Model-Based Test Case Generation in Ecdar

Offline Conformance Testing Using Refinement in Ecdar

AALBORG UNIVERSITY
STUDENT REPORT

Master Thesis

cs-23-sv-10-01

Aalborg University
Department of Computer Science

AALBORG UNIVERSITY
STUDENT REPORT

Department of Comp. Science
Aalborg University
https://www.aau.dk

Title:
Model-Based Test Case Generation in Ecdar

Scientific Theme:
Master Thesis

Project Period:
SW10, Spring 2023

Project Group:
cs-23-sv-10-01

Participant(s):
Esben Rask Bach
Yann Bernard Bastrup Perruchon

Supervisor(s):
Florian Lorber

Copies: 1

Page Numbers: 43

Date of Completion:
June 8, 2023

This project presents an offline model-based testing approach of real-time systems using Ecdar. A test suite is generated that can be run locally on a system. Firstly, the timed input-output automaton was extended to contain test code in locations and edges. A new issue arose: how to handle clocks in the test code. This was solved by using zones, and asserting that the symbolic values are valid in the test code. Each test case in the test suite is generated by reachability queries to each edge, and extending the trace to the next output edge. The approach was tested in three different case studies, and evaluated using PIT mutation testing on the system. The case studies revealed that the approach was able to detect all the refinement violations introduced by mutants in the system under test.

Table of Contents

Preface

Esben Rask Bach
<erb18@student.aau.dk>

Yann Bernard Bastrup Perruchon
<yperru18@student.aau.dk>

Summary

This report gives a comprehensive overview of a project about developing an offline model-based conformance testing approach for real-time systems using Ecdar. The aspiration of the project is to automate the testing process, enhance coverage, reduce cost, and improve reproducibility. The project aims to generate a test suite that can effectively detect refinement violations between the System Under Test (SUT) and the specification.

The introduction gives an overview of the importance of software testing when ensuring that a system behaves as the specification dictates. The limitations of manual testing is discussed, and model-based testing is presented as a promising solution. However this project pertains the testing of real-time systems, and such the models have to be extended to timed models. The problems relating to testing real-time systems is acknowledged, as well as the challenged introduced when handling timed models. Existing tools and techniques are identified, and the strengths and weaknesses are evaluated. This identifies the need for introducing clock variables into the Yggdrasil functionalities.

After the problem analysis, the problem statement was narrowed down to: How can a test suite be generated using Yggdrasil-like functionality in Ecdar, that has good coverage and can detect refinement violations within the covered behavior of the SUT?

The first step was to extend the timed input/output automata to include test code directly in the model. Also the definition and valuation of test cases was defined. A general overview of the approach was then described. The test suite creation is presented, in the form of pseudo code for the algorithm, along with a description of the algorithm. The creation of the test code in the models is also described, along with the challenges associated with it.

The effectiveness of the approach was evaluated by conducting three case studies. three different systems were implemented, and a test suite was generated for each of them. The implementations were mutated by using PIT mutation testing. Different configurations of the test suite generation was tested, and evaluated against each other by looking at the mutation coverage. The case study concludes a good mutation coverage, and indicates a consistent ability to detect refinement violations.

The discussion section touches upon various considerations and trade-offs of the approach. The considerations and trade-offs being: Differences to real-time vs symbolic time representations, the use of BVA when creating a test suite, the challenges of deriving test cases from from an input-enabled specification and the soundness and portability of the approach.

The conclusion section summarizes the main contributions of the project. The primary achievement being the development of an approach to test for refinement between a real-time system and a specification, in an offline approach.

Lastly, the future work of the project is suggested. This includes: Testing on larger models, testing compositional models, switching from symbolic time to real-time, and adding an inconclusive verdict.

1. Introduction

Software testing is the process of evaluating and verifying that a system behaves as expected. It is a crucial part of software development, as it prevents bugs, reduces the cost of development, and improves performance. The most used testing technique in the industry is manual testing. However, manual testing is a tedious and error-prone process. Model-based testing makes it possible to automate the process, by deriving test cases from a model. This has several advantages over manual testing; it can improve the coverage of the system's behavior, and reduce the cost of testing, it adds structure to the testing process and improves reproducibility [18]. When testing real-time systems, models can be extended to handle the passing of time by introducing clock variables. Timed models have a greater order of complexity than non-timed models because of the state-space explosion; there are infinitely many possible clock valuations. Furthermore, they also complicate the creation of test cases as they, in addition to inputs and outputs, need to also consider time.

UPPAAL is a tool for modeling and verifying real-time systems [17]. Yggdrasil is an extension of UPPAAL and is able to produce test cases in the form of test code, reminiscent of unit tests, that can be executed by the SUT [1]. Unit tests are for small units of code and focus on singular functionalities, the tests derived from Yggdrasil are more comparable to system testing. This circumvents the need for a test driver and allows the testing to be easily accessible and re-runnable within the development environment. A problem becomes apparent when this approach is used on real-time systems, as there are no features available for accessing clock variables in UPPAAL, and therefore no way to observe whether the SUT operates within specified time constraints.

The literature review by Nidhra et al. [12], considers different black-box and white-box testing techniques. White-box testing assumes that the internal state of the system is observable, a typical example of a white-box testing technique is Unit testing. Black-box tests are typically created purely from the requirement specification and work under the assumption that the internal state of the system is non-observable. Model-based testing is inherently tied to black-box testing, as a model is an abstract representation of the system derived from a requirement specification. In model-based testing black-box testing often involves a test driver that executes the test case on the System Under Test (SUT). This is done by sending inputs, awaiting outputs, and measuring the time between them.

Ecdar is a tool for compositional design that implements the specification framework for real-time systems presented by David et al. [5]. Ecdar uses Timed I/O Automata to model systems, and implements various operators to support compositional design such as: refinement, logical and structural composition, quotient, and consistency checking. These operators are used between two models: a specification and an implementation. Refinement specifically can be considered a conformance relation, as it defines rules that need to be adhered to between an implementation and a specification.

Ecdar currently lacks test generation capabilities therefore this project will look into implementing Yggdrasil's test case generation functionality in Ecdar, and extending it so that it can be used to test real-time systems. Finally, the project will attempt to create a test suite that can detect whether a SUT conforms to the behavior specified in the model.

2. Related Work

In this chapter, related work to the problem will be presented. All the papers concern model-based testing. For a general overview of model-based testing, the following taxonomy by Utting et al. can be observed [18].

The tool T-UPPAAL was created by Larsen et al. [11], for online black-box testing of real-time embedded systems from non-deterministic timed automata specifications. A randomized online testing algorithm was implemented in the tool. A new notion of relativized timed input/output conformanc was presented in the paper, and was utilized as the basis for the online testing algorithm. The tool was tested and showed promising results in terms of error detection capability and computational performance.

Krichen et al. propose a new framework for black-box conformance testing of real-time systems [10]. The tests are based on models of partially-observable, non-deterministic timed automata. Two types of tests are described: analog-clock tests and digital-clock tests. Both types of tests are generated by an on-the-fly algorithm. The output of the test generation is test cases in a tree-like structure. Using a prototype tool called TTG, and applying it to a Bounded Re-transmission Protocol, shows that only a few test cases are sufficient to cover thousands of reachable symbolic states in the specification.

Hessel et al. developed a tool called Uppaal Co✓er [9]. The tool is a test case generation tool for timed systems. The test cases are derived from a timed model, and some predefined coverage criteria are expressed in an observer language. The algorithm used to generate the test suite is described by Hessel et al. [8]. The algorithm stores the coverage items globally when creating a test case. The tool was used in a large case study, where a WAP gateway was tested. Co✓er created a test suite with *full feasible* coverage of edge coverage, switch coverage, and projected state coverage.

The documentation for Uppaal Yggdrasil can be observed [1]. The main features are that the generated test code is backend agnostic. This means that the test code is user-defined, and is independent of the specific device the test code is executed on. There are three modes for generating test cases: Test purpose mode, auto-depth mode, and singe-step mode. Test purpose mode generates test cases specified by a reachability query. This is relevant for this project, and should also be implemented.

A model checker tool, Ecdar [5], was developed on top of UPPAAL-Tiga [3]. Ecdar is an implementation of a complete specification theory. Specification theory combines notions of specifications and implementations with a satisfaction relation, a refinement relation, and a set of operators that support stepwise design. Ecdar also provides constructs for refinement, consistency checking logical and structural composition, and quotient of specifications. However, Ecdar does not have any test case generation capabilities.

The paper by Gunderson et al. [6] proposes an extension of Ecdar that performs conformance testing, using Model-Based Mutation Testing (MBMT). The conformance relation is a refinement check. The adaptive test cases are executed on the SUT which will lead to the desired state. The goal is to find a fault. The test execution can be performed in real and virtual time. The paper uses a test driver to communicate with the SUT.

3. Problem Analysis

This chapter analysis the problem presented in the introduction. Specifically the model-based testing process will be split into 5 steps, and each step will be described. The problems that occur will be discussed, such as input enabled specifications, and zeno behavior. The chapter ends in a problem statement, that lays the basis for the rest of the report.

3.1 Model-Based Testing

Mark Utting et al. [18] define model-based testing as the automatable derivation of concrete test cases from abstract formal models. Model-based testing is reliant on models that encode the intended behavior of a system. The appeal of model-based testing is that it makes the testing process structured and reproducible. The test cases derived from model-based testing are often in the form of sequences of inputs and outputs. In the case of timed models, the test cases should also specify the allowed delays of the SUT.
The process of Model-Based testing can be observed in Figure 3.1. The first step (1) is to build the model based on the requirements.

The second step (2) is to define test selection criteria, also based on the requirements. Test selection criteria describe what is to be tested on a high level. This can be based on specific functionalities of the system (requirements-based), and the structure of the model (state-coverage, transition-coverage, etc.).

The third step (3) is to transform the test selection criteria into formal test case specifications. The notion behind the test selection criteria is distilled into concrete queries to be performed. For example, if the test selection criteria were: "location coverage". Then statements of the form *"reach loc x"* would be produced for each location x in the model.

