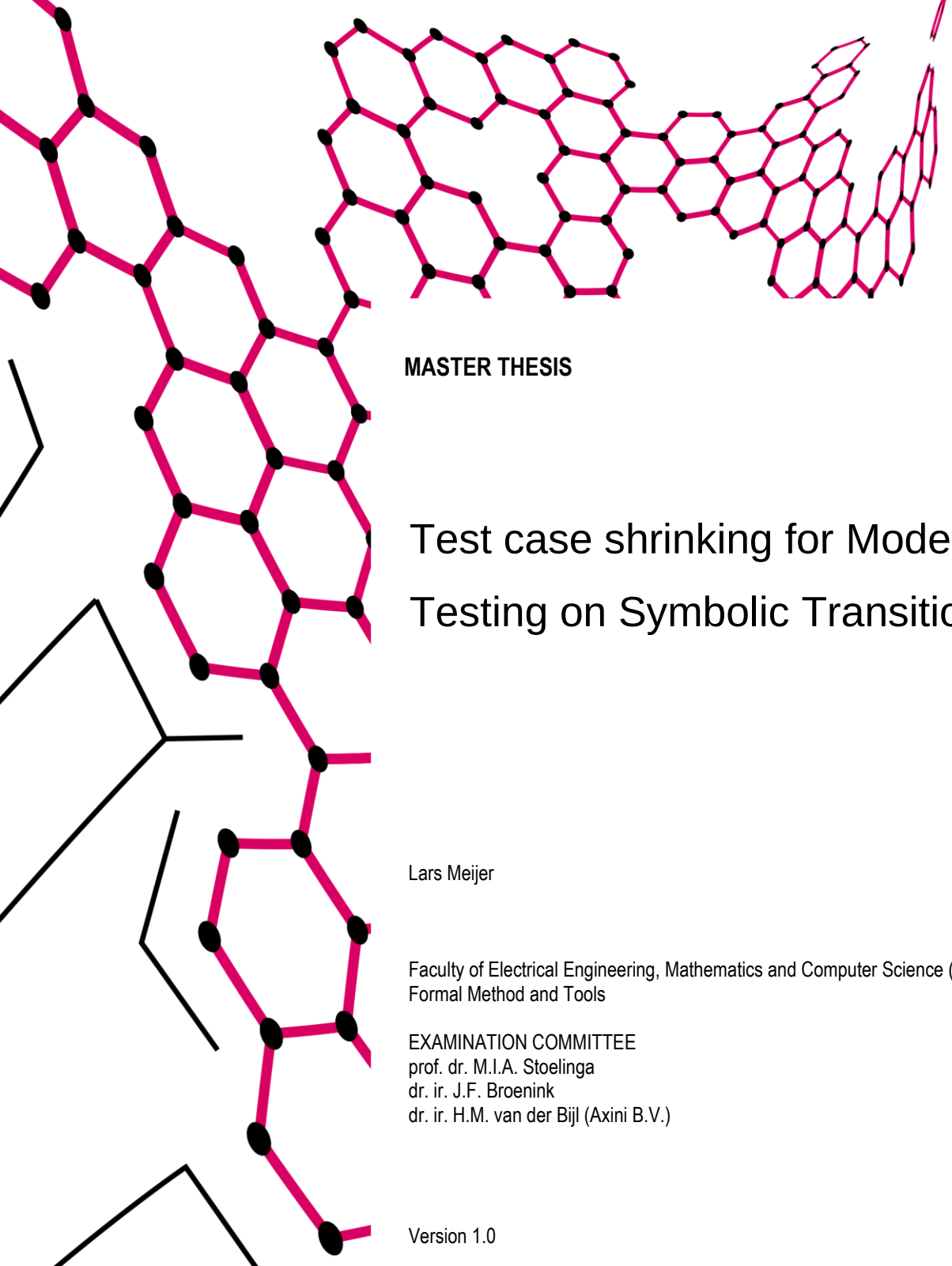

MASTER THESIS

Test case shrinking for Model Based Testing on Symbolic Transition Systems

Lars Meijer

Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)
Formal Method and Tools

EXAMINATION COMMITTEE
prof. dr. M.I.A. Stoelinga
dr. ir. J.F. Broenink
dr. ir. H.M. van der Bijl (Axini B.V.)

Version 1.0

18-02-2021

UNIVERSITY OF TWENTE.

ABSTRACT

Test case shrinking is the process of reducing the size of failing test cases to make them easier to analyse and debug. It is not hard to imagine why test case shrinking can be important to help the debugging process. To illustrate this, Andreas Zeller used an example of a web page that made a browser crash. The page that caused the crash was 896 lines long. How can one determine the underlying problem that caused the browser to crash?

To solve this issue, Zeller and Hildebrandt introduced an algorithm that can reduce the size of inputs such as this web page [1]. This algorithm is called Delta Debugging minimisation or ddmin for short. The result of applying ddmin to the web page was quite dramatic, the page could be reduced to a single line which still caused the web browser to crash. Having a single line to analyse a single line is obviously preferable to analysing all 896 lines.

This thesis sets out to evaluate the effectiveness of shrinking algorithms such as Delta Debugging minimisation on failing test cases derived from Symbolic Transition Systems (STSs). Koopman et al. introduced three shrinking algorithms and applied these to test cases derived from Extended State Machine (ESM) [2]. A formalism based on Finite State Machines, but with the added notion of data. In this thesis, these three shrinking algorithms were first applied to test cases derived from STSs. These algorithms are: The Element Elimination Shrinking algorithm, the Binary Elimination Shrinking Algorithm and the Cycle Shrinking algorithm. The other two algorithms have not yet been used on test cases derived from state based models such as ESMs and STSs. They are the Location Cycle Shrinking algorithm, based on the Cycle Shrinking algorithm by Koopman et al. and the Delta Debugging shrinking algorithm based on the ddmin algorithm.

These five algorithms were tested in two different experiments. The first experiment is based on the experiment by Koopman et al. [2] on a simple vending machine. The second experiment tested the effectiveness of the shrinking algorithms on test cases for a real-world system called PPR-AAP. This system is used by ProRail to manage part of the Dutch rail network.

The experiments showed that it is indeed possible to shrink test cases derived from STSs. They also showed that the algorithms performed differently on the two different systems. On the vending machine the Cycle Shrinking algorithm provides a reasonable amount of shrinking while using very few interactions with the System Under Test (SUT). On PPR-AAP, this was generally the case for the Location Cycle Shrinking algorithm instead of the normal Cycle Shrinking algorithm.

The best way to shrink test cases derived from STSs depends on the system and the type of bug. In general the Cycle Shrinking algorithm or Location Cycle shrinking algorithm is the best algorithm to start the shrinking process. These algorithms can be followed up by the Delta Debugging shrinking algorithm to get the shortest traces in a relatively efficient manner. In some cases, namely those were a cycle in the execution is the cause of a bug (for example the case where a bug only occurs after a certain piece of code is visited seven times), the Delta Debugging shrinking algorithm it the best algorithm to shrink test cases.

ACKNOWLEDGEMENTS

When I started thinking about the possibilities for a final project back in 2019, no one knew how 2020 would turn out. For me personally, it has indeed been a challenging year. I would like to extend my gratitude to everyone who has in any way contributed to this project. The result would not be what it is without them.

First, I would like to thank Mariëlle Stoelinga for her academic guidance in the subject matter, all the discussions we had during our meetings and her helpful feedback. I would also like to thank Jan Broenink for his role as secondary supervisor.

Next, I would like to thank Machiel van der Bijl. He helped me from the start by sharing his ideas, giving me feedback on my project and helping me write this thesis.

I would also like to thank my other colleagues at Axini. Their company, discussions and feedback on my work and presentations has been very helpful. In particular, I want to thank Peter Verkade for his help in the technical aspects of this project.

I want to thank my parents for their tremendous support during this whole project and their practical, emotional and moral support throughout my academic years.

Finally I want to thank my sister, brother, other family and my friends who provided me with much needed comfort, fun and entertainment during the course of this project.

CONTENTS

Acronyms

AML Axini Modeling Language

AMP Axini Modeling Platform

ESM Extended State Machine, *see glossary:* Extended State Machine

FSM Finite State Machine

LTS Labelled Transition System, *see glossary:* LTS

MBT Model-Based Testing, *see glossary:* MBT

STS Symbolic Transition System, *see glossary:* STS

SUT System Under Test

Glossary

Extended State Machine An Extended State Machine (ESM) is a modified Finite State Machine (FSM) that has a notion of data, in the form of variables. Additionally, an ESM has one input and zero or more outputs on each transition.

LTS A Labelled Transition System (LTS) is a model used in model based testing. A labelled transition system consists of states and transitions between them. Each transition has a Label. See also Section 2.1.1.

MBT Model Based Testing is an innovative and formal way to test software. The specifications of a System Under Test (SUT) are described in a model. This model can be used to generate test cases, which can verify if the SUT was correctly implemented.

PPR-AAP A system used by ProRail, part of the software package that is used to manage the Dutch rail network. See also Section 5.2.

STS A Symbolic Transition (STS) is a model used in model based testing. STS introduce a notion of data and data depended control flow on top an STS. See also Section 2.1.2.

1 INTRODUCTION

Software plays an important role in our lives. Nowadays software can be found everywhere. Software is in televisions, phones and even smart light bulbs. More vital systems, such as medical equipment and control systems for railways also use a lot of software. Companies spend a lot of time, effort and money making sure these systems work as intended. The most prominent way to do this is by testing the software. Over the years, many forms of testing have been proposed and used in industry. These include testing methods such as unit testing, regression testing and integration testing.

Model-Based Testing (MBT) is an innovative way to test software. In Model-based testing specifications or requirements of a software system are formally recorded in a model. A model is usually modelled in a special modelling language such as Promella [3] or as a mathematical structure such as a Labelled Transition System [4]. Based on a model, test cases are generated and executed against the System Under Test (SUT). If the SUT does not adhere to the model, a test will fail. A failing test will produce a list of steps to reproduce this failure, this is called a *trace*. Chapter 2 gives a more detailed overview of the theory behind model-based testing.

One of the greatest strengths of MBT is the automatic generation of test cases from a model. Because test-generation is automatic, many long and complicated test cases can be generated. When a test case fails however, this also is a drawback: The test case needs to be analysed to find the underlying reason why the test failed, which is made harder by the length of the test case.

Analysing long, failing, test cases is not a new problem, nor is it a problem exclusive to MBT. A method that can be used to tackle this problem, is generating a shorter test-case, based on a failing test case that has already been found. This is called test case shrinking.

One well-known method to shrink test cases is called Delta debugging minimisation (ddmin) [1], an extension to the original delta debugging algorithm [5]. This algorithm turned out to effective at minimising test cases in several case studies.

This thesis focuses on shrinking traces of failing tests in MBT. Specifically MBT using Symbolic Transition Systems (STSs), a formalism used to model systems. An overview of the theory behind STS can be found in Section 2.1.2.

1.1 Method

The thesis is centred around the research question:

“What method can be used to shrink traces of failing tests in test cases derived from Symbolic Transition Systems?”.

The sub questions that help answer this research question are defined in Chapter 3. To answer these questions, five algorithms were evaluated by shrinking test cases derived from STSs.

Three algorithms were first introduced in earlier work by Koopman et al. [2]. Two algorithms have not previously been used to shrink test cases derived from state-based models. The three algorithms by Koopman et al. are:

- Element Elimination Shrinking algorithm: An algorithm that tries to remove single steps from a failing test case at the time.
- Binary Elimination Shrinking Algorithm: An algorithm that starts by trying to remove half of the steps of a failing test case at the time. If this is unsuccessful the algorithm tries to remove a quarter of the steps of a failing test case, then one eight of the steps and so on.
- Cycle Shrinking algorithm: This algorithm tries to remove cycles from a failing test case. A cycle is a part of a test case where the start state and end state are identical.

