framework and implement it in the MATS tool. In addition, we evaluated this tool in order to answer our three research questions.

- The first question was: How to generate test cases for testing resource consumption in embedded systems by using UPPAAL SMC? This thesis demonstrates one way to generate these test cases which is described in Section 6.2.
- The second question was: Using mutation analysis as the underlying technique, is it possible...
to minimize the test suite while retaining the same fault detection? Section 6.4 presents a method that shows that it is possible to minimize the test suite automatically. This is also integrated in the framework.

- The third question was: *Are the proposed solutions applicable on an industrial system?* The validation on the BBW prototype indicates that it is indeed applicable on an industrial system with encouraging results. However, the framework needs to be further validated on other systems.

In addition to the future work stated in the Discussions section, there are other possible future works related to this thesis. Our framework is currently missing the ability to generate mutants automatically. There is also a need to evaluate what kind of mutation operators can be created and how these operators can be automatically applied on a timed automata model. Our validation has been limited to the BBW system and several manually created mutations. Further case studies and improvements in the framework are still needed. In addition, making the tool platform independent is also of interest, mainly to support the same operating systems as UPPAAL SMC does. The tool could be improved by evaluating more than one output signal and use other test selection criteria for test suite minimization.
References