The fourth step (4) occurs after the model and test selection criteria have been created. One test case is produced for each test case specification. The sum of these test cases

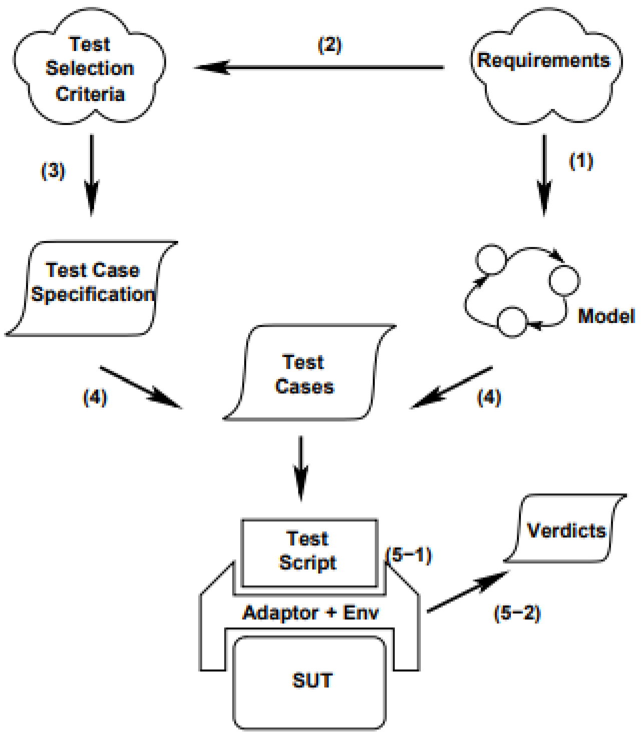

Figure 3.1: The Process of Model-Based Testing

constitutes the test suite. After the test suite generation, the test cases are then executed on the SUT.

The model and the SUT exist on different layers of abstraction, which is why test cases are typically executed by applying inputs and observing outputs and delays. The adaptor illustrated in the figure is used for concretization of inputs so that it can be interpreted by the SUT, and abstracting outputs.

The test script contains executable code that executes the test case on the SUT. It also passes a verdict: *pass*, *fail*, or *inconclusive*. If the behaviour of the SUT conforms to the expected behaviour derived from the model, it would result in a *pass*, if not it would result in *fail*. *Inconclusive* is when neither verdict can be made yet.

3.2 Step 1: Building the model

On Figure 3.1 the process of model-based testing is illustrated. In this section the first step in the process is described. The formal definition of the model used in Ecdar is described.

3.2.1 Timed Input/Output Automata

Timed Input/Output Automata is an extension of a Fine State Machine, that uses clock variables to measure the passing of time. Let X be a finite set of clock variables. A clock valuation $v(x)$ is a function that assigns a real value to a clock $x \in X$. The function is defined as: $v : X \to \mathbb{R}_{\geq 0}$. The set of all clock valuations is denoted by V, and $\mathbf{0}$ denotes the valuation of assigning 0 to every clock. For a valuation v and $d \in \mathbb{R}_{\geq 0}$ we define $v + d$ to be the valuation $(v + d)(x) = v(x) + d$ for all $x \in X$. We denote the subset $x \subseteq X$, by the valuation $v[\delta]$ such that for every $x \in \delta, v[\delta](x) = 0$ and for every $x \in X \setminus \delta, v[\delta](x) = v(x)$.
A clock constraint ϕ is a conjunction of predicates of the form: $x \sim n$, where $x \in X$, $n \in \mathbb{N}$ and $\sim \in \{<, \leq, =, >, \geq\}$. Given a clock valuation v, we write $v \models \phi$ when v satisfies ϕ.
A Timed Input/Output Automaton (TIOA) is a tuple A $= (L, l_0, E, C, B, G, \Sigma, I)$, where

- L is a finite set of locations
- l_0 is an element of L, and is the initial location
- E is the finite set of edges of the form (l, g, u, a, l')
  - l is the source location
  - g is a guard. $g \in G$

- u is the set of variable to be updated, where $u \subseteq C \cup B$
  - A Boolean update is of the form: $b = \{true, false\}$
  - A Clock update is of the form: $c = \mathbb{R}_{\geq 0}$
- α is an action. $\alpha \in \Sigma$
- l' is the target location

- C is a finite set of clocks
- B is a finite set of Boolean variables
- G is the finite set of guards, and each guard is a conjunction of constraints of two forms:
  - $c \sim n$, where $c \in C, \sim \in \{<, \leq, ==, >, \geq\}$ and $n \in \mathbb{N}$
  - b $\sim$ m, where $b \in B$, $\sim \in \{==, \neq\}$ and $m \in \{true, false\}$
- Σ is the finite set of actions, partitioned into inputs (Σ_i) and outputs (Σ_o)
- $I : L \rightarrow LI$ is a mapping of *locations* to *location invariants*, where each location invariant $li \in LI$ is a conjunction of constraints of the form: $true, c < n$, or $c \leq n, b == m, b \neq m$, where $c \in C$, $n \in \mathbb{N}$ and $m \in \{true, false\}$

3.2.2 Timed Input/Output Transition System

The semantics of a TIOA can be described with a Timed Input/Output Transition System (TIOTS). A TIOTS is a tuple $= (S, s_0, \Sigma, \rightarrow)$

- $S = \{(l, v)$, where $l \in L$ and $v \in V | v \models I(l)\}$
- $s_o = \{(l_0, \mathbf{0})\}$
- Σ is the set of observable actions partitioned into inputs (Σ_i) and outputs (Σ_o)
- $\rightarrow \subseteq S \times (\Sigma \cup \mathbb{R}_{\geq 0}) \times S$ is the transition relation consisting of *timed* and *discrete* transitions:
  - *Timed transitions (delay)*: $(l, v) \xrightarrow{d} (l, v + d) \in \rightarrow$, where $d \in \mathbb{R}_{\geq 0}$, if $v + d \models I(l)$
  - *Discrete transitions (jump)*: $(l, v) \xrightarrow{\alpha} (l', v') \in \rightarrow$, where $\alpha \in \Sigma$, if there exists an edge $(l, g, u, \alpha, l') \in E$, such that $v \models g \wedge v' = v[\delta] \wedge v' \models I(l')$
    - Discrete transitions are split into input and output actions, denoted by:
    - Input: $(l, v) \xrightarrow{i?} (l', v')$
    - Output: $(l, v) \xrightarrow{o!} (l', v')$

$(l, v') = (l, v)$ after d denotes the state reached from (l, v) by a delay d. $(l', v') = (l, v)$ after α denotes the state reached by performing action α on state (l, v). If a delay d is not possible in (l, v) we denote it $(l, v) \nrightarrow^{d}$.

3.2.3 Input-Enabledness

A TIOTS is input-enabled iff $\forall s \in S . \forall i? \in \Sigma_i . s \xrightarrow{i?}$. This means that the TIOTS can always accept any of its defined inputs.

Angelic completion [16] or *demonic completion* [4] can be used to transform an automaton to an input-enabled automaton.

3.2.4 Zeno Behavior

Zeno behavior occurs when there is an infinite number of transition can happen in a finite time interval [2]. If the model has zeno behavior, measures have to be taken when creating test cases.

3.2.5 Independent progress

For a specification $T = (S^T, t_0, \Sigma, \rightarrow^T)$ such that for each state $t \in S^T$ we have:

$$\textit{either } (\forall d \geq 0 . t \overset{d}{\rightarrow}^{T}) \textit{ or } \exists d \in \mathbb{R}_{\geq 0} . \exists o! \in \Sigma_o . t \overset{d}{\rightarrow} t' \textit{ and } t \overset{o!}{\rightarrow}^{T}$$

The specification can not get stuck in a state, where it is up to the environment to induce progress [5]. There must either be an output available or delay until an output becomes available. For this project, we assume that the implementation also has independent progress.

3.2.6 Output urgency

If an output is available, then it cannot be delayed [5].

3.3 Step 2: Test Selection Criteria

Conformance relations can be used as the basis for generating a test suite. The purpose of the test suite is to ascertain if the SUT conforms to its specification. The test suite must then be generated in such a way that it checks for conformance. When generating a test suite, some test selection criteria are specified. The challenge then becomes: Which test selection criteria should one use if one wishes to test for conformance?

3.3.1 Refinement

Ecdar stands for **E**nvironment for **C**ompositional **D**esign and **A**nalysis of **R**eal time Systems. Ecdar uses a conformance relation called *refinement*, which is used in Ecdar to compare two input-enabled TIOTS [14]. The refinement relation can be formally defined: A implementation $P = (S^P, p_0, \Sigma, \rightarrow^P)$ refines a specification $T = (S^T, t_0, \Sigma, \rightarrow^T)$, denoted by $P \leq T$, iff there exists a binary relation $R \subseteq S^p \times S^T$ containing (p_o, t_0) such that for each pair of states $(p, t) \in R$, the following rules are true:

- **Input Rule** $Whenever\ t \xrightarrow{i?}^{T} t'\ for\ some\ t' \in S^T\ then\ p \xrightarrow{i?}^{P} p'\ and\ (s', t') \in R\ for\ some\ p' \in S^P$
- **Output Rule** $Whenever\ p \xrightarrow{o!}^{P} p'\ for\ some\ p' \in S^P\ then\ t \xrightarrow{o!}^{T} t'\ and\ (s', t') \in R\ for\ some\ t' \in S^T$
- **Delay Rule** $Whenver\ p \xrightarrow{d}^{P} p'\ for\ d \in \mathbb{R}_{\geq 0}\ then\ t \xrightarrow{d}^{T} t'\ and\ (p', t') \in R\ for\ some\ t' \in S^T$

3.3.2 Structural Coverage Criteria

In order to check for refinement, Ecdar explores the whole state space symbolically and checks whether there are any violations to the refinement relation. It is not feasible to perform the same complete refinement check when comparing a SUT to a specification.

The edges in a TIOA are either associated with an input or output action. Logically this means that in order for the input and output rules to be covered, it is at the very least, necessary to cover all edges.