The other two algorithms are:

- Delta Debugging Shrinking algorithm: An algorithm based on the ddmin algorithm, adapted to Shrink test cases derived from STSs.
- Location Cycle Shrinking algorithm: An adaptation of the Cycle Shrinking algorithm that uses a less strict definition for detecting cycles.

These five algorithms were evaluated in two experiments. The first is a replication of the experiment originally described by Koopman et al. on a simple vendinging machine. In their work, Koopman et al. noted that the algorithms have not yet been tested on real-world systems. The second experiment was done on a system called PPR-AAP. This is a system used by ProRail, a Dutch government agency responsible for maintenance and traffic control of the Dutch rail network, to manage parts of the network.

In each experiment several bugs were introduced into the system, then failing test cases were generated and shrunk using the shrinking algorithms as well as combinations of several of these algorithms. To evaluate each algorithm, four key metrics were measured: Shrinking percentage, number of interactions with the SUT, number of test cases executed during the shrinking process and the time it took to shrink a test case. Chapter 5 describes the methods used in this thesis in more detail.

1.2 Results

The experiments showed that it is indeed possible to apply shrinking algorithms to failing test cases derived from STSs. Interestingly, the experiments showed that not all algorithms worked equally well on the different systems. Shrinking using the Element Elimination shrinking algorithm took too long to be considered feasible to use in the second experiment.

In the first experiment, the Cycle shrinking algorithm used the fewest interactions with the SUT, while shrinking the original test by an average of 73%. By following this up with either the Delta Debugging shrinking algorithm or the Binary Elimination shrinking algorithm, the shrinking percentage can be increased to 82%.

In the second experiment, the Cycle Shrinking algorithm performed significantly worse, only shrinking a failing test case by 25% while using relatively many interactions with the SUT. In this experiment the Location Cycle Shrinking algorithm proved to be quite efficient, in most cases it used the fewest interactions with the SUT. The Location Cycle shrinking algorithm shrunk traces on average by 70%. By following this up with delta debugging, the shrinking percentage can be increased to about 80%.

In cases where a cycle is the cause of a bug, for example when a bug only triggers after a certain

action has been executed n times, the Location Cycle Shrinking algorithm did not perform well. In cases like these, the Delta Debugging shrinking algorithm performs the best.

The full results and discussion can be found in Chapter 7.

1.3 Axini

This research was performed at and in collaboration with Axini. Axini is a spin-off company of the University of Twente. They are located in Amsterdam and specialise in model-based testing and model-driven engineering. Since 2007 Axini has been working on bringing so-called 'formal methods' to industry. First, by providing expertise and tools for model-based testing to companies. Later they also started providing tools and expertise for a completely model-driven work style.[1,2]

1.4 Reading guide

The next chapter will discuss the theoretical background of model-based testing. Chapter 3 will discuss the objective and motivation for this research and contains the research goal and research questions. In chapter 4 related work will be discussed. Chapter 5 describes the research methodology and how the results are verified. Chapter 6 describes how the shrinking algorithms were implemented and integrated into the Axini Modeling Platform (AMP). Chapter 7 will show the results of the experiment and give answers to the research questions. The thesis ends with a conclusion and a chapter on possible future work.

[1] https://www.axini.com/en/students/
[2] https://www.axini.com/en/about/

2 BACKGROUND

Testing is often seen as an essential part of the software engineering process. Testing can be described as observing the execution of software to see if it works as intended and to uncover faults in the software [6]. It is seen as tedious by developers, and requires significant effort. Testing takes up a significant part of the total development time [7].

Testing can be done completely by hand. A tester can try out the system and manually confirm that everything functions as expected. Testing this way is extremely time consuming and costly.

For this reason parts of the testing process are often automated. A particularly well-known method for testing is unit testing. In unit testing, individual components ('units') of software are tested. Each unit test validates a single component. Unit tests are often developed concurrently with the System Under Test (SUT). While tests still need to be created and maintained by a developer or tester, they are automatically executed by a unit testing framework [8].

2.1 Model-based testing

Model-Based Testing (MBT) is an innovative and formal way to test software. The specifications of a SUT are described in a model. A model that describes a SUT can be expressed in a modeling language such as Promela [3]. A model can also be expressed as a mathematical structure, such as a Finite State Machine (FSM), Labelled Transition System (LTS) [9] or Symbolic Transition System (STS) [4].

In MBT, tests can be generated automatically from a model. This is unlike unit testing, where each test needs to be written and maintained by a software developer or tester. The generated test cases will then be executed, for each test case it is automatically determined if the test passes or fails.

This chapter will focus on Model-Based Testing using LTSs and ioco theory first introduced in by Tretmans in [9]. It will also discuss STSs, an extension to LTSs, introduced in [4].

2.1.1 Labelled Transition System

A Labelled Transition System (LTS) [9, 10] is a model used in computer science. In MBT it is used to model the requirements of a system. The STS is a preliminary of the formalism used within the Axini Modeling Platform (AMP), the Symbolic Transition System.

Every LTS consists of a set of states and transitions between them. Definition 2.1.1 gives a formal definition of an LTS. Each transition has a label, this label can be either an input label (a stimuli) or an output label (a response). Usually, input labels are denoted with a ?-mark, while outputs are denoted with an !-mark. An LTS can also have unobservable actions. These actions cannot be observed from the outside world and are labelled with τ.

Definition 2.1.1 (Labelled Transition System)**.** A Labelled Transition System (LTS) is a four-tuple $A = \langle S, L, T, s_0 \rangle$:

- S is a set of states.
- L is a countable set of labels, representing actions. Where $L = L_I \cup L_O$ and $L_I \cap L_O = \emptyset$. L_I are the input labels and L_O are the output labels.
- $T \subseteq S \times (L \cup \{\tau\}) \times S$ is the transition relation, with $\tau \notin L$.
- $s_0 \in S$ is the initial state.

Figure 2.1 shows a simple coffee machine as an LTS. This machine accepts 10 or 20 cent pieces. If 40 cents are inserted and the user presses a button, then the machine should dispense coffee.

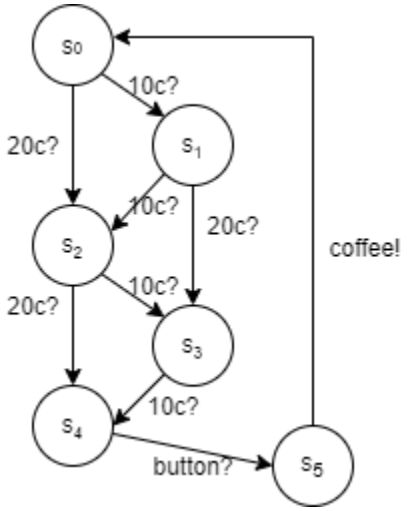

Figure 2.1: A simple labelled transition system

A single transition is a triple $\langle s, l, s' \rangle \in T$, and is also written as $s \xrightarrow{l} s'$. Multiple transitions can be composed: $s \xrightarrow{l_1} s' \xrightarrow{l_2} s''$ can be written as $s \xrightarrow{l_1 \cdot l_2} s''$. If from s a sequence of actions $s \xrightarrow{a \cdot \tau \cdot b \cdot c} s'$ can be preformed, this can be written as $s \xRightarrow{a \cdot b \cdot c} s'$. It is said that s can preform the *trace* $a \cdot b \cdot c \in L^*$. If a state p can perform a trace σ, this is written as $p \xRightarrow{\sigma}$. If this state cannot preform the trace σ, this is written as $p \xRightarrow{\sigma}\!\!\!\!\!\!/\;$. An STS is called input enabled if all input actions are available in every state.

The conformance relation ioco

A conformance relation is a mathematical relation between a specification and an implementation. The conformance relation describes whether an implementation is correct with respect to a specification.

The input-output conformance (ioco) relation is used as the basis for conformance testing of a SUT using an LTS. Informally, an implementation ioco conforms to a specification if for all tests generated from the specification, the output produced by the SUT is a subset of the output defined in the specification.

Before a formal definition of ioco can be given, a few more concepts need to be introduced. Quiescence is the absence of output. An outside observer looking at a quiescent system will

not see any outputs from the system. A state is called quiescent if it has no output transitions. Quiescent states can be labelled with the special δ label. The transition $s \xrightarrow{\delta} s$ can be added if s is quiescent.

Definition 2.1.2. Let s be a state of a labelled transition system.

- s is quiescent, denoted $\delta(s)$ if $\forall x \in L_O \cup \tau : s \overset{x}{\not\rightarrow}$
- $L_\delta =_{def} L \cup \{\delta\}$

The following concepts are formally defined in Definition 2.1.3, in the examples below the LTS from Figure 2.1 is used.

The set $init(p)$ contains all labels that are available in state p. The set can contain both input labels and output labels as well as the unobservable (τ) label, so $init(s_5) = \{coffee!\}$.

The set of states s **after** σ is composed of all states that are reachable from s after the trace σ. In the example s_0 **after** $20c? \cdot 20c? = \{s_4\}$. Note that if an LTS is non-deterministic, this set can contain multiple elements.

The set $out(s)$ contains all available output transitions in s, so $out(s_5) = coffee!$. The set $out(S)$ contains all available output transitions for the set of States S.

The set of all traces in an LTS is given by $traces(s)$. In the example $traces(s_0) = \{\epsilon, 10c?, 20c?, 10c? \cdot 10c?, 10c? \cdot 20c?, 10c? \cdot 10c? \cdot 10c?, ...\}$. Because there is a cycle in the LTS, this set has infinitely many elements. The suspension traces (Straces(s)) of an LTS include all traces that may include the quiescent action δ.

Definition 2.1.3. Let p be a state of a Labelled Transition System, let P be a set of states and $\sigma \in L^*$ then.

- $init(p) =_{def} \{ x \in L \cup \{\tau\} \mid p \xrightarrow{x} \}$
- $traces(p) =_{def} \{\sigma \in L^* \mid p \overset{\sigma}{\Rightarrow}\}$
- p **after** $\sigma =_{def} \{p' \mid p \overset{\sigma}{\Rightarrow}\}$
- $out(p) =_{def} \{x \in L_O \mid p \xrightarrow{x}\}$
- $out(P) =_{def} \bigcup \{out(p) \mid p \in P\}$
- $Straces(p) =_{def} \{\sigma \in L_\delta^* \mid p \overset{\sigma}{\Rightarrow}\}$

With these definitions, a formal definition of ioco can be given.

Definition 2.1.4 (Input-output conformance (ioco))**.** The relation ioco is defined as:

$$i \textbf{ ioco } s \Leftrightarrow_{def} \forall \sigma \in Straces(s) : \ out(i \textbf{ after } \sigma) \subseteq \ out(s \textbf{ after } \sigma) \tag{2.1}$$

Where

- i is the implementation.
- s is the specification.

The conformance relation describes if an implementation is correct with regards to a specification. Using a conformance relation, test cases can be generated. A *test-case* is a specification of stimuli and the expected responses. During test execution, stimuli are applied to the system and responses are observed. Execution of a test against a SUT results in a sequence of stimuli and responses, a trace. If such a trace corresponds to a trace in the model, the test is said to

pass. If the trace does not correspond to a trace is the model, the test *fails*. A set of test cases is called a test suite. A formal definition of test cases and test suites, from [10] is given in Definition 3. Figure 2.2 shows an example of a test case with $L_I = \{?but\}$ and $L_O = \{!liq, !choc\}$.

A test suite is used to asses whether a SUT ioco conforms to the model or not. Since the set of possible traces is almost always infinite, a test suite is an approximation of the conformance relation.

Definition 2.1.5. A test case as defined by [10]. The special label θ denotes the observance of quiescence.

1. A test case $t = \langle S, L, T, s_0 \rangle$ for an implementation with inputs in L_I and outputs in L_O is an LTS such that:
   - t is finite and deterministic.
   - S contains two special states, **pass** and **fail**, **pass** $\neq$ **fail**, with
     pass $:= \Sigma \{ x;$ **pass** $\mid x \in L_O \cup \{\theta\} \}$
     fail $:= \Sigma \{ x;$ **fail** $\mid x \in L_O \cup \{\theta\} \}$
   - t has no cycles except in the states **pass** and **fail**
   - for any state $s \in S$ of the test case
     either $init(s) = \{a\} \cup L_O$ for some $a \in L_I$
     or $init(s) = L_O \cup \theta$
2. A class of test cases for implementations with inputs L_I and outputs L_O is denoted as $TTS(L_O, L_I)$.
3. A test suite T is a set of test cases: $T \subseteq TTS(L_O, L_I)$.

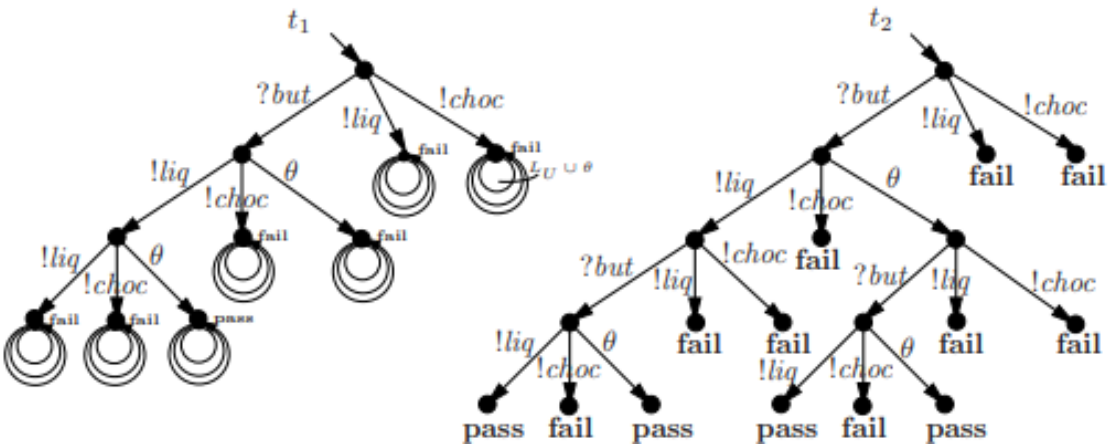

Figure 2.2: "Two test cases with $L_I = \{?but\}$ and $L_O = \{!liq, !choc\}$. Test case t_1 provides input ?but to an implementation. If this is successful t_1 expects to receive $!liq$ from the implementation followed by nothing, i.e., quiescence. Any other reaction is considered erroneous and leads to fail." from Figure 7 in [10].

Tretmans introduces a batch test generation algorithm in [9]. One on-the-fly test generation algorithm is introduced in [11]. The on-the-fly test-generation algorithm is explained below and is fairly straight forward. For a more detailed explanation and psuedocode for the test generation algorithm, see [11].

The algorithm starts with an empty trace. While this trace is shorter than a set maximum length n, first observe the implementations next output. If this output was unexpected according to the

specification, the test fails. Otherwise, add the output to the trace. Now, if, according to the specification, it is possible to do a stimuli after the current trace, apply this stimuli. Then repeat this process by observing output from the SUT again.

2.1.2 Symbolic Transition System

Complex data types that are often used in real systems can lead to a very large, or even an infinite data domain. In an LTS, every valuation of a variable, could be a new state. This makes it very hard to describe a system with large domains as an LTS. Symbolic Transition Systems (STSs) [4] are an extension to LTSs, they are used by Axini to model systems. STSs introduce a notion of data and data depended control flow on top an LTS. This results in the following definition for an STS:

Definition 2.1.6. A Symbolic Transition System (STS) is a seven-tuple $\langle L, l_0, V, \iota, I, \rightarrow \rangle$:

- L is a set of locations (called states in STSs).
- $l_0 \in L$ is the initial locations.
- V is a set of location variables. These are variables that describe the state of the model. For example the current balance (of coins) of a vending machine.
- ι, the initial valuation of the location variables.
- I is a set of interaction variables, disjoint from V. An example of an interaction variable would be the value of a coin that was inserted.
- Λ is a finite set of observable gates, in called labels in an LTS. τ is the unobservable gate. Λ_τ is written for $\lambda \cup \tau$.
- $\rightarrow$ is the transition relation. The transition $(l, \lambda, \varphi, \rho, l') \in \rightarrow$ is a transition from state l to l' and can be written as $l \xrightarrow{\lambda,\varphi,\rho} l'$. λ is a gate, such as $coin?$. φ is a transition restriction, such as $balance > 40$. If a restriction evaluates to true the transition can be followed, otherwise it cannot be followed. ρ is an update mapping. An update mapping will change the value of location variables, written as $balance := balance + 1$.

Figure 2.3 shows a vending machine similar to the LTS of the previous section as an STS. This STS can be written as:

$$\langle L, l_0, V, \iota, I, \Lambda, \rightarrow \rangle = \langle \{S_0, S_1\}, S_0, \{\textit{money}\}, \{\textit{balance} := 0\}, \{\textit{coin\_value}\}, \{\textit{coin}?, \textit{button}?, \textit{coffee}!\}, \rightarrow \rangle$$

The transitions in $\rightarrow$ are shown in Figure 2.3 on the solid arrows. The dashed arrow shows the initial state and the initial valuation of the location variables.

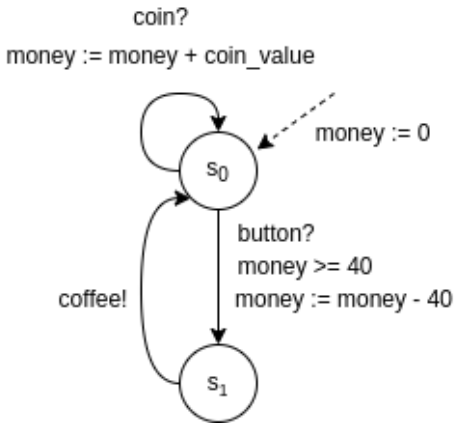

Figure 2.3: An STS of a system similar to the coffee machine in Figure 2.1 with initial state S_0 *and valuation* $balance := 0$

From state S_0, a coin with a certain value (in cents) can be inserted, this value is then added to the total amount of money. If there is more than 40 cents inserted a transition to state S_1 can be followed, which will decrease the current value of money by 40. The machine returns back to state 0 by following a transition that dispenses coffee. It is clear that this model is more concise and compact than the LTS in Figure 2.1.

3 RESEARCH OBJECTIVE AND MOTIVATION

When a test fails, the failure needs to be analysed. The software engineer or tester has to know what caused a test to fail to repair the underlying problem. In model-based testing, finding the cause of a failure can be done by analysing the trace of the failing test. It is easy to imagine that a long trace is harder to analyse than a short trace. In this chapter a motivating example will be discussed, which is used to motivate the goal of this research: *Find a way to automatically and effectively shrink traces of failing tests for systems that are modelled as Symbolic Transition Systems.*

3.1 Motivating example

Figure 3.1 shows the model of a simple vending machine as a LTS. This vending machine accepts a coin, after which a product can be selected. When the 'go' button is pressed, it will produce the selected product. Coins inserted while not in state 1 will be ejected, for readability, this is not shown in the figure. The machine will return to the state the coin was inserted in.

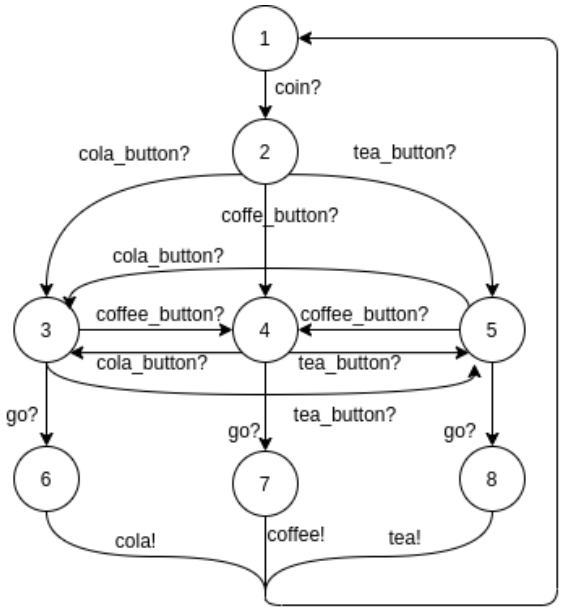

Figure 3.1: LTS representing the simple vending machine. For readability, self loops have been omitted.

The following is an example, where someone tries to debug an error by hand, by trying to remove parts of a trace. To make the example easier to follow this section uses paths, not traces. A path is a trace with states included. Consider the following path:

$$1 \xrightarrow{coin?} 2 \xrightarrow{coffee\_button?} 4 \xrightarrow{go?} 7 \xrightarrow{coffee!} 1 \xrightarrow{coin?} 2 \xrightarrow{coffee\_button?} 4 \xrightarrow{tea\_button?} 5 \xrightarrow{go?} 8 \xrightarrow{tea!}$$
$$1 \xrightarrow{coin?} 2 \xrightarrow{cola\_button?} 3 \xrightarrow{go?} 6 \xrightarrow{cola!} 1 \xrightarrow{coffee\_button?} 1 \xrightarrow{go?} 1 \xrightarrow{cola!}$$

The machine produced cola, while it should have produced nothing. It is not immediately clear what causes this error. To debug this relatively short trace by hand, one might try to reproduce this failure with less steps. With less steps, it is easier to identify the cause of the problem. All three drinks have been ordered, but the failure occurred when the last ordered product was cola. So a tester might try to reproduce this bug, with the steps of ordering cola removed, to see if this test passes. During a new test, the bug still occurs, but now the final output is tea. The new path looks like this:

$$1 \xrightarrow{coin?} 2 \xrightarrow{coffee\_button?} 4 \xrightarrow{go?} 7 \xrightarrow{coffee!} 1 \xrightarrow{coin?} 2 \xrightarrow{coffee\_button?} 4 \xrightarrow{tea\_button?} 5 \xrightarrow{go?} 8 \xrightarrow{tea!} 1 \xrightarrow{go?} 1 \xrightarrow{tea!}$$

We look at the same path again and notice that there was no coin inserted. It might be the case that the machine outputs a drink when there is no balance. To test if this is the case, we try just ordering coffee without inserting a coin. Now the machine outputs nothing, which is expected, so the test passes. Finally, we realise that, when there isn't enough balance, the machine outputs the drink that was last dispensed. To investigate if this is indeed the case, we try inserting a coin, ordering a tea and then ordering a coffee without inserting a coin. The machine outputs tea and the test fails.

It turns out the hypothesis that if there is no coin inserted, the vending machine outputs the last dispensed item, is indeed true. A shorter failing path is:

$$1 \xrightarrow{coin?} 2 \xrightarrow{coffee\_button?} 4 \xrightarrow{go?} 7 \xrightarrow{coffee!} 1 \xrightarrow{go?} 1 \xrightarrow{coffee!}$$

The example shows that even relatively short traces are not trivial to analyse. This problem gets worse when traces of failing tests get longer.

3.2 Research goal

In the Axini Modeling Platform (AMP), tests are generated on-the-fly. AMP will generate a configured number of tests. Each test continues until a certain number of steps is reached, or until the test fails. This means that traces of failing tests can be rather long. It is not unusual that a trace of a failing test is several hundred steps long.

Any failing test demonstrates that the SUT does not conform to the model. If the goal is exclusively to show that a system does not conform to the model, a long trace of a failing test case is just as good as a short trace. In practice however, we want to analyse the failing test case to find the underlying cause. Ultimately, we want to repair the issue that caused the test case to fail. This means that shorter traces are preferred.

If analysing the trace and repairing the underlying issue takes a long time, it will increase the cost of testing. The higher costs and conceived tediousness of analysing long, failing, tests might hurt the adoption of model-based testing in industry. It also makes the debugging process more expensive for companies that are currently using MBT such as Axini.