It is also worth considering whether test cases that end on input are *sufficient*. When giving a system an input it is normal to expect an output in response. Therefore test cases should be extended to finish on outputs. Furthermore, it is worth pointing out that the effect that an input has on a SUT is in many cases not observable by the tester, and can only be observed by an output triggering afterward.

3.3.3 Data Coverage Criteria

In order to check for the delay rule, it is necessary to include instructions in the test code that pass time in order to trigger outputs or to delay before applying an input. The actual time that should be delayed is then the question.

3.3.3.1 Boundary Value Analysis

Delays for input and output edges are inherently different as it is only possible to control the delay of inputs. There are two delay cases for inputs, either there is a limit to how much you can delay before applying the input or there is not. Most likely the lower bound of the delay will be 0, meaning that no time passes. However, it could be the case that some time needs to pass before the input will be accepted by the system.
Therefore in order to properly test the SUT, there needs to be a variance in the delays. Boundary Value Analysis (BVA) works under the assumption that behavior within a boundary is equivalent [12]. BVA separates the values into partitions, where the partition can either be **valid** or **invalid**. BVA normally partitions the values as follows:

1. Just below the minimum value (**invalid**)
2. The minimum value (**valid**)
3. The maximum value (**valid**)
4. Just above the maximum value (**invalid**)

For example, if some SUT accepts delays in the range of 1-10 seconds in which an input can be given, then by using BVA only four test cases need to be considered, two valid and two invalid: [0,1,10,11]. It is necessary to consider the invalid delays, as the **Delay Rule** states that any delay performed in the *implementation* should also be valid in the *specification*.
There are four cases that need to be considered when choosing a delay for an input.

- The input can arrive at an interval of delays where the input can arrive without delay e.g. x <= 5.
  - Valid delays: 0 and 5. Invalid delays: 6
- The input can arrive at an interval of delays where it must delay before arriving e.g. 2 <= x <= 5
  - Valid delays: 2 and 5. Invalid delays: 1 and 6
- The input can arrive at one specific point in time e.g. x == 5.
  - Valid delays: 5. Invalid delays: 4 and 6
- The input has no constraints on when it arrives e.g. x >= 0.
  - Valid delays: Any

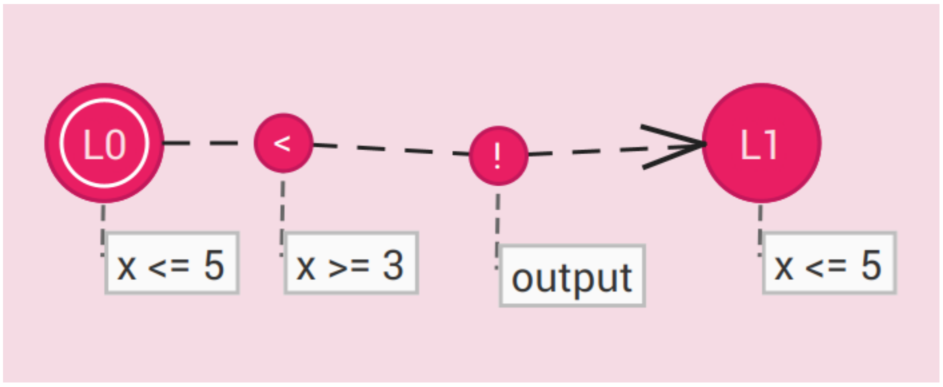

Figure 3.2: Simple Model

3.3.3.2 Clock Values Extension

As we intend to extend the Yggdrasil functionality so that it can be used on real-time systems, clock values need to be accessible. As described in the definition of TIOTS section 3.2, a state $s = (l, v)$, is a location and a clock valuation. Often it is the case that a clock can have a range of possible values in a location. This means that in order to properly encapsulate the range of clock values, the *zone* needs to be used. For example, consider Figure 3.2: The zone of clock values in $L0$ is $(x <= 5 \&\& x >= 0)$ and $L1$ is $(x <= 5 \&\& x >= 3)$.
Figure 3.2 can be considered a specification. It states that it is possible to delay for 3, 4, and 5 seconds before the output occurs. If for example that the output occurred after 2 seconds, then that would be outside the zone of $L1$, and thus the violation of the delay rule would be caught.

3.4 Step 3: Test Case Specification

When the test selection criteria have been selected, they need to be formalized in order to generate test cases. The formal test selection criteria are called *test case specification* [18]. For example, if the test selection criterion is that we want to cover a location, the test case specification in UPPAAL would be "E<> Model1.L1". The syntax varies, depending on which model checker is used. The statement essentially means "check if there is a trace where Model1 is in the location L1". The next step is to repeat the statement for each location in the model. Each of these statements produces one test case.

In this case, the reachability queries concern the edges of the model: one query should be formulated for each edge. If it is an input edge the query should be expanded to continue until it reaches an output.

In order to formulate queries that cover edges, the model in question must be modified. Firstly a boolean variable called *boolVar* should be added to the model. Then, a boolean assignment should be added to the edge to be covered, that assigns the *boolVar* to *true*. The query in UPPAAL would be: "E<> boolVar == true". The assignment is then removed, and the process begins again with another edge.

3.5 Step 4: Test Case Generation

The test case generation is divided into two phases: trace generation and test code extraction [18]. Traces are generated based on the test case specification. Test code can then be extracted from transitions in each trace, creating a test case for each trace. Variants of the test cases can be made that implement the delays that are computed by performing a BVA on the model.

3.5.1 Yggdrasil Test Code Customization

The test code is divided between Enter Location, Exit Location, and Edge. Yggdrasil also offers customizability when generating test cases [1]. On Code example 3.1 the customization option can be observed. Yggdrasil creates a file for each test case, by using `TEST_FILENAME` and `TEST_FILEEXT` it is possible to define the name of the file and to define the file extension. This is shown on lines 1-2 where the filename is defined as `TestCase-`, which is automatically supplemented with the test case number e.g. *000*, *001* etc. The file extension defines the file as being a Java file.

`TEST_PREFIX` is used to define what comes *before* the test code from a trace. In Code example 3.1 it is used to define the package, the imports, and a class `TestCase`, which can execute the test by running the main method. The test code from the trace will then be inputted after the `TEST_PREFIX`. If something is defined in `TEST_INFIX`, it will be added between each line of test code. `TEST_POSTFIX` is appended after the test code, in the case of Code example 3.1 it is used to close off the scope of the class and method with curly brackets.

```
1  /** TEST_FILENAME TestCase- */
2  /** TEST_FILEEXT .java */
3
4  /** TEST_PREFIX */
5  package app;
6  import app.App;
7
8  class TestCase extends App
9  {
10     public static void main(String[] args)
```

```
    {
        expect_on($(System.val));


/** TEST_INFIX */

/** TEST_POSTFIX */
        expect_on($(System.val));
    }
}
```

Code example 3.1: Test Customizability in Yggdrasil

3.5.2 Extending Test Code Customization

The test code extracted from a trace is already user-defined and is therefore expected to be compatible with the syntax of the SUT. However since the goal is to test for refinement, there need to be recurring assert statements to verify that the SUT is operating within the constraints set by the specification.

We intend to automate the process of verifying whether the SUT is working within the timed constraints of the specification, by creating assert statements *after* Enter Location code and *before* Exit Location code. This necessitates additional input from the user such as the syntax for an assert function and for accessing variables if they are encapsulated inside a class for example.

3.6 Step 5: Test Execution

The tests are intended to be executed locally in the SUT's development environment similar to a unit test. The verdicts are then passed locally. If all *assert* statements in the test are true, then the test passes. If a single assert statement is false, then the test fails.
The formal process described by Utting et al. [18], used an adaptor. The adaptor is used to *translate* inputs and outputs. This is necessary because the Test Script and SUT exist on two different abstraction layers, which in most cases needs to be *translated* so that they can communicate properly.

By using a Yggdrasil-like functionality the need for an adaptor is bypassed. The assumption is that the test code that is produced is already on the same abstraction level as the SUT.

3.7 Problems

This section describes some of the model-based testing problems that are relevant to our project.

3.7.1 Input-enabled Specification

Refinement, as defined by David et al. [5] concerns two input-enabled models. The intuition is that a real-time system can at any time receive any input, however, this does not mean that the input is accepted by the system. By making the model input-enabled this behavior is encapsulated in the model.

A method of making a model input-enabled to create self-loops for inputs in each location. This means that an input can be accepted, but there is no change in the current state. The problem arises when creating traces that involve the self-loops, as they both exit and enter the location resulting in duplicate code. This also goes against the notion of the inputs having no effect on the system.

Therefore we will not be using input-enabled models to derive test cases.

3.7.2 Zeno Behaviour in the Specification

Zeno behavior occurs when there is an infinite number of transitions that can happen in finite amount of time. In Figure 3.3, a cut-out of a larger model called Car Alarm System can be observed. This model has zeno behavior, it is possible to go from L0 to L2, and back to L0 without any time passing in the model. However when this translates to a test case, and the test case is being executed on the system, time passes.

Zeno behaviour needs to be accounted for, either by modifying the model to be non-zeno by demanding that 1 time unit passes before taking a transition, or by using symbolic time.

3.8 Problem Statement

The problem analysis considered how model-based testing could be performed by using a Yggdrasil-like functionality to test for refinement conformance between a SUT and a specification. Test Selection Criteria were defined to adequately cover potential refinement violations, which were then distilled into Test Case Specifications,in the form of reachability queries as seen in UPPAAL. These queries generate traces, which are distilled into test code that will be executed directly on the SUT as unit tests. This leads to the following problem statement:

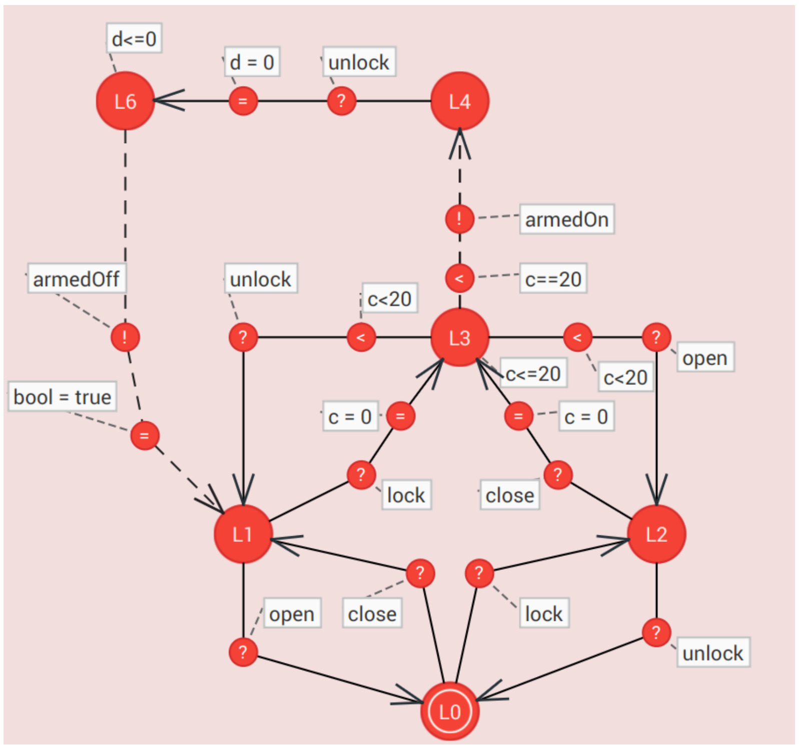

Figure 3.3: Car Alarm System cut-out in UPPAAL

How can a test suite be generated using Yggdrasil-like functionality in Ecdar, that has good coverage and can detect refinement violations within the covered behavior of the SUT?

4. Methodology

This chapter describes the solution to the problem statement. The Timed I/O Automate first has to be extended to incorporate test code into the model. The definition of a test case is then defined. A diagram of the whole solution is presented and described.

4.1 Extending TIOA

In section 3.2 The formal definition of Time I/O Automata was defined. As mentioned in chapter 1, this project uses a different notion of test cases compared to most other model-based testing approaches. In order to accommodate for this notion, the formal definitions must be extended.

4.1.1 Timed Input/Output Test Code Automata (TIOTCA)

The extended Timed Input/Output Test Code Automata (TIOTCA) is a tuple A = $(L, l_0, E, C, B, G, \Sigma, I, tc_{en}(L), tc_{ex}(L), tc(E))$, where

- $tc_{en}(L) : L \rightarrow L_{en}$ is a mapping of *locations* to *enter test code*, where each $l_{en} \in L_{en}$ is a string that can also be the empty string
- $tc_{ex}(L) : L \rightarrow L_{ex}$ is a mapping of *locations* to *exit test code*, where each $l_{ex} \in L_{ex}$ is a string that can also be the empty string
- $tc(E) : E \rightarrow E_{tc}$ is a mapping of *edges* to *test code*, where each $e_{tc} \in E_{tc}$ is a string that can also be the empty string

4.2 Definition of a test case

A test case is the test code collected along a trace through the model, starting in the initial location and leading to a predefined target location. The trace is a sequence of edges. We describe a trace $\omega = \{(l, g, u, \alpha, l')_1, ..., (l, g, u, \alpha, l')_n \mid (l, g, u, \alpha, l')_i \in E \wedge l'_i = l_{i+1}\}$ as a trace with n number of edges. The test case Ψ of ω is then evaluated to:

$$\Psi(\omega) = tc_{ex}(l)_1 + tc((l,g,u,\alpha,l')_1 + tc_{en}(l')_1 + tc_{ex}(l)_2 + tc((l,g,u,\alpha,l')_2 + tc_{en}(l')_2 + \\ ... + tc_{ex}(l)_n + tc((l,g,u,\alpha,l')_n + tc_{en}(l')_n, \text{ where } + \text{ is a concatenation function of strings.}$$

Additional assert statements are inserted into the test code, that define the zone that the clocks can be in. The assert statements are intended to detect violations in the delays of the SUT. They are placed when entering and exiting a location.

4.2.1 Valuation of a test case

A test case consists of test code collected along a trace, at each edge in the TIOTCA. Each 'piece' of test code can either *pass* or *fail*. Thus, we formally define a valuation function ν that valuates test code from an edge (l,g,u,α,l'):

$$\nu(tc_{ex}(l) \in \mathbb{B}$$
$$\nu(tc(E) \in \mathbb{B}$$
$$\nu(tc_{en}(l') \in \mathbb{B}$$

$\mathbb{B}$ is the set of boolean variables, which is *TRUE* and *FALSE.* The valuation is $TRUE$ if the test code passes, and $FALSE$ if it fails. We introduce the notion of valuating an entire test case $\Psi(\omega)$ of n edges:

$$\nu(\Psi(\omega)) = tc_{ex}(l)_1 \wedge tc((l,g,u,\alpha,l')_1 \wedge tc_{en}(l')_1 \wedge tc_{ex}(l)_2 \wedge tc((l,g,u,\alpha,l')_2 \wedge \\ tc_{en}(l')_2 \wedge ... + tc_{ex}(l)_n \wedge tc((l,g,u,\alpha,l')_n \wedge tc_{en}(l')_n = TRUE|FALSE$$

4.3 Overview of the solution workflow

On Figure 4.1 a workflow diagram of our approach can be observed. The first step is to create a model and a system from the requirements. Then a test suite is created from the model. The test suite in its basic form consists of one trace for each edge in the model, such that all edges are covered. The traces that end on an input are then extended so that end on an output. The traces that are prefixes of other traces are then eliminated [8]. Boundary Value Analysis is then applied to the remaining traces, this results in variants of existing traces being created which feature different delays. The traces that are left are now the updated test suite. The test cases Ψ are then incorporated into the SUT, and a verdict is passed.

4.4 Creating a Test Suite

This section describes the process of creating the test suite. The creation of traces through reachability queries, the elimination of *duplicate* traces through prefix elimination. The extension of traces that end on an input, and finally the creation of trace variants that test for values derived from boundary value analysis.

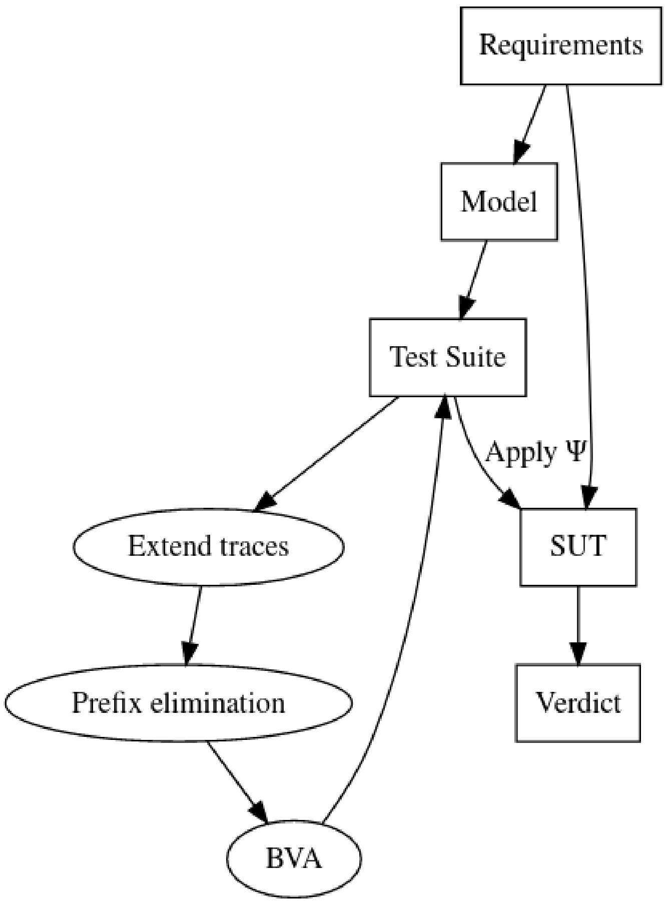

Figure 4.1: Workflow of the solution

4.4.1 Generating and Extending Traces

The pseudo-code observable in code example 1 describes the trace-creation process. Given TIOTCA = $(L, l_0, E, C, B, G, \Sigma, I)$. For each edge $e \in E$ an assignment is added to u, the set of variable assignments for e, which assigns $boolVar$ to $true$. A trace is added to the set of traces by calling the function $RQ(boolVar == true)$, which returns a trace to a state where the input parameter is valid. If the trace ends on an input, the trace is extended so that it ends on an output. The assignment is then removed from u. Line 13 uses the function $prefixElimination()$ to reduce the amount of traces. The function iterates over all traces and makes sure there are no pair of traces $(\sigma, \sigma') \in traces$ such that σ is a prefix of σ' or the other way around [8].

Algorithm 1 Pseudo code for the GenerateTraces algorithm

```
 1: $List < Trace > traces$
 2: $B \leftarrow B \cup \{boolVar\}$
 3: for all $edges\ (l, g, u, a, l') \in E$ do
 4:     $u \leftarrow u \cup \{boolVar = true\}$
 5:     $trace \leftarrow RQ(boolVar == true)\}$
 6:     if $trace.lastTransition\ is\ input\ transition$ then
 7:         $tempTrace \leftarrow extendToOutput(trace.lastTransition)$
 8:         $trace \leftarrow trace + tempTrace$
 9:     end if
10:     $u \leftarrow u - \{boolVar = true\}$
11:     $traces \leftarrow traces \cup \{trace\}$
12: end for
13: $traces \leftarrow prefixElimination(traces)$
14: $traces \leftarrow boundaryValueAnalysis(traces)$
```

Finally, boundary value analysis is performed on the remaining traces. The goal of using BVA is to show that the implementation does not allow behavior outside its timed constraints. This is done by creating traces that attempt to delay for longer, or shorter, time than is allowed for by the specification. This relates to invariants and guards, which define the bounds of timed behavior.