Besides making a test case easier to analyse, effectively shrinking test cases could be a step in the process of grouping together failures by root-cause and doing pattern detection on these traces [12]. Root-cause analysis and pattern detection can help to make the debug process even less time-consuming.

Given the motivating example and the problem described above, the goal of this research is defined as: *Find a way to automatically and effectively shrink traces of failing tests for systems that are modelled as Symbolic Transition Systems.*

The idea of shrinking failing tests to make them easier to analyse is not new. One earlier effort is delta debugging by Andreas Zeller [5] (explained in section 4.2.1). Delta debugging is a way to find the smallest set of changes that causes a test to fail. In another paper, Zeller introduces the delta debugging minimisation ($ddmin$) algorithm [1] (explained in section 4.2.1). This adaptation of delta debugging is a way to find the smallest failing test case, based on a larger failing test.

The paper "Model-Based Shrinking for State-Based Testing" [2] by Koopman et al., is discussed in detail in section 4.1. This paper proposes three algorithms to shrink traces of failing test cases in model-based testing using Extended State Machines (ESMs).

3.3 Approach

This research will adapt and apply the algorithms proposed in the paper by Koopman et al. to test cases derive from STSs. To do this, it is assumed that a SUT behaves deterministically. This means that when executing the same test multiple times, the result will be the same for each test.

Koopman et al. showed that their algorithms worked well on "relatively small" systems, but note that larger "real-world systems" still have to be investigated. The applicability of these algorithms to real-world systems is important for their adoption. Since the research will be carried out in collaboration with Axini, this is a good opportunity to try the adapted algorithms on such real-world systems.

Other algorithms that could be used for shrinking include the previously mentioned $ddmin$ algorithm. It is worthwhile to see if this algorithm can be used to faster, or more effectively shrink traces of failing tests. The *ddmin* algorithm is used to find a minimal failing test case based on any test case that can be divided. The *ddmin* algorithm can be applied to the list of inputs, just like the algorithms by Koopman et al. [2] are.

Finally, there are other algorithms that could be used to shrink failing test cases. A shortest part algorithm could be used to find the shortest path to a failing transition. The shortest path to a certain transition could be determined from only the model, without needing to run a lot of tests against the SUT. This could drastically decrease the time needed to shrink a trace. A combination of previously mentioned algorithms might also be the best option to shrink failing test cases.

3.4 Research questions

Based on the objective and motivation above, the following research question is defined:

What method can be used to shrink traces of failing tests in test cases derived from Symbolic Transition Systems?

The answer to this question is an algorithm that automatically shrinks traces. Possible solutions are explored in the sub-questions. Algorithms are evaluated using the size of the newly found traces, the amount of time the shrinking process takes, the number of interactions with the SUT and the number of executed test cases during the shrinking process. Further explanation of these metrics and the exact method for answering the research questions can be found in Chapter 5.

The following sub-questions are defined to explore possible solutions and to help answer the main question:

1. *"Can the results of Koopmans et al. be reproduced on Symbolic Transition Systems?"*

2. *“What is an effective way to handle models that are not input enabled for the element elimination and binary elimination algorithms?”*
3. *“How do the algorithms by Koopman et al. perform when they are algorithms applied to real-world systems?”*
4. *“Can modifications be made to these three algorithms to get shorter traces or complete the shrinking process faster?”*
5. *“How effective is the delta-debugging minimisation algorithm for shrinking traces?”*
6. *“What other techniques, such as shortest path algorithms, can be used to shrink traces and how effective are they?”*

4 RELATED WORK

This chapter discusses work related to the work presented in this thesis. The first important work discussed in this chapter are the shrinking algorithms that were introduced by Koopman et al. [2] to shrink test cases derived from ESMs. The second important work is the work on Delta Debugging and the Delta Debugging Minimisation algorithm by Zeller and his co-authors.

At the end of this chapter some applications of delta debugging, and shrinking test cases in other contexts are discussed.

4.1 Model-Based Shrinking for State-Based Testing

The paper "Model-Based Shrinking for State-Based Testing" [2] proposes several ways to shrink test cases based on traces in an Extended State Machine [13], which are used in the testing tool Gast[1]. Gast is a tool for MBT written in the programming language Clean. Gast uses Extended State Machines (ESMs) [13] to model systems. ESMs are basically finite state machines, but with the added notion of variables, similar to an STS.

First, the authors discuss a 'binary search' method. This method looks for a shorter trace of a failing test, based on the length of a trace of a failing test. If a trace length n is found, the algorithm tries to find a (completely new) trace with length of less than $n/2$. If a shorter trace is found, the algorithm tries to find a trace with a maximum length of $n/4$. If no such trace can be found, the algorithm tries to find a trace with length $3n/4$. By repeating this process, relatively small traces can be found. The biggest drawback of this approach is that, it does not scale well to find minimal traces. This is the case because the algorithm is searching for completely new traces. This means that many attempts might be needed before a trace of length $n/2$ is found, if it can be found at all.

Next, the authors look at three shrinking algorithms. These algorithms try to find smaller non-conforming traces, by removing inputs from the sequence of inputs of a trace that has already been found. Testing conformance is done on-the-fly, by testing a predetermined number of transitions or until non-conformance is detected. In this paper, the SUT that was used is input enabled, thus any sequence of inputs is always valid. The model used for the experiment in this paper is also input enabled.

While executing a test case, Gast maintains a set of states that are reachable in the model after the current trace. This is the so called **after** set, as explained in Definition 2.1.1. When this set is empty Gast has found non-conformance and the test fails. If the maximum number of transitions is reached, without finding non-conformance, the test passes. Example 1 illustrate how this would work using Gast.

[1] `http://www.cs.ru.nl/~pieter/gentest/gentest.html`

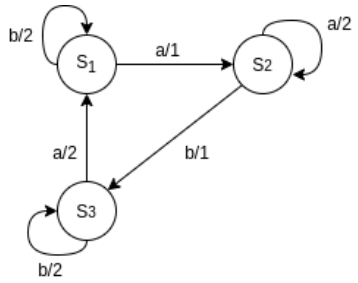

Figure 4.1: A simple input enabled FSM with input labels a and b and initial state S_1. The

Example 1 conformance testing and shrinking using Gast.

Figure 4.1 shows an (input enabled) FSM where S_1 is the initial state. If Gast first stimulates the SUT with an a, and the SUT responds with 1, this is conformant (since state s_2 is reachable). If the SUT had responded with 2, this would not be conformant, since from state s_1, there are no states reachable with a transition $a/2$. If Gast found a non-conformant trace $b/2 \cdot b/2 \cdot a/2$. The list of inputs is b, b, a. If the first input is removed, the new list of inputs would be b, a and these inputs can now be reapplied to the SUT. If a is applied to the SUT, the SUT responds with 1, this is conformant since state S_2 is reachable with an a/1 transition. Then b can be applied to the SUT, and if the system responds with 1, this is conformant, since state s_3 can be reached via this transition.

The first algorithm the authors describe tries to remove a single input at the time from the list of inputs. This algorithm is called Element Elimination. The algorithm will eliminate the first input, then test the new list of inputs against the SUT. If the new list of inputs also produces non-conformance when tested, the eliminated input is removed from the list of inputs permanently, otherwise the step is added back to the list of inputs. The algorithm moves on to the next input in the list and repeats the process.

Since the model and SUT are input enabled eliminating inputs from the list of inputs will never cause a problem. In systems where the model is not input enabled, testing can be truncated when an input is supplied that is not defined in the model. At this point, the shrink was unsuccessful, so the test can be treated like a passed test case for the purpose of shrinking.

The next introduced algorithm tries to remove larger parts of the sequence of inputs and is called binary elimination. Table 4.1 shows the process using binary elimination for the trace $a \cdot b \cdot c \cdot d \cdot e$. Binary elimination uses the of inputs, just like element elimination, but the last input is excluded, since that is the input that caused the test to fail, in Table 4.1 this is e.

First, the whole remaining list of inputs is removed. this leaves just the last input, this is step 1 in the table. This input is tested, if the test fails, shrinking is done since non-conformance can be shown with a single input. Otherwise, the removed inputs are added back, and only the first half of the inputs is removed, this is step 2 in the table. If testing this leads to non-conformance, this half is permanently removed and the algorithm will move on to the next half that has not yet been removed. If it does not lead to non-conformance, the half is split into two smaller halves, and the algorithm is repeated for the first half. This is seen in step 2 and 3 in Table 4.1, where the half [a,b] is split in to [a] and [b] and [a] is removed. If a half can no longer be split (when a half is only one input long) the algorithm will move to the next half that has not yet been removed from the list of inputs. This can be seen in step 3 and 4 of the table, here the algorithm removes [c,d], since [b] can not be split into smaller pieces. Removing [b] would have resulted in inputs [c,d,e], which was already tested in step 2.

The last introduced algorithm can be used to eliminate cycles. By looking at the states visited in

	Inputs	Tested inputs	Result
1	~~[a,b,c,d]~~	[e]	Pass
2	~~[a,b]~~[c,d]	[c,d,e]	Pass
3	~~[a]~~,[b],[c,d]	[b,c,d,e]	Fail
4	~~[a]~~,[b],~~[c,d]~~	[b,e]	Fail

Table 4.1: A simple overview of the binary elimination of the trace $a \cdot b \cdot c \cdot d \cdot e$.

the model (the specification), a cycle can be detected. A (sub)trace is a cycle if the same state of the model is visited twice. The cycles that are found are sorted from longest, to shortest, and removed from the trace in this order. After a cycle is removed the trace is tested again. If the trace with a cycle removed still shows non-conformance, the cycle is removed permanently form the trace, otherwise it is added back into the trace.

They continue to compare their algorithms on a state machine of a vending machine. They tested on 10 different mutants, that each introduced one specific bug. Each mutant is selected to represent common mistakes made by programmers. Based on measurements, it is concluded that cycle elimination, followed by binary elimination, is the best way to shorten traces.

The authors claim that the vending machine system is representative of a real-world system, but that more research is needed to validate the results for larger, real-world applications.

4.2 Delta debugging

Delta Debugging is an algorithm that was originally described by Andreas Zeller [5]. Imagine there are two versions of the same program. The old version of this program passes a regression test, but the new version does not. Between the versions many changes have been made, how can it be determined which change, or combination of changes caused this regression test to fail? Delta-debugging is an algorithm to find the minimal set of changes that causes a test to fail.

A change is a modification to a program than can be applied and unapplied. One change can be a single line of code that has been modified. But multiple modifications can also be grouped as a single change, for example by time. Another option is to group modified lines close to each other together as a single change. A single commit in a versioning system such as git can also be seen as a single change.

Let $C = \{\Delta_1, \Delta_2, ..., \Delta_n\}$ be the set of all possible changes. A change set $c \in C$ is called a configuration. A configuration constructed by applying the changes to a baseline. The baseline is an empty configuration, $c = \emptyset$. In other words, it is the old version of the program, that is known to work.

The test function $2^c \rightarrow \{\text{✗}, \text{✓}, \textbf{?}\}$ determines for a configuration whether a failure occurs (✗), it passes (✓) or it is unresolved (**?**).

The old version of the program worked, this baseline passes the test, so $test(\emptyset) = \text{✓}$. On the other hand, the new version of program, with all changes applied, does not work. This means the test fails, $test(C) = \text{✗}$. A failure-inducing change set is any set of changes S for which $test(S) = \text{✗}$.

Definition 4.2.1 (**Minimal failure inducing-set**)**.** A failure-inducing change set $B \subseteq C$ is minimal if

$$\forall c \subset B \ (test(c) \neq \text{✗})$$

$holds$

In other words, the minimal failure-inducing change set is a set of changes, that when applied to the baseline, makes a test fail. Any subset of this minimal set of changes, will not make the test fail. The delta debugging algorithm is a divide and conquer method to look for such a minimal failure-inducing set.