BVA will be performed on guards. The first step is to find a trace that contains a transition where the BVA is applicable. Then copies of that trace are made, each containing different delays. This will impact the size of the test suite: each guard in the model will create four additional test cases.
Finally, the assert statement following invalid delays should be negated, as they are expected to fail.

4.5 Creating Test Code for CAS Implementation

This section describes the creation of the test code for the Car Alarm System implementation. This is a visualization of the approach described in the previous sections.

4.5.1 SUT - CAS Implementation

CAS is implemented in Java and is contained within one class. A method has been implemented for each input and output as well as a `wait(int delay);` method, which will be used to *pass time* symbolically. Additionally boolean variables: armed, locked, flash, sound, and closed are used to keep track of whether an input or output has occurred. More formally it means that the internal state of the SUT is observable.

The enum Loc represents the changing locations. On Code example 4.1 the `lock()` and `wait()` methods can be observed. It is intended that the test case will only call the input methods, and use the wait method to trigger the output methods indirectly. On line 8-17, a switch checks the current location and performs the appropriate actions. Line 20 represents the inputs and outputs of the SUT, that have been omitted from this code example. On line 35-37, the clock values of the system are incremented by the value `int delay`. On line 23 a switch statement checks for the current location, and then checks if the current clock values will trigger an output.

```
1  public class CAS {
2      boolean armed, flash, closed, locked, sound = false;
3      int c, d, g = 0;
4      enum Loc { L0, L1, ..., L16};
5      Loc location = L0;
6
7      void lock() {
8         switch (location) {
9             case L1:
10                locked = true;
11                location = Loc.L3;
12                break;
13            case L0:
14                locked = true;
15                location = Loc.L2;
16            default:
17                break;
18        }
19    }
20   ...
21    public void wait (int delay){
22        while (delay >= 0) {
```

```
            switch (location) {
                case L3:
                    if (c == 20) {
                        armedOn();
                        return;
                    }
                    break;
                ...
                default:
                    break;
            }
            if (delay > 0) {
                c++;
                d++;
                g++;
                delay--;
            }
        }
}
```

Code example 4.1: CAS Implementation

4.5.2 Wait Method

During the test code generation, integer values are computed for `wait()` method calls in the test code. This value has so far been based on the *fastest* possible transition. This is acceptable for specifications like CAS, which defines one single viable time for an action to occur. However, if the specification defines a range of valid clock values $x >= 0 \&\& x <= 5$. The delay would be `wait(0)`. This delay does not capture the range of delays that are allowed by the specification. An implementation could have the action occur at any point between 0 and 5.

To properly consider the whole range of valid delays `wait` should delay for the maximum delay: `wait(5)`. This should be combined with early termination of the `wait()` method, meaning that it only waits until an output occurs in the implementation.

4.5.3 Synthesizing Test Code

In section 3.5 the assert function was described. The usage of the assert function is to take an expression as input and evaluate the expression to be either true or false. If all assert statements in a test case are true, the test case should pass. If just one assert statement is evaluated as false, the whole test case should fail. This functionality is available in the JUnit framework [15]. There exist equivalent features in different programming languages, it is up to the user to define what assert statement should be used. Therefore the expression that the function takes as input is the only thing that has to be determined.

Asserts for clock values should be placed when entering and leaving a location. The input should be boolean expressions defining the zones that clocks can be in for a specific state. It is also assumed that the internal state of the system is observable. Each location has possible combinations of boolean variables, symbolizing the outputs and inputs that have occurred, which will also be asserted. For example, location L0 would be `assert(!armed && !locked && !closed && !sound && !flash);` Such an assert has been written into the test code field of each edge. For CAS we used boolean variables to define the state that the system is in, but it may be described in other ways as well. The intention is to have a *state* that changes that can be observed.

The method calls for input actions that have also been manually written into the test code of the appropriate edges, examples are present on lines 8 and 15. The rest of the test code is automatically generated, although it is necessary to define the syntax for the different method calls in the test code.

Having a method like `wait()` is important. Intuitively it is a way to give control to the system from the test so that it can pass time and possibly trigger outputs. Otherwise, the test would just rush through its instructions without passing time or giving the system the possibility of producing outputs.

Code example 4.2 features the test case for the following trace:

$$\begin{gathered}(L0, c == 0) \xrightarrow{lock?} (L2, c == 0),\\ (L2, c == 0) \xrightarrow{close?} (L3, c == 0),\\ (L3, c == 0) \xrightarrow{20} (L3, c == 20),\\ (L3, c == 20) \xrightarrow{armedOn!} (L4, c == 20)\end{gathered}$$

It is initially denoted if a line is automatically added when synthesizing the test code, or if it is manually added, meaning that it is derived from the test code fields in the model.

```
1 @Test
2 public void test0() {
3     //L0 Auto-generated asserts and wait method call
4     assertTrue(d-c<=0 && g-c<=0 && c-d<=0 && g-d<=0 && c-g<=0 && d-g<=0);
5     wait(0);
6     assertTrue(d-c<=0 && g-c<=0 && c-d<=0 && g-d<=0 && c-g<=0 && d-g<=0);
7     //L0 --> L2 edge test code
8     lock();
```

```
    assertTrue(locked && !armed && !closed && !sound && !flash);
    //L2 Auto-generated clock asserts and wait method call
    assertTrue(d-c<=0 && g-c<=0 && c-d<=0 && g-d<=0 && c-g<=0 && d-g<=0);
    wait(0);
    assertTrue(d-c<=0 && g-c<=0 && c-d<=0 && g-d<=0 && c-g<=0 && d-g<=0);
    //L2 --> L3 edge test code
    close();
    assertTrue(locked && !armed && closed && !sound && !flash);
    c = 0;
    //L3 Auto-generated clock asserts and wait method call
    assertTrue(c<=20 && c-d<=0 && g-d<=0 && c-g<=0 && d-g<=0);
    wait(20);
    assertTrue(c<=20 && c-d<=0 && g-d<=0 && c-g<=0 && d-g<=0);
    //L3 --> L4 edge test code
    assertTrue(locked && armed && closed && !sound && !flash);
    //L4 Auto-generated clock assert
    assertTrue(c>=20 && d>=20 && g>=20 && c-d<=0 && g-d<=0 && c-g<=0 &&
        d-g<=0);
}
```

Code example 4.2: Test Code for CAS

5. Case Study

This chapter will describe three case studies. In order to evaluate each case study; Mutation Testing will be described. Mutation testing will be applied to three different systems; Car Alarm System (CAS), Coffee Tea Machine (CTM), and Train Travel (TT). The results of the cases are presented at the end of the respective subsections. An additional experiment is performed on CAS, that does not have an observable internal state.

5.1 Mutation Testing

Mutation testing is performed by introducing errors to the code and then running your test suite on the faulty system [7]. If the tests fail, then the mutant is *killed*, and if they pass the mutant *survived*. The assumption is that if the suite is of good quality, it should be able to *kill* the mutants. Therefore the quality of the test suite can be determined by the percentage of mutants that it kills. Mutation testing proves that the test suite is able to detect faults, as opposed to using test coverage, f.x. line coverage, as criteria for the quality of a test suite. The intention is to create mutants that break the refinement rules and demonstrate that the test suite kills them. We will only consider mutants in terms of coverage, namely how many mutants are killed out of the generated mutants:

$$\frac{number\ of\ killed\ mutants}{total\ number\ of\ mutants}$$

5.1.1 PIT Mutation Operators

PIT is a mutation testing system for Java that automatically creates mutants based on predetermined mutation operators, a plugin is available for IntelliJ that runs the test suite on the mutants. A mutation operator defines which parts of the code should be changed, and how it should be changed. A full list of the mutation operators can be observed in the documentation [13].
The intention is to use mutation operators that produce mutants that break refinement. The following subsections will discuss which of the mutation operators have proven

to violate refinement.

5.1.2 Void method calls

The "void method calls" mutator removes calls to void methods. Creating tests with this mutator should break the output rule, as the output is never produced by the system.

5.1.3 Member variable

The "member variable" mutator goes through classes, and mutates their member variables. The mutation depends on the type of the variable:

- **boolean:** false
- **int, byte, short, long:** 0

This causes the edges to self-loop or removes actions from the edges, thus breaking refinement.

5.1.4 Inline constant

An inline constant is a literal value assigned to a non-final variable. This value is then mutated, based on the type of the variable.

- **Boolean:** replace the unmutated value *false* with *true*, and vice versa
- **Integer:** Replace the unmutated value *1* with *0*, *-1* with *1*, *5* with *-1* or otherwise increment the unmutated value by *1*

This mutator could break refinement in a number of ways. For example, it is defined that some output should occur when `x == 5`, inline constant changes the expression to `x == 6`, thus breaking refinement.

5.1.5 Remove conditionals

The "remove conditionals" mutator replaces conditional statements with *true* and *false*. This will either cause them to always execute or to never execute. In terms of a model, this mutator either removes the guard from an edge, by setting it to *true*, making it possible to pass at any point, or makes it impossible to pass by setting it to *false*. This breaks refinement as it hinders both outputs and inputs from occurring.

5.2 Car Alarm System Case Study

We created mutants using the ALL parameter, which applies all valid mutations to the implementation. This includes the previously mentioned mutation operators that relate to refinement. We are using all possible mutation operators, as that would reveal any refinement breaking operators we might have missed. It also adds to the general evaluation of the strength of the test suite.
The testing is done on CAS, which can be observed on Figure 5.1. Simply explained: CAS models the behavior of a car whose alarm will be armed after being closed and locked. If the car is opened while armed the alarm goes off, triggering sound and flashing lights. The alarm can then be turned off by unlocking the car.

Initial testing showed that it is necessary to not only extend traces that end on input but to extend all traces to have satisfactory mutation coverage. In the following results, *Trace Extension* extends all traces.

5.2.1 Results

Table 5.1 contains 8 different test configurations which show the effect that a combination of *Trace extension*, *Boundary Value Analysis*, and *Prefix Elimination* has on the test suite.