The basic delta debugging algorithm can be described as follows: If a set of changes c fails a test, divide it into (roughly equally sized) subsets c_1 and c_2, and test them both. This can have three outcomes:

1. The test with c_1 fails, a failure inducing change is in c_1.
2. The test with c_2 fails, a failure inducing change is in c_2.
3. Both tests pass. The failure is cause by a combination of changes in c_1 and c_2. When a combination of two or more changes cause a failure, this is called interference.

In case one and two, the algorithm can just be repeated in the subset that fails. In case three, the algorithm is applied again on both subsets individually, but with all changes in the other subset applied.

The next section of the paper discusses how to deal inconsistencies. If a set of changes is applied, this can cause inconsistencies. For example code might not compile when one set of changes is applied, but another (that fixes this) is not applied. If an inconsistent configuration is tested, the result of the test will be unresolved. The dd^+ algorithm is an adaptation of the dd algorithm that can handle inconsistencies.

4.2.1 Delta debugging minimisation

The algorithm introduced in the previous section was later adopted by Zeller and Hildebrand to find a minimal test case, that reproduces a failure [1]. The terminology and definitions from the previous section about delta debugging are used in this section. The Delta Debugging minimisation paper introduces two new delta debugging algorithms.

The first algorithm is called $ddmin$. This algorithm is different from the dd^+ algorithm because it finds the minimal failing test, instead of the minimal set of changes that causes a test to fail. More specifically the $ddmin$ algorithm will find a test case from which removing any part will cause the test to pass. This test case might be the smallest failing test case possible. It might also be possible that there is a smaller failing test, that cannot be derived from the test case on which delta-debugging minimisation is applied. In other words, $ddmin$ will find a local minimal test case and not a global minimum test case.

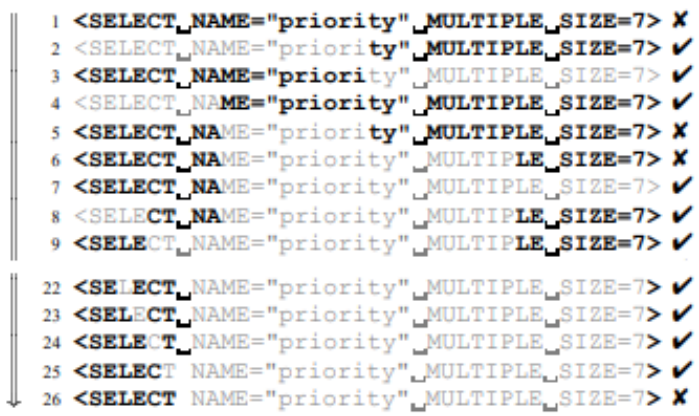

Figure 4.2: Visual representation of the minimisation algorithm. (Excerpt of Figure 2 in [1])

The paper uses some HTML that makes a browser crash as an example. The test in this case is 'does the browser crash'. In this example, each line is considered *change* at first. The baseline for this test is an empty HTML page. When all changes are applied to the baseline, the result is the complete HTML page that makes the browser crash. By applying the detla debugging minimisation algorithm explained below, a single line that makes the browser crash can be found.

When the line that causes a failure is found, this line can be further minimised. This is done by making each character a change and applying the minimisation algorithm again. Figure 4.2 on the next page shows the result of $ddmin$ on the single line where each character is considered a change. Each line in this figure is a test case (several lines in the middle have been omitted). If the browser crashes the test fails (marked with ✗). If the browser doesn't crash the test passes (marked with ✓). This is the case for the minimisation shown in Figure 4.2.

The delta debugging minimisation algorithm works as follows. The set of changes is split into two subsets Δ_1 and Δ_2, each subset is tested, this gives three options:

- Δ_1 fails.
- Δ_2 fails.
- Δ_1 and Δ_2 both pass, or are unresolved.

In the first case, Δ_1 is a smaller failing test, and the process can be continued by repeating the algorithm in this subset. The same is true for Δ_2.

If all tests pass or are unresolved, as in the third case, the test is split into smaller subsets, each subset, and the complements of these subsets are tested. If one of these tests fails, this is the new smaller test case, and the algorithm is repeated. If there comes a point again where all tests pass or are unresolved, the sets are split into smaller subsets and the process is repeated. This is the case on line 3 and 4 of Figure 4.2.

The other newly introduced algorithm is a replacement for the original dd and dd^+ algorithms. This algorithm, like the original delta debugging algorithm, is used to find a failure inducing difference.

4.2.2 Applications of Delta Debugging

The $ddmin$ algorithm was successfully applied by Lei and Andrews to shrink automatically generated unit tests [14]. Zeller et al. also created a combination of slicing [15] and the $ddmin$ algorithm that can automatically shrink generated unit tests. This effort was more efficient than the earlier attempt by Lei and Andrews [16].

The delta debugging minimisation algorithm has also been applied to model-based testing, specifically as a part of model-based testing of satisfiability (SAT) solvers [17]. Another paper on model-based testing of satisfiability modulo theories (SMT) solvers [18] uses the techniques of the first paper but applies it to SMT solvers. This section only explains the first paper.

This paper proposes a way to apply model-based testing to the SAT solver Lingeling. The authors of this paper use three kinds of models to generate tests: The option model, that describes the valid options or combinations of options, the API model, that describes traces of valid calls to the SAT solvers API and the data model, representations of valid formulas. These are used to generate test, and executed against Lingeling. Error traces are then shrunk using delta debugging. During the delta debugging process the debugger makes sure that the new shrunk traces are still valid according to the API model.

The effectiveness of the method is shown by an empirical evaluation. In the evaluation, the authors looked at code coverage, throughput and the number of injected defects found using model-based testing. The paper also introduces a metric called relative trace length, that is used in the proposed method (see section 5.4) to give an indication of the effectiveness of shrinking.

The paper concludes that “MBT is substantially more effective in finding defects than previously used techniques” and that delta debugging is particularly effective in shrinking traces.

Based on the papers found during the literature review, the delta debugging minimisation algorithm has not been used to shrink traces for tests generated from LTSs or STSs. The $ddmin$ algorithm has seen success in different types of generated test cases. The minimisation algorithm algorithms handles unresolved test cases, this might be useful for shrinking traces of models that are not input enabled. If a new trace does not fit in the model anymore, the test can be marked as unresolved.

4.3 Other ways of shrinking traces

Several other papers have tried to shrink test cases. For example, in the field of automated user-interface testing several studies have looked at trace reduction. A paper by Clapp et al. implemented a modified delta debugging algorithm [19] to shrink GUI event traces. Another paper by Heiskanen et al. minimises GUI error traces by replaying only the last step in the trace, and gradually adding more steps back until the test fails again [20]. This section focuses on shrinking traces for model-based testing of state based systems.

4.3.1 QuickCheck on Finite State Machines

QuickCheck is a tool, originally written for the functional programming language Haskell. The tool is used for automatic, random testing of properties in Haskell programs. [21] QuickCheck was later adopted to several other languages, including Java and Erlang.

One paper discusses a way to generate QuickCheck tests based on finite state machines in Erlang [22]. The paper explains a way to model an Erlang system as a finite state machine and generating tests based on such a state machine.

For each command (a command is a method in a program, such as subtract) a precondition is defined, it tells the test generator if it is appropriate to generate that specific command given the current state. The postcondition is used to check if after execution, the command satisfies the properties that it should satisfy.

The QuickCheck shrinking process deletes commands from a test case. Deleting a command might change the state that the next commands are executed in. If a certain command changes the state of a program, from s to s', deleting this command will cause a subsequent command to be executed in state s, instead of s'. It can however be the case that the subsequent command is not appropriate for state s. For this reason, preconditions are used to check if the command is still appropriate for the new state if a command is deleted.

It is clear that the shrinking method used in QuickCheck is not suitable to accomplish the goals of this proposal. This method method requires defining preconditions and postconditions for every command. This would introduce significant cost and effort to update already existing models.

4.3.2 Trace reduction

The paper “Trace Reduction and Pattern Analysis to Assist Debugging in Model-Based Testing” [12] discusses analysis of failures in model based testing. The authors propose a technique to aid in root cause analysis, based on pattern mining. To make this pattern mining easier, the authors briefly explain a way to shrink (or reduce) traces.

Test cases are generated using a MBT tool called OSMO. Models in OSMO are annotated Java programs. Such a model basically consists of a set of actions, and guards. Actions are stimuli that can be applied to the SUT. A guard defines when a certain action is allowed. In the model postconditions can be defined for each action. Postconditions can be used to check if the state of the model matches the state of the SUT. In OSMO scenarios can be defined. Scenarios can be used to focus test generation to a specific part of the model. A scenario sets the minimum and maximum amount of times a certain action can be generated by the OSMO test generator.

Action	Guard
Increase	
Decrease	amount $>$ 0

Action	Minimum	Maximum
Increase	0	18
Decrease	0	11

Table 4.2: A model (left) with two actions and an example scenario (right) that limits the generation of increase and decrease actions to a maximum of 18 and 11 respectively.

First, test cases are generated, until a preset number of tests fail, or a timeout is reached. The set of failing test (FS) is then put into a reduction algorithm that works as follows:

1. Set the maximum test length of the generator to the length of the shortest failing test in the set of failing tests (FS).
2. For each test in the set of failing tests
   (a) Create a scenario where the maximum number that each action can be generated is reduced by one, compared to the failing test. For example, if a failing test had 12 *decrease* steps, the generator will be limited to generate a maximum of 11 *decrease* steps.
   (b) Generate new tests using this scenario.
   (c) Add every found failing test to a new set (FS2) of failing test.
3. If FS2 has one or more elements, set FS = FS2 and repeat from step 1.

The described algorithm does not shrink a previously found trace. Rather, it creates a trace from scratch, but with constraints based on a previously found trace. This way is used by the authors to not only find shorter traces, but also different traces of the same length. These traces can then be used for pattern mining. Since the algorithm relies on generation of completely new tests, it is likely that it does not scale well for longer traces. Just like the similar binary search algorithm by Koopman et al. [2] that also relies on finding completely new tests.

5 RESEARCH METHOD

This research investigates shrinking algorithms for failing test cases that were generated from systems that are modelled as an STS. To answer the research question “What method can be used to shrink traces of failing tests in test cases derived from Symbolic Transition Systems?” a practical approach will be used. By using a practical approach, and doing an experiment on a real-world system, we can verify that these algorithms are also applicable to the real-world, and not just in an academic setting.

The paper by Koopman et al. [2] discussed in section 4.1 will be used as a starting point for these experiments. The three algorithms from this paper have been implemented and integrated into the AMP. Two additional algorithms, one algorithm based on the Delta Debugging minimisation algorithm [1] and an adaptation of Koopman’s cycle shrink algorithm, called location cycle shrink, have also been implemented. Details about AMP, and the implementation of these algorithms can be found in Chapter 6.

To test the five implemented algorithms algorithms, two experiments have been setup. In both experiments several bugs were introduced into a SUT. Using these bugs, failing tests were generated and subsequently shrunk using the test case shrinking algorithm.

The first experiment uses the same vending machine as the one that was used in the experiment done by Koopman et al. [2]. This experiment also tests two newly introduced shrinking algorithms.

The second experiment is an experiment tests the shrinking algorithms on a real system from ProRail, that is currently tested with Axini software. ProRail is a Dutch government agency responsible for maintenance and traffic control of the Dutch rail network.

5.1 Experiment One: Simple Vending Machine

The first experiment uses a vending machine that was included in the paper by Koopman et al. [2]. This vending machine accepts coins with a value of one or two. A product can be chosen, if the user presses go after a product has been chosen and there is sufficient balance, the machine should produce the selected product. Otherwise no product will be returned. There’s also a reset button that should return any remaining balance, and unsets the currently selected product.

In the original paper the vending machine specification was modelled as an Extended State Machine. This specification was translated to an STS and written in the Axini Modeling Language (AML). Figure 5.1 shows the STS of the specification of the vending machine.

The STS of the vending machine uses 2 location variables, *bal*, which represents the current balance of the machine and *chosen* which represents the product that is currently selected. Below each label, there can be a transition constraint, which is shown in bold. In a constraint || means a logical OR while && means a logical AND. Updates of the location variables are

shown beneath the constraint (if there are any). The function *val(chosen)* returns the value of the product in chosen (2 for coffee and 3 for all other products). The restriction **Product in items** simply means the value for the interaction variable *Product* must be in the list of items that the vending machine offers.

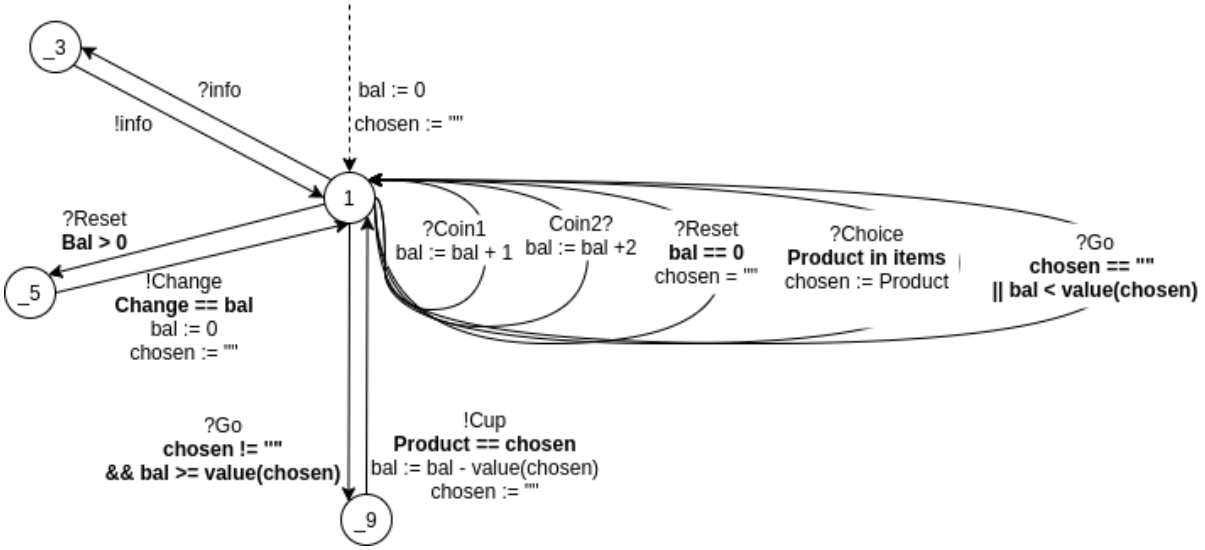

Figure 5.1: STS model of of the vending machine used to evaluate the algorithms. Transition constraints are shown in bold, updates are shown under the constraints. The variable bal contains the current balance of the machine, chosen contains the item that is currently selected. The method value(chosen) will return the value of the chosen product.

The paper also includes a correct implementation and ten mutants of this implementation. Each mutant is a copy of the correct implementation, that introduces exactly one bug. These faulty implementations were translated to ruby and are used to generate failing test cases.

The failing test cases were shrunk by the algorithms and combinations of algorithms detailed in Section 5.3.

5.2 Experiment Two: Real-World System

The second experiment is an an experiment on a real system that is tested using AMP. This system is from ProRail and is tested by Axini.

ProRail manages the Dutch rail network. To do this ProRail uses several systems, one of which is the Procesleiding (PRL) system. The system is used by ProRail to manage train traffic in real time. PPR-AAP is an interface of PRL with which other systems can subscribe to the train schedule. PRL provides the client with a snapshot of the current schedule and notifies them of any subsequent changes to the schedule. Clients can unsubscribe at any time.

Axini has a model for PPR-AAP along with a simulator which is known to be a correct implementation of the specification. For this experiment a simulator was used to make it easier to test all the bugs without relying on another party. From this simulator 15 copies with one bug each were made.

The bugs introduced into the system are based on expert opinion of common bugs in systems like PPR-AAP. The bugs were constructed in the following way:

1. Two experts examined reports of bugs found in PPR-AAP and several other systems that are similar to PPR-AAP.
2. From this, a list of the type of bugs that are generally found in systems similar to PPR-AAP was made.
3. Based on this list, fifteen mutants were made, that resemble one of the common bugs from the list the experts constructed.
4. Each bug was then verified by one of the experts to confirm that it was indeed a bug similar to what they had seen in real systems.

For each of the fifteen bugs, a patch file was created, which would be applied before generating and shrinking the failing test cases. Each bug was again used to generate failing test cases, which were subsequently shrunk by the algorithms and combinations of algorithms detailed in Section 5.3.

5.3 Evaluated shrinking algorithms

During these experiments, the following shrinkers five shrinking were evaluated:

- Element Elimination Shrinking algorithm (only during the experiment on the vending machine).
- Binary Elimination Shrinking algorithm
- Delta Debugging Shrinking algorithm
- Cycle Shrinking algorithm
- Location Cycle Shrinking algorithm

Because of the very high number of interactions with the SUT using the Element Shrinker on the vending machine, it was decided not to test this shrinker in the second experiment.

The Cycle Shrinking and Location Cycle Shrinking algorithms shrink test cases less on average than the other algorithms. These algorithms can be followed up by another algorithm to get a better result. The following combinations of shrinking algorithms were evaluated:

- Cycle Shrinking algorithm followed by the Element Elimination Shrinking algorithm.
- Cycle Shrinking algorithm followed by the Binary Shrinking algorithm.
- Cycle Shrinking algorithm followed by the Binary Elimination Shrinking algorithm.
- Location Cycle Shrinking algorithm followed by the Element Elimination Shrinking algorithm.
- Location Cycle Shrinking algorithm followed by the Binary Elimination Shrinking algorithm.
- Location Cycle Shrinking algorithm followed by the Delta Debugging Shrinking algorithm.

5.4 Result Validation

Each implemented algorithm needs to be evaluated in both experiments. To evaluate the algorithms, several metrics have been measured:

- **Shrinking effectiveness**: The length of the new trace, as a percentage reduction of the originally found trace. If a trace is 100 steps at first, this will be 0%, if a trace is then reduced to 34 steps, the new trace will be a 66% reduction. This metric is an indication of how well a certain algorithm shrinks a trace. Since effectively shrinking traces is the main goal of this research, this is the most important metric. The MBT SAT paper [17] also measured the relative length after shrinking.
- **Shrinking time**: The amount of time taken for the shrinking process. During the shrinking process, a test has to be executed many times, to see if a newly shrunken trace still fails the test. Because running a test on a SUT many times takes time, it is important to know how much time the process takes. This metric is important, but it is not as important as the effectiveness. Time is a metric that is often used. The MBT SAT paper [17] uses this metric in their evaluation.
- **Number of interactions with the SUT**: Each input to the SUT is counted as an interaction with the SUT. While some SUTs can be very fast, the interaction with the SUT is often time consuming in the execution of a test. This metric was also used by Koopman et al. in [2].
- **Number of executed test cases**: The number of test cases executed to shrink a test case. After each test, the SUT needs to be reset to the initial state to start a new test case. This metric more important for SUTs that have an expensive reset sequence.

These metrics give insight in how effective the different algorithms and combinations of shrinkers are. The algorithm that performed the best based on these metrics is the answer the research question “What method can be used to shrink traces of failing tests in test cases derived from Symbolic Transition Systems?

5.4.1 Research questions

The goal of these experiments is to answer the sub questions that were first introduced in Chapter 3. This section will provide an overview of the relation between each sub question and the experiments. First, we will discuss the questions related to the effectiveness and efficiency of shrinking algorithms:

1. *“Can the results of Koopmans et al. be reproduced on Symbolic Transition Systems?”*
2. *“How do the algorithms by Koopman et al. perform when they are algorithms applied to real-world systems?”*
3. *“Can modifications be made to these three algorithms to get shorter traces or complete the shrinking process faster?”*
4. *“How effective is the delta-debugging minimisation algorithm for shrinking traces?”*

The first sub question was answered by performing the first experiment, while the second sub question will be answered using the second experiment. The last two sub questions will be partially answered in both experiments: The implemented Location Cycle Shrinking algorithm and Delta Debugging Shrinking algorithm were used as shrinking algorithms in both experiments.

The sub question *“What is an effective way to handle models that are not input enabled for the element elimination and binary elimination algorithms?”* was not tested in either experiment. When a shrinking algorithm tries to provide a stimuli to the SUT that was not defined in the model, the test will be aborted. The test case will then be treated as if it passed, so the shrinking was unsuccessful.

Finally, the sub question *"What other techniques, such as shortest path algorithms, can be used to shrink traces and how effective are they?"* will be left as future work. During the course of this research a simple shortest path algorithm was implemented and experimented with. The results of this experimentation is however, not yet advanced enough to be a meaningful inclusion in this thesis.

6 DESIGN AND IMPLEMENTATION

This chapter describes the design and implementation of the algorithms explained in section 4.1 and 4.2.1 used to shrink test cases. The implementation was directly integrated into AMP. The chapter will give specifics about the design and implementation of the shrinking algorithms. In section 6.4 there will be a short example of how each algorithm works when shrinking test cases within AMP.

6.1 Axini Modeling Platform

The Axini Modeling Platform (AMP) is software that is used for Model-Based Testing (MBT) using Symbolic Transition System. To understand the implementation and integration of the shrinking algorithms, first an understanding of the Axini Modeling Platform is needed.

AMP consists of two main components: A graphical user interface (GUI), in the form of a web-interface that a user can use to interact with AMP and the STS Engine, which is an implementation of STSs and the formal parts of AMP. Figure 6.1 shows a simplified overview of the Axini Modeling Platform.

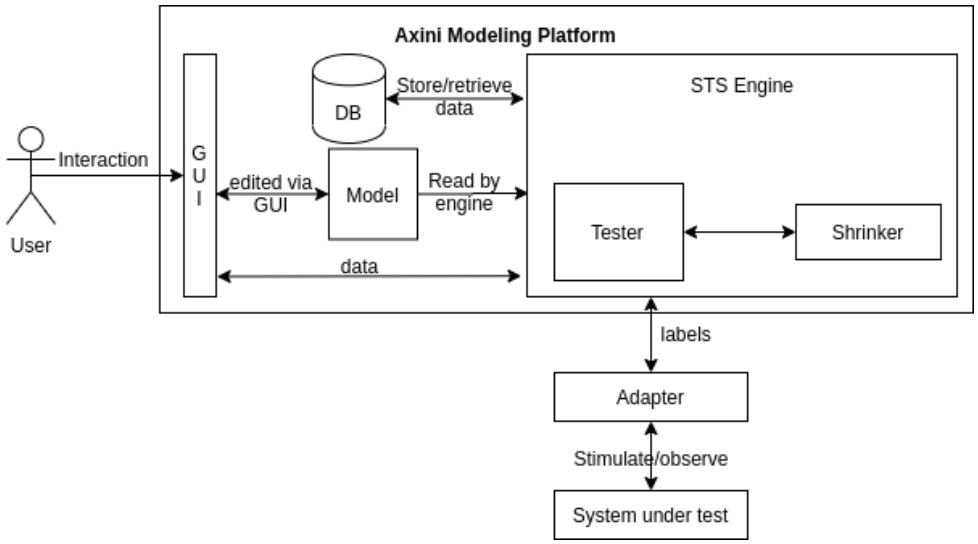

Figure 6.1: A simplified overview of AMP. A user interacts with the system using a web-based GUI. Models are read by the STS Engine. The tester uses the model to generate tests on-the-fly, which are executed against the SUT using an adapter.

The web-interface is the main way to interact with AMP. Using this interface a user can create models and update models. Models are written in a special language created by Axini, the Axini Modeling Language (AML). The user can also view visualisations of the models, and explore this

model step by step using the model explorer. A user can start test runs, view which transitions and states were covered by a test run and view the history of previous test runs.