It is apparent that prefix elimination does not affect the mutation coverage, but reduces the amount of test cases. BVA increases the number of test cases but does not affect the mutation coverage. Trace extension increases the mutation coverage. The table can be divided into the upper four rows, which include trace extension, and the lower four which do not. There are 8 surviving mutants for the test suites that include trace extension, and 16 for those that do not.

The 7 remove conditional mutants occur on all test suites. Conditionals such as `if(d == 0)` are changed to `if(true)`, specifically the mutants concern cases where the `wait();` method triggers an output instantly because no time passes, and thus the conditionals evaluate to true. The survival of these mutants is acceptable.

The 1 member variable mutant concerns the edge between $L12 \rightarrow L11$. In terms of the specification, it can be said that the edge is replaced by the edge $L12 \rightarrow L12$ with the same action. The remaining transition in the trace $L11 \rightarrow L14$ does not change the state, as `soundOff` has already been set to `true`. The 8 additional member variable mutants that survive when not using trace extension, also concern cases where an edge is replaced by a self-looping edge.

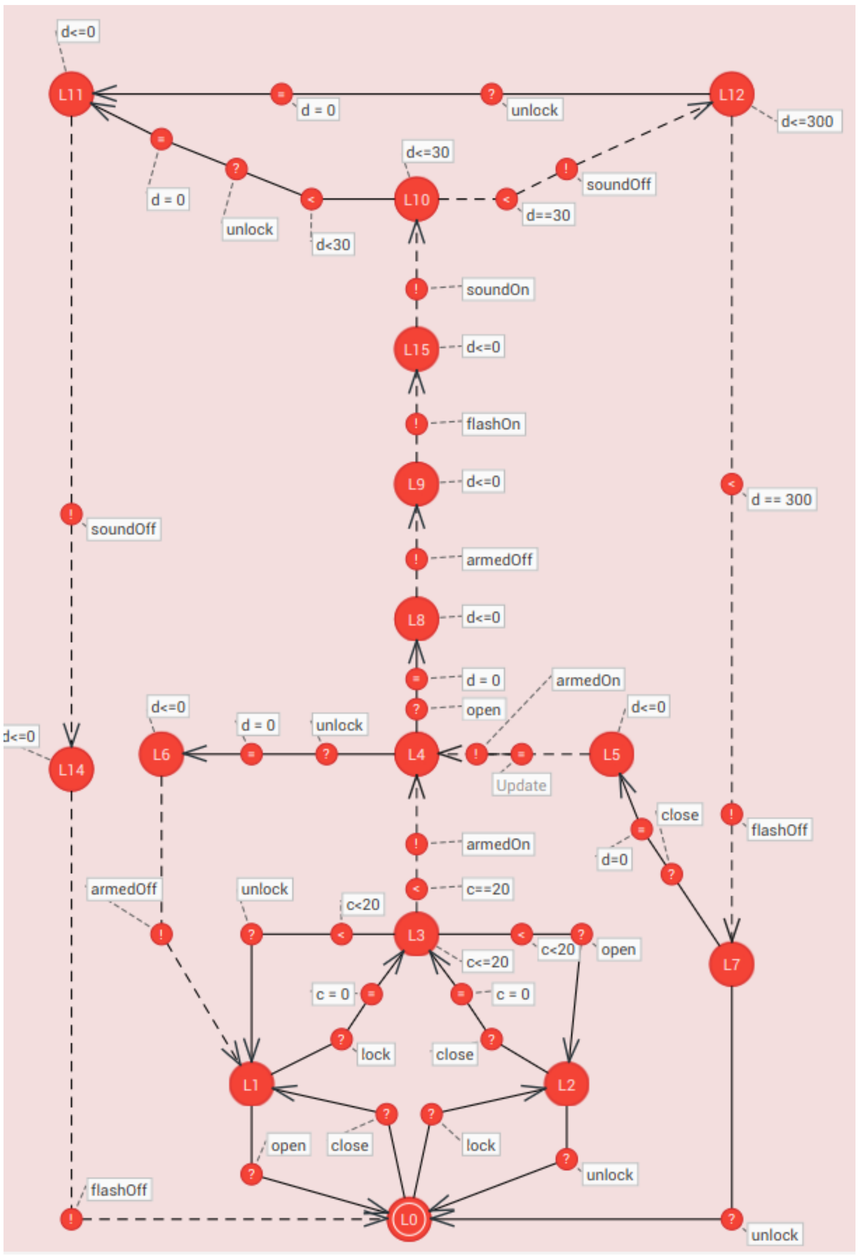

Figure 5.1: Car Alarm System Ecdar Model

It can be concluded that prefix elimination is useful for reducing the size of the test suite, and that trace extension is necessary to get a stronger test suite. Boundary Value Analysis is redundant in this case study, as the behavior the additional test cases add is already detectable by the base test suite, more specifically it is caught by the clock assertions.

	# Test cases	Mutation coverage	Surviving mutants	#
All	29	174/182	Remove conditional	7
Extend and BVA	42	174/182		
Extend and Prefix	11	174/182	Member variable	1
Extend	24	174/182		
Prefix and BVA	28	166/182	Remove conditional	7
Prefix	11	166/182		
BVA	42	166/182	Member variable	9
None	24	166/182		

Table 5.1: PIT mutation testing results for Car Alarm System

5.3 Coffee Tea Machine Case Study

The previous case study demonstrated that our approach can detect when an implementation breaks refinement rules. However, it does not demonstrate whether an implementation refines the specification. The CAS model is not very abstract and has very strict timed behavior. Therefore an additional case study will be performed on a model with more abstract timed behavior. The internal state of the system is observable as it was in the CAS case study.

CTM can be observed on Figure 5.2. Given a coin, the user has to choose either coffee or tea before 5 seconds, otherwise, a refund occurs. Based on the choice of the user, the machine either pours coffee or tea before returning to its initial state where it awaits a coin.

5.4 Results

Two test suites were created one with BVA and one without BVA. Trace Extension and Prefix Elimination was used on both test suites. The results of the mutation testing can be observed in Table 5.2. Both test suites have the same line and mutation coverage. The delays of the implementation were set to the upper bound, meaning that the outputs arrive at the latest possible time according to the specification. For example, the transition $L2 \rightarrow L5$ states that the output can arrive between in the range $6 \leq x < 10$. In this case, the upper bound is 9.

As mentioned before, "Remove Conditional" mutants replace equality checks with *false* or *true*. In this case, only when replacing conditionals with *true* survived. As an example, consider this code snippet Code example 5.1, which contains the coin method of CTM. The equality check on line 2 is changed to `if (true)`. This mutant

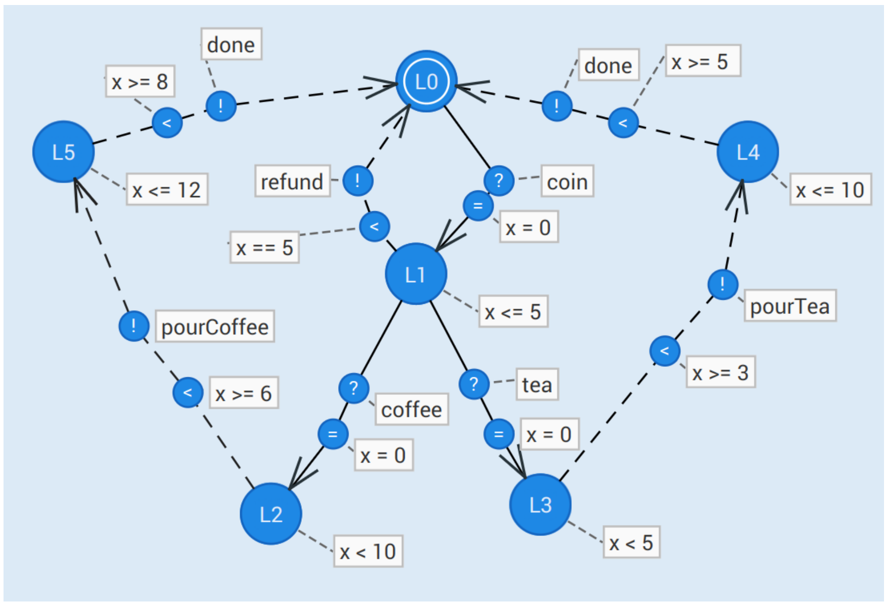

Figure 5.2: Coffee Tea Machine Ecdar Model

	# Test cases	Mutation coverage
With BVA	28	80/81
Without BVA	3	80/81

Table 5.2: Coffee Tea Machine Test Suite Mutation Coverage

survives because `coin()` is only called while being in location `L0`, if it was called while in another location the mutant would be killed.

```java
public void coin() {
     if (this.location == Loc.L0) {
         coin = true;
         this.location = Loc.L1;
     }
 }
```

Code example 5.1: Test Code for Coffee Tea Machine

Boundary Testing As mentioned previously, the implementation has equivalent timed behavior to the specification, meaning it accepts the same interval of clock values as the specification. The tests will reduce the clock range to singular values.

Table 5.3 shows the edges that were mutated, their original clock range, and the reduced clock range. The reduced clock ranges are based on the lower bound (LB) and upper bound (UB), following the principles of boundary value analysis. If the clock value triggered failing test cases it is marked as red, if not, it is marked as green. The expected outcome was that values outside LB and UB should fail, as they break the refinement relation. As expected, the mutants that used UB and LB passed the tests, while the other mutants failed.

	Clock Range	LB - 1	LB	UP	UP + 1
$L2 \rightarrow L5$	$x > 5 \;\&\&\; x < 10$	🟥 x == 5	🟩 x == 6	🟩 x == 9	🟥 x == 10
$L3 \rightarrow L4$	$x > 2 \;\&\&\; x < 5$	🟥 x == 2	🟩 x == 3	🟩 x == 4	🟥 x == 5
$L4 \rightarrow L0$	$x > 4 \;\&\&\; x < 11$	🟥 x == 4	🟩 x == 5	🟩 x == 10	🟥 x == 11
$L5 \rightarrow L0$	$x > 7 \;\&\&\; x < 13$	🟥 x == 7	🟩 x == 8	🟩 x == 12	🟥 x == 13

Table 5.3: Coffee Tea Machine Mutations Results

5.5 TrainTravel Case Study

The intent of this case study is to demonstrate that our approach is able to consider boolean variables in a model and integrate them into the test code. TT can be observed in Figure 5.3. The intuition is that if a ticket is not purchased before getting on a train, it is necessary to hide. If the person is caught without a ticket, they will receive prison time and on arrival will be arrested and sent to prison. Otherwise, if the person is not caught or has a ticket, the destination can be reached after arrival.