Lastly, a user can configure settings that AMP uses while testing. This includes basic settings like the number of executed tests and the maximum number of steps in each test as well as SUT specific configuration options, such as connection details to connect to the SUT. The user can also configure which strategy to use to generate test cases. This strategy is used for selecting the next input during the generation of a test case. A strategy usually tries to maximise a coverage criteria, for example, the percentage of transitions covered during testing (transition coverage).

6.1.1 Test cases in AMP

The STS Engine provides an implementation for the formal parts of AMP. The model, which is written in AML, is read by the engine to create an STS model that can be used within the engine. The 'Tester' component of the STS Engine contains an on-the-fly test case generator. The generator works similar to the on-the-fly test generation algorithm for LTSs explained in chapter 2. Instead of randomly selecting the next input, the tester will ask the configured strategy what input to apply to the system next.

Input labels that are generated by the test generation algorithm have to be translated to a format that can be understood by the SUT. Outputs from the SUT have to be translated back to labels that AMP can understand. This translation is done by an adapter. For every SUT, an adapter has to be created. To create an adapter, an interface provided by AMP has to be implemented.

A test case stops when it fails, this happens when an unexpected output is observed, or when quiescence was observed when there should not have been quiescence. A test case also stops when the configured maximum number of steps is reached, at which point the test passes. The verdict of all test cases is communicated back to the user via the GUI. Using the GUI, the user can inspect the traces of each test case. In case a test case failed, the user can see why a test case failed, and which outputs were expected.

Figure 6.2: The information a user has to debug a failing test. This shows that after $Reset$ input, the system output $Change$, where the amount of change was 2. Since the balance in the machine was not two, this is an unexpected response. The interface also shows two responses that were expected.

Figure 6.2 shows the debug information a user has available after a failing test. The top of the image shows (part of) the trace that lead to the failing test. The last step in red is an unexpected response. In this case, the machine had a balance of three, so the change should have been three, not two. Underneath this are expected responses, these are the responses that would

have been accepted. It also shows that the condition $Change == bal$ was not met by the current output.

6.2 Design of implementation

The shrinking algorithms were directly integrated into the AMP. The most important class of the Tester component of AMP shown in Figure 6.1 is the TestApi class. Normally the TestApi uses the configured strategy to select the next input. During shrinking, the shrinker will provide the next input instead. Figure 6.3 shows a class diagram of the architecture of the implemented solution.

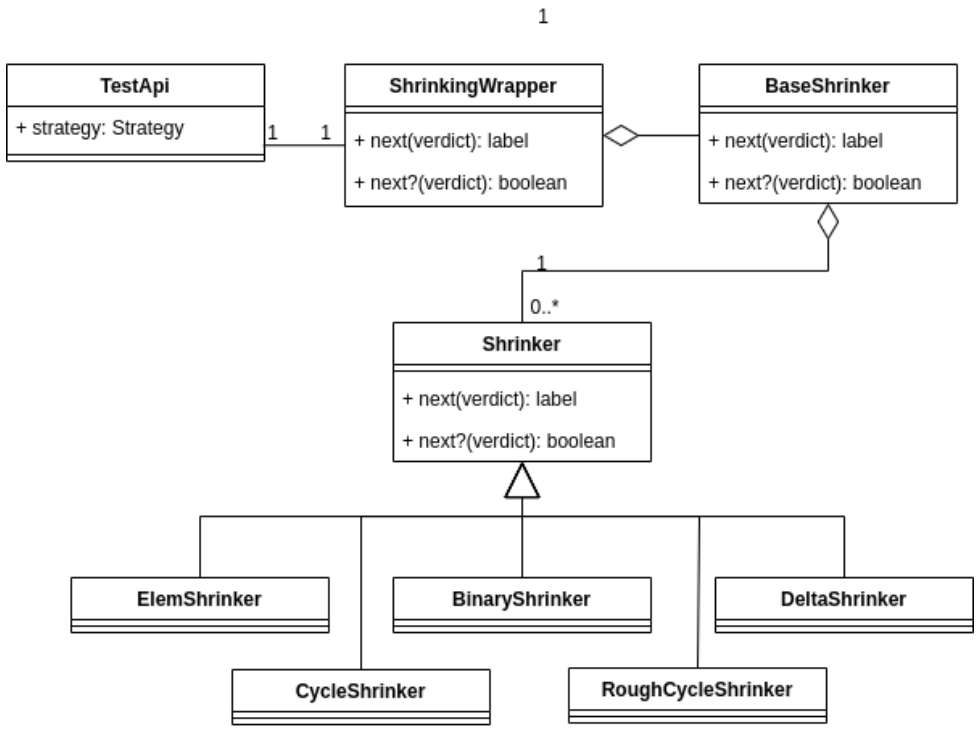

Figure 6.3: A simplified overview of AMP. A user interacts with the system using a web-based GUI. Models are read by the STS Engine. The Tester component uses the model to generate tests on-the-fly, which are executed against the SUT using an adapter.

The classes are each responsible for various tasks:

- **Shrinker**: An interface that provides the methods each concrete shrinker should implement. The method $next?$ should return true if there is a next step in the shrinking process, based on the verdict of the previous test case. The method $next$ returns the next list of inputs that should be tested in the shrinking process.
- **BaseShrinker**: The BaseShrinker handles the shrinking of exactly one test-case, using either one, or a combination of shrinking algorithms. When a combination of algorithms is used, the BaseShrinker will pass the result of the first shrinking algorithm to the next shrinking algorithm.
- **ShrinkingWrapper**: The primary purpose of the ShrinkingWrapper class is measuring the metrics specified in chapter 5. It collects every label sent to and received from the SUT and will report statistics after shrinking is done. The ShrinkingWrapper outputs a summary to the console and a log file, full data will be saved to a comma separated values (csv) file.

The classes under the Shrinker interface are the concrete implementations of each shrinking algorithm.

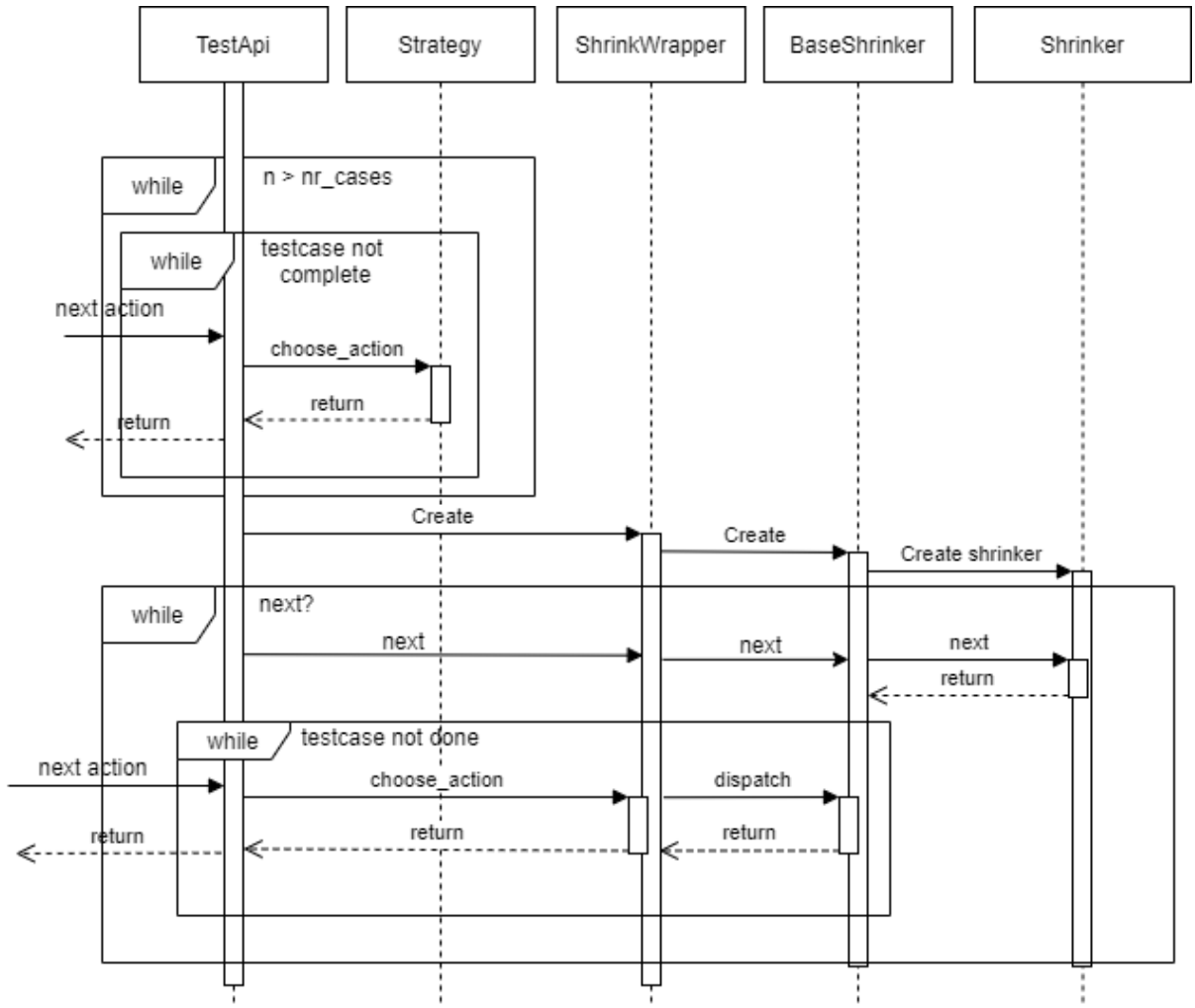

Figure 6.4: A simplified overview of the shrinking process.

Figure 6.4 shows a simplified sequence diagram of the shrinking process. The block at the top of the sequence diagram shows the generation of test cases to shrink. When n test cases have been generated, the shrinking process will begin.

The shrinking process begins by creating A Shrinking Wrapper. The diagram only shows the process for one shrinking algorithm. Everything after the creation of the ShrinkingWrapper will be repeated for each shrinking algorithm that is being evaluated.

The while $next?$ loop will run while the method $next?$ returns true, if this method returns true, it means that there are still test cases left to test in the shrinking process. The method $next$ will signal the shrinking algorithm to start the next test case. The TestApi will request next actions from the BaseShrinker, until the test case is done. The test case is done when it fails, when a shrinking algorithm supplies an input that is not defined in the model, or when the last input of the test has been given (at this point the test passes).

6.3 Implemented Algorithms

In total five algorithms were implemented. Three of these algorithms were first described by Koopman et al. in [2]. The fourth algorithm is an adaptation of Delta Debugging to shrink test cases derived from STS. The fifth algorithm is an adaptation of the Cycle Shrink algorithm by Koopman et al. called Location Cycle Shrink.

6.3.1 Koopmans Algorithms

The three algorithms by Koopman et al [2] were adopted to work on test cases that were generated from an STS model. In a FSM one transition has both an input and an output, while in the ESM originally to test these algorithms, a transition has both the input and zero or more outputs.

In an STS this is not the case, a transition is either an input or an output. An executed test case can be thought of as list of input labels and output labels. A model can consists of multiple STSs. The TestApi will will keep track of a list of labels and a list of states, including valuation after each transition.

For both the element shrinking algorithm and the binary shrinking algorithm only a few changes had to be made. From the list of labels, all outputs are removed. The algorithms by Koopman will then be applied to a list of inputs. The TestApi will request the next input from the instead of a strategy, like it normally would.

The cycle detection algorithm for the Cycle Shrinking algorithm had to be slightly modified. The Cycle Shrinking algorithm is given a list of labels and a list of states, starting with the initial state of the STSs.

The list of states at index i contains the state of the STS before the label at position i in the list of labels. First, all output labels and their respective states are removed.