The test code differs in that clock assert statements also include the boolean variables in the model. However, they are only included when entering a location. The reasoning is that boolean variables only change when traversing an edge, whereas clocks can also change their value by performing delays in a location.

5.5.1 Results

Two test suites were generated, one with and one without BVA. The results can be observed in Table 5.4. The same mutation coverage was achieved in both test suites, with one of the test suites being significantly smaller.

	# Test cases	Mutation coverage
With BVA	19	78/83
Without BVA	5	78/83

Table 5.4: Coffee Tea Machine Test Suite Mutation Coverage

Three of the mutants are Conditional Removal and occur inside the input methods. Each input method contains an if-statement, as can be observed in Code example 5.2.

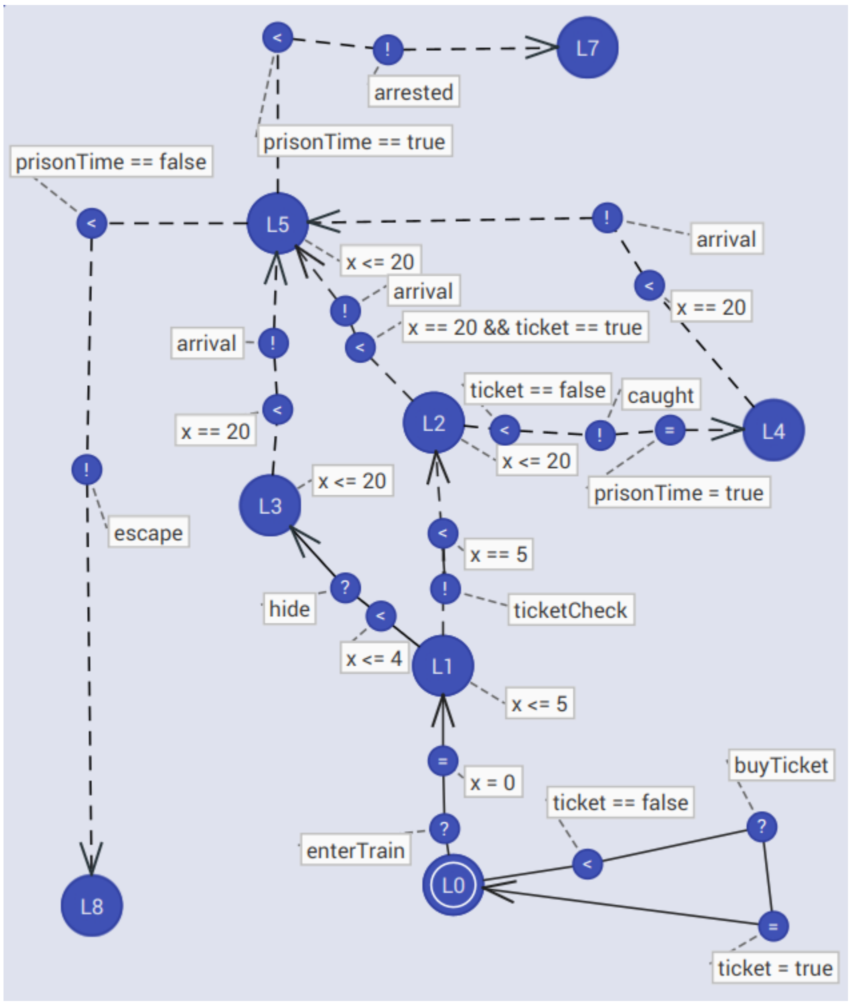

Figure 5.3: Train Travel Ecdar Model

The conditional statement is replaced by `true`. This mutant survives because the input methods are only called when they are valid.

```java
public void buyTicket() {
    if (this.location == Loc.L0) {
        ticket = true;
    }
}
```

Code example 5.2: Train Travel Implementation Snippet

The final two mutants are Member Variable mutants, specifically, they remove the location assignment from the final output calls `escape` and `arrested`. Respectively the location is set to L8 and L7, as these are the final locations in the model, the location variable is not used again and the assignment is therefore redundant.

5.6 CAS - Non-Observable Internal State

All case studies so far used implementations that had boolean representations for each input and output in the model. Intuitively it means that we observed the internal state of the system. In accordance with black-box testing, where the only observable thing about the system is outputs and the time between actions, we will attempt to follow this notion of what is observable in the test code.

5.6.1 Reduced Observable Internal State

First, every redundant boolean was removed from the assert statement, meaning that only the boolean that relates to the previous action, input or output, is asserted upon. This can be observed in Code example 5.3, where the original assert statement has been out-commented on line 7, and the new assert on line 8 only contains the boolean `locked`, as the input lock was used on line 6.

```
1 @Test
2 public void test0() {
3     assertTrue(d-c<=0 && g-c<=0 && c-d<=0 && g-d<=0 && c-g<=0 && d-g<=0);
4     wait(0);
5     assertTrue(d-c<=0 && g-c<=0 && c-d<=0 && g-d<=0 && c-g<=0 && d-g<=0);
6     lock();
7     //assertTrue(locked && !armed && !closed && !sound && !flash);
8     assertTrue(locked);
9 }
```

Code example 5.3: Test Code for Car Alarm System

A test suite was generated with and without BVA. Running mutation testing on the test suites with minimal use of boolean resulted in the same line coverage and mutation coverage. There was no difference in coverage between BVA and no BVA.

5.6.2 Non-Observable Internal State

This prompted us to take an additional step and completely remove assert statements relating to inputs. The intuition is that the goal of each individual test case is to supply the system with inputs and delays in order to reach a certain output. An input not triggering, should therefore affect our ability to reach the desired output. Output booleans are still asserted upon, however, this is considered an alternative to observing that an output occurs, and not an observation of the internal state.

In the initial round of testing, nothing was done to the implementation. The now redundant input boolean variables in the implementation created 26 surviving mutants, which led to removing the boolean variables. The input boolean variables did not affect the SUT in any way and were only accessed from the test code. First, a test suite with no BVA was used which resulted in a line coverage of 129/129 and a mutation coverage of 140/151.

8 of the surviving mutants are shared with the original test suite and implementation. The 3 new surviving mutants all relate to removing a location assignment. In terms of the CAS model, it corresponds to removing the following edges: $L3 \to L2$, $L1 \to L0$, $L3 \to L1$. The relevant transitions of the traces containing these edges can be observed below. The three traces have one thing in common: the first discrete transition is immediately followed by a transition that returns it to its previous state.

$$\begin{gathered}
(L3, c == 0) \xrightarrow{0} (L3, c == 0),\\
(L3, c == 0) \xrightarrow{open?} (L2, c == 0),\\
(L2, c == 0) \xrightarrow{close?} (L3, c == 0),\\
(L3, c == 0) \xrightarrow{20} (L3, c == 20),\\
(L3, c == 20) \xrightarrow{armedOn!} (L4, c == 20)
\end{gathered}$$

$$\begin{gathered}
(L3, c == 0) \xrightarrow{0} (L3, c == 0),\\
(L3, c == 0) \xrightarrow{unlock?} (L1, c == 0),\\
(L1, c == 0) \xrightarrow{lock?} (L3, c == 0),\\
(L3, c == 0) \xrightarrow{20} (L3, c == 20),\\
(L3, c == 20) \xrightarrow{armedOn!} (L4, c == 20)
\end{gathered}$$

$$\begin{gathered}
(L1, c == 0) \xrightarrow{open?} (L0, c == 0),\\
(L0, c == 0) \xrightarrow{close?} (L1, c == 0),\\
(L1, c == 0) \xrightarrow{lock?} (L3, c == 0),\\
(L3, c == 0) \xrightarrow{20} (L3, c == 20),\\
(L3, c == 20) \xrightarrow{armedOn!} (L4, c == 20)
\end{gathered}$$

The reason that these mutants survive is that when the edges are used in a trace, they do not affect the system in a meaningful way. As the input they use is canceled out: *open* is followed by *closed* and *unlock* is followed by *lock*. The middleman is removed and the car stays locked and closed.

A test suite using BVA was then used, which caught two of the *new* mutants. This resulted in a mutation coverage of 142/151. The one mutant that survived was the one that relates to the edge $L1 \rightarrow L0$. The reason is that no delay will have an effect on the behavior of the system in those locations. This mutant surviving is acceptable.

Only using asserts on outputs means that the approach is much more valid. The constraint that stipulated that the internal state of the system should be observable that was used in the case studies was somewhat unrealistic, and would in many cases not be applicable to a system. Outputs being observable is a far less demanding assumption on a system. Additionally, by not having access to the internal state of the system BVA has shown its usefulness by improving the mutation coverage.

6. Discussion

This chapter will touch upon various considerations and trade-offs of our approach. This includes the difference between real-time and symbolic time representations. Additionally, the discussion delves into the use of BVA, the challenges of deriving test cases from an input-enabled specification, and the soundness and portability of our approach.

6.1 Implementing the SUT

The implementations that were used for the case studies were tailored to their respective specifications. Going as far as mimicking their locations by using an enum, and transitions by changing the location and the boolean variables that are associated with inputs and outputs. When testing a real implementation this would most likely not be the case.

The implementation mimicking the model is not important for our approach to succeed, it is simply easier to use this method to ensure that the implementation does implement the behavior as the model specifies. The crux of our approach is that there is an observable "state", that changes when receiving inputs or sending outputs.

The three case studies all used a very *strict* state representation, with both inputs and outputs represented by a boolean. Additionally, each assert statement considered the current value of each boolean variable. This restricted how the specifications could be modeled and were tedious to write test code for.

In section 5.6 we tried to reduce the rigidity of the state representations by only checking the boolean associated with the last action that was taken. Mutation testing was then used on the resulting test suite, which saw no decrease in line coverage or mutation coverage. An additional step was taken where assert statements were only used after an output. A few more mutants survived, however, they do not discredit the approach. Showing that our approach will function on systems where there is no observable internal state.

6.2 Portability

One of the draw-points of Yggdrasil is that it is backend agnostic. Almost everything is customizable and is up to the user. This should mean that it can be used in any language that has testing functionalities comparable to JUnit. The only criterion other than having a way to assert expressions, is having a state representation.

The case studies featured implementations that had clock representations. The clock representations could in many cases be moved into the test code as variables.

6.3 Real-time vs Symbolic time

Both case studies were performed on implementations that used symbolic time. This was done by using integers as clock representations, and incrementing them inside the `wait()` method in order to simulate time passing. A similar approach could be used for a real-time version. The criteria are that there should be a *wait()* method, which given an integer input will either return when that corresponding time has passed, or when a *transition* occurs to a new "state" in the implementation.

Currently, only natural numbers are considered as any measured real numbered time is rounded to its closest natural number. In order to accommodate real values, a clock representation's data type should be *double* instead of *integer*.

A possible concern is that the test cases are executed in a sequence of instructions. While assert statements are performed in the test code, the implementation is at a standstill. This prompted us to measure the time it takes to perform the assert statements. The measurement was in nanoseconds and ranged between 600-1000 nanoseconds. Intuitively this means that it would take hundreds of thousands of assert statements in order to have spent 1 second outside the implementation. This might affect the assert statements for clocks, but it can be addressed by adding a small error-margin, for example it could be allowed to pass the upper bound of a clock assert by 0.02 seconds.

6.4 Boundary Value Analysis

A test suite using BVA was produced for each case study. For each case study performed on SUTs with an observable internal state, it was found that it did not increase the mutation coverage and increased the number of test cases by a considerable amount. However, BVA showed its usefulness when testing a system where the internal state was not observable in section 5.6.

Additionally, in contrast to the *normal* test cases, the BVA test cases actively attempt illegal behavior. This applies to illegal delays before applying inputs. The intuition is that actively trying to perform illegal behavior, will increase the chance of triggering refinement-breaking behavior in the SUT.

6.5 Deriving test cases from an Input-Enabled Specification

Using an input-enabled specification to derive test cases was quickly abandoned in the earlier part of this project. Our initial misgivings were due to duplicate code being created from entering and exiting locations. However, test code inside locations was abandoned in favor of only specifying test code in edges. An input-enabled specification would result in test cases that will often use inputs that should have no effect on the system. In the case that it does have an effect on the system, and it triggers a transition to a new location, then that would break refinement.

There are however trade-offs. Since we use the structural coverage criteria of covering all edges, it would result in a big increase in the number of test cases. If the tests were performed in real-time it would have a huge impact on the run-time. It would also reduce the readability of the test code.

6.6 Soundness

Refinement was used as the conformance relation for this project. The approach was specifically designed to account for the three refinement rules. The soundness criteria are that our test suite correctly identifies implementations that break refinement and does not produce false negatives or false positives.

The soundness of our approach was put to the test during the various case studies. The case studies helped identify shortcomings, which were addressed to improve upon the approach. A few mutants survived during the various case studies, however, they were the result of either redundant actions in the specification, or that the mutants did not actually break refinement. We conclude that our approach is sound.

7. Conclusion

This project set out to implement the test case generation functionality used in UPPAAL Yggdrasil and extend it to consider real-time systems. The Yggdrasil-like functionality was implemented in Ecdar and used to generate a test suite from the model, to show conformance between the SUT and the specification. The conformance relation is refinement, thus the test suite was generated with the intent of detecting refinement violations in the SUT.

The main contribution of the project is a methodology to test for refinement between a real-time system and a specification through the use of test code. This has to our knowledge not been explored before.

Three case studies were performed, where PIT mutation testing was used to ascertain the test suites' capabilities to check for refinement. The test suites adequately caught refinement violations for both concrete and abstract models. Three case studies were done under the assumption that the internal state of the systems was observable. Under this assumption, BVA did not provide additional mutation coverage. However, an experiment was performed that relaxed this assumption by only making outputs observable. BVA was able to detect violations that otherwise went undetected.

All case studies achieved good mutation coverage, which shows the test suites' ability to consistently detect refinement violations. This demonstrates our solution's potential to be used as a reliable, robust, and sound testing approach for real-time systems.

8. Future Work

Our project focuses on model-based testing of real-time systems. We implemented a test suite generation, that can show if an implementation refines a specification. There is still a good amount of future work to be done in this avenue.

Testing on larger models
An important next step is to test our approach on larger models, as we have only tested on small models so far. When testing large models, the time factor becomes a real challenge, as the state space increases really fast. Thus if we want to test these models, it would be necessary to develop techniques to tackle the large state space. If this is achieved, we would gain a better insight into the approach.

Testing compositional models
It would also be interesting to look into testing composite models. In the real world, systems are made up of multiple components that can send data from and to each other. This can also be modeled, by using synchronous models. A state in such a system would need to keep track of all the different locations and variable valuations.

Switching from symbolic time to real-time
Currently we use symbolic time represent the passing of time. In the real world, systems operate under real-time. It would be advantageous to gain a more realistic and comprehensive view evaluations to test real-time system. Thus, to test real-time systems, it is necessary to extend the solution to handle real-time. This introduces some challenges: How to handle time constraints, and how to keep track if inputs and outputs occur at acceptable times.

Adding an inconclusive verdict
As mentioned in chapter 3, our test suite can have two types of verdicts: *Passed* and *failed*. However, in most testing approaches, there is a third verdict: *Inconclusive*. This verdict is given when a decision cannot be made [18]. In our approach, the verdict could be inconclusive when the output produced by the system is valid behavior according to the conformance relation, but is not the specific path we are testing. Inconclusive verdict comes into play when the model is non-deterministic.

Bibliography

[1] Kenneth Yrke Jørgensen et al. *TEST CASES (YGGDRASIL)*. On-line: `https://docs.uppaal.org/gui-reference/yggdrasil/`. Accessed: 14-10-2022.

[2] A.D. Ames, A. Abate, and S. Sastry. “Sufficient Conditions for the Existence of Zeno Behavior”. In: *Proceedings of the 44th IEEE Conference on Decision and Control*. 2005, pp. 696–701. DOI: `10.1109/CDC.2005.1582237`.

[3] Gerd Behrmann et al. “UPPAAL-Tiga: Time for Playing Games!” In: *Computer Aided Verification*. Ed. by Werner Damm and Holger Hermanns. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 121–125. ISBN: 978-3-540-73368-3.

[4] Machiel Van der Bijl, Arend Rensink, and Jan Tretmans. “Compositional testing with IOCO”. In: vol. 2931. Oct. 2003, pp. 86–100. ISBN: 978-3-540-20894-5. DOI: `10.1007/978-3-540-24617-6_7`.

[5] Alexandre David et al. “Timed I/O automata: A complete specification theory for real-time systems”. In: Jan. 2010, pp. 91–100. DOI: `10.1145/1755952.1755967`.

[6] Tobias R. Gundersen et al. “Effortless Fault Localisation: Conformance Testing of Real-Time Systems in Ecdar”. English. In: vol. 277. 9th Symposium on Games, Automata, Logics and Formal Verification, (GandALF’18 ; Conference date: 26-09-2018 Through 28-09-2018. Open Publishing Association, Sept. 2018, pp. 147–160. DOI: `10.4204/EPTCS.277.11`.

[7] Guru99. *Mutation Testing in Software Testing: Mutant Score & Analysis Example*. On-line: `https://www.guru99.com/mutation-testing.html`. Accessed: 10/05/2023.

[8] Anders Hessel and Paul Pettersson. “A Global Algorithm for Model-Based Test Suite Generation”. In: *Electronic Notes in Theoretical Computer Science* 190.2 (2007). Proceedings of the Third Workshop on Model Based Testing (MBT 2007), pp. 47–59. ISSN: 1571-0661. DOI: `https://doi.org/10.1016/j.entcs.2007.08.005`. URL: `https://www.sciencedirect.com/science/article/pii/S1571066107005403`.

[9] Anders Hessel and Paul Pettersson. “Cover - A Real-Time Test Case Generation Tool”. In: 2007.

[10] Moez Krichen and Stavros Tripakis. “Conformance testing for real-time systems”. In: *Formal Methods in System Design* 34 (Jan. 2004), pp. 238–304. DOI: `10.1007/s10703-009-0065-1`.

[11] Kim G. Larsen, Marius Mikucionis, and Brian Nielsen. “Online Testing of Real-time Systems Using Uppaal”. In: *Formal Approaches to Software Testing*. Ed. by Jens Grabowski and Brian Nielsen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 79–94. ISBN: 978-3-540-31848-4.

[12] Srinivas Nidhra and Jagruthi Dondeti. “Black box and white box testing techniques-a literature review”. In: *International Journal of Embedded Systems and Applications (IJESA)* 2.2 (2012), pp. 29–50.

[13] Pitest. *mutators*. On-line: `https://pitest.org/quickstart/mutators/`. Accessed: 06/04/2023.

[14] Ecdar team. *Environment for Compositional Design and Analysis of Real Time Systems*. On-line: `https://www.ecdar.net/`. Accessed: 30-12-2022.

[15] JUnit Team. *JUnit 5*. On-line: `https://junit.org`. Accessed: 28/03/2023.

[16] Jan Tretmans. “Model Based Testing with Labelled Transition Systems”. In: *Formal Methods and Testing*. 2008.

[17] UPPAAL. *UPPAAL Webpage*. On-line: `https://uppaal.org/`. Accessed: 01/06/2023.

[18] Mark Utting, Alexander Pretschner, and Bruno Legeard. “A taxonomy of model-based testing approaches”. In: *Software testing, verification and reliability* 22.5 (2012), pp. 297–312.