The method find cycles will now find all cycles based on the state names, and the valuation of the states. This can be seen in the following code fragment:

```
1  def find_cycles
2    cycles = []
3    @states.each_with_index do |_b, i|
4      @states.each_with_index do |_e, j|
5      # Check if the name of the states of all STSs is the same
6        if j > i && @states[i].name == @states[j].name
7        # Check if the valuation of the states of all STSs is the same
8        && @states[i].valuation == @states[j].valuation
9          cycles << [i, j]
10       end
11     end
12   end
13   # Sort cycles from longest to shortest and return them
14   cycles.sort! {|a, b| b[1] - b[0] <=> a[1] - a[0]}
15 end
```

Code Fragment 6.1: Algorithm to find cycles and sort them by their size.

This rest of this algorithm works the same as described by Koopman, it will remove the largest cycle, from the test case and try to test this new test case. If the new test case fails, the cycle will

be permanently removed, otherwise the cycle is put back and the algorithm will try to remove the next largest cycle.

6.3.2 Location Cycle Shrinking

The Location Cycle Shrinking algorithm is an adaptation of Koopman's Cycle Shrinking algorithm. The original Cycle Shrinking algorithm uses the location, as well as the valuation of the location variables to determine cycles. The Location Cycle Shrinking algorithm only cares about the location determine cycles.

This means that a sub-trace is considered a cycle, if the location that the STS is in, is the same. In practical terms, this means that as line 8 of the Cycle Shrinking algorithm can be removed. The code from line 3 to 12 will be replaced by the following code.

```
1 @states.each_with_index do |_b, i|
2   @states.each_with_index do |_e, j|
3     # Check if the name of the states of all STSs is the same
4     if j > i && @states[i].name == @states[j].name
5       cycles << [i, j]
6     end
7   end
8 end
```

Code Fragment 6.2: This code fragment replaces lines 3 to 12 in Code Fragment 6.1 when looking for cycles based only on the location.

The algorithm works very similar to the normal cycle shrinking algorithm:

1. From the trace of the failing test case remove all outputs.
2. Determine all cycles based on the location the model is in just before each input.
3. Order these detected cycles from largest to smallest.
4. For each cycle:
   (a) Remove the the cycle from the trace, remove all overlapping cycles from the list of cycles to try to remove and retest with this cycle removed.
   (b) If the test fails, keep this cycle removed, otherwise go back to the previously failed test.

After this process is finished, the shortest trace without any cycle that can be removed will remain.

6.3.3 Delta Debugging

A shrinking algorithm based on the Delta Debugging Minimisation algorithm [1] explained in section 4.2 was implemented.

This algorithm, tries to remove larger part of inputs from failing traces, it is in a way similar to the binary elimination algorithm by Koopman et al. but based on the Delta Debugging minimisation algorithm.

This algorithm, just like all others starts by removing all output labels from the trace, leaving only the input labels. The final input will also be removed, this label will be added as the final label for each test case generated by the delta debugging algorithm. Since we know this is the

label that causes the test case to fail in the end, it would be nonsensical to include this label in the input for delta debugging.

The code fragment below shows how the main part of the Delta Debugging Shrinking algorithm. This is the implementation of the $next()$ method for Delta Debugging. This is the method that returns the next test case to be tested.

```
# Create a list of subsets to test  (T)
# create a list of complements to test (C)
# Filter test cases from (C) That already exist in (T),
# Apply ddmin Algorithm
def next(verdict)
  # Case: Testing subsets.
  if @subset
    # Reduce to subset, if the subset failed.
    if verdict == :failed && @last != nil
      @n = 2

      start_next
      return @last
    end

    # There are more subsets to test, continue with this
    if @tests != []
      # More tests do execute
      @last = @tests.shift
      return @last
    else
      # No more subsets to test, move on to complement testing.
      @subset = false
      @complement = true
    end
  end

  # Case: Testing complements
  if @complement
    # Reduce to complement if the test failed
    if verdict == :failed
      @n = [2,@n-1].max

      start_next
      return @last
    end

    # If the test passed, move on the next complement, if there are any
    if @complements != []
      @last = @complements.shift
      return @last
    else
      # done testing complements, increase granularity.
      @complement = false
    end
```

```
46      end
47
48      # Case: Increase granularity.
49      if @n < @inputs.size
50        @n = [@inputs.size, 2*@n].min
51        @last = @inputs
52        start_next
53        return @last
54      end
55    end
```

Code Fragment 6.3: The $next()$ method for the Delta Debugging shrinking algorithm. The implementation shows how the algorithm works within AMP.

You can clearly see the steps that steps of the $ddmin$ algorithm: Reduce to Subset, Reduce to Complement and increase granularity. The variable `@n` keeps track of the granularity. The method `start_next` is a helper method that makes the next test-cases based on the granularity and the last failed test. It creates the subsets and compliments that will be tested next.

6.4 Shrinking examples

This section will show a short example of how each algorithm works while shrinking test cases generated from STSs. These examples give some insight into the working of the algorithms in practice and shows of the differences between the algorithms.

Consider the STS shown in Figure 6.5, a simple vending machine that accepts coins with a value in euro. It can output a cup of coffee, the user can choose to reset it to get their money back. It has a bug where it does not return any remaining balance when a user confirms a reset, after at least one coffee has been ordered.

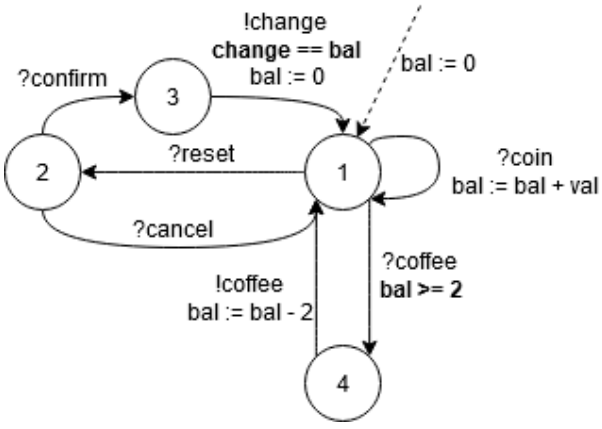

Figure 6.5: A simple STS that accepts coins with a value in euro. It can output a coffee when the balance is over two euro. After a reset it should return any balance to the user. The implementation used in the example does not return change if at least one coffee has been ordered, even if there is balance remaining.

The Consider the failing test case shown in Table 6.1. The test fails because the remaining balance is not returned to the user, the change given back to the user should be one.

#	Transition	Location after Transition	Valuation
1	(initialisation)	1	bal = 0
2	?coin (val = 1)	1	bal = 1
3	?coin (val = 2)	1	bal = 3
4	?coffee	4	bal = 3
5	!coffee	1	bal = 1
6	?reset	2	bal = 1
7	?cancel	1	bal = 1
8	?coin (val = 2)	1	bal = 3
9	?coffee	4	bal = 3
10	!coffee	1	bal = 1
11	?reset	2	bal = 1
12	?confirm	3	bal = 1
13	!change **(change = 0)**	-	-

Table 6.1: A failing test case, the test fails because the remaining balance is not returned.

The examples below will show the first few steps of every shrinking algorithm. The examples use a shorthand notation to keep them concise. The value of inserted coins is indicated with (v=). The balance of the machine is only shown after the location if the balance changed from the last step, indicated by (bal=). Each algorithm works on the list of inputs of a failing test. This results in the list: [?coin(v=1), ?coin(v=2), ?coffee, ?reset, ?cancel, ?coin(v=2), ?coffee, ?reset, ?confirm].

A shrink step is successful if the newly executed test also fails. A shrink step is unsuccessful if the test passes, or if the test cannot be completed because it tries to give an input that is not specified in the model. Each step in the algorithms is shown as follows:

1. The list of inputs the algorithm is operating on. Inputs that are removed in the current shrink step have a line through them.
    - The complete test case that was executed in this shrink step.
    - A short description of what happened and an indication if the shrink step was successful or unsuccessful.

6.4.1 Element Elimination Shrinking algorithm

The element elimination algorithm tries to remove a single input from the list of input during each step of the shrinking process. If a shrink step is successful, the latest removed input will be permanently removed, otherwise it will be put back into the list of inputs. This will result in the following test-cases being executed:

1. ~~?coin(v=1)~~, ?coin(v=2), ?coffee, ?reset, ?cancel, ?coin(v=2), ?coffee, ?reset, ?confirm
    - 1 $\xrightarrow{?coin(v=2)}$ 1 (bal=2) $\xrightarrow{?coffee}$ 4 $\xrightarrow{!coffee}$ 1 (bal=0) $\xrightarrow{?reset}$ 2 $\xrightarrow{?cancel}$ 1 $\xrightarrow{?coin(v=2)}$ 1 (bal=2) $\xrightarrow{?coffee}$ 4 $\xrightarrow{!coffee}$ 1 (bal=0) $\xrightarrow{?reset}$ 2 $\xrightarrow{?confirm}$ 3 $\xrightarrow{!change(c=0)}$ 1
    - The tests passes because there's no remaining balance to be returned. The shrink step was unsuccessful.
2. ?coin(v=1), ~~?coin(v=2)~~, ?coffee, ?reset, ?cancel, ?coin(v=2), ?coffee, ?reset, ?confirm
    - 1 $\xrightarrow{?coin(v=1)}$ 1 (bal=1) $\xrightarrow{?coffee}$
    - The tests stops after the first ?coffee input, since in location 1 there's no ?coffee input defined for bal = 1. The shrink step was unsuccessful.

3. ?coin(v=1), ?coin(v=2), ~~?coffee~~, ?reset, ?cancel, ?coin(v=2), ?coffee, ?reset, ?confirm
   - 1 $\xrightarrow{?coin(v=1)}$ 1 (bal=1) $\xrightarrow{?coin(v=2)}$ 1 (bal=3) $\xrightarrow{?reset}$ 2 $\xrightarrow{?cancel}$ 1 $\xrightarrow{?coin(v=2)}$ 1 (bal=5) $\xrightarrow{?coffee}$ 4 $\xrightarrow{!coffee}$ 1 (bal=3) $\xrightarrow{?reset}$ 2 $\xrightarrow{?confirm}$ 3 $\xrightarrow{!change(c=0)}$
   - The test fails since there was balance that was not returned. The shrink step is successful.
4. ?coin(v=1), ?coin(v=2), ~~?coffee, ?reset~~, ?cancel, ?coin(v=2), ?coffee, ?reset, ?confirm
   - 1 $\xrightarrow{?coin(v=1)}$ 1 (bal=1) $\xrightarrow{?coin(v=2)}$ 1 (bal=3) $\xrightarrow{?cancel}$
   - The test stops after the ?cancel input, since there is no ?cancel input available in location 1. The shrink step is unsuccessful.

The shrinking algorithm is not done, it will continue to remove single inputs as above, until all inputs have been considered for removal.

6.4.2 Binary Elimination Shrinking algorithm

Binary shrinking tries to remove larger sections from the list of inputs at the time. The Binary Elimination Shrinking algorithm will always add add the last input from the list of inputs to the inputs that are being tested. This means the last ?confirm input will never be removed. The list of inputs also shows how the algorithm has currently divided the trace by means of square brackets.

1. [[~~?coin(v=1), ?coin(v=2), ?coffee, ?reset, ?cancel, ?coin(v=2), ?coffee, ?reset~~]], ?confirm
   - 1 $\xrightarrow{?confirm}$
   - The Binary Elimination Shrinking algorithm always starts by trying just the final input. The ?confirm input is not available in state 1. The shrink step is unsuccessful.
2. [[~~?coin(v=1), ?coin(v=2), ?coffee, ?reset~~], [?cancel, ?coin(v=2), ?coffee, ?reset]], ?confirm
   - 1 $\xrightarrow{?cancel}$
   - The algorithm tries to remove the first half of the list of inputs. The test stops after the ?cancel input, since there is no ?cancel input available in location 1. The shrink step was unsuccessful.
3. [[~~?coin(v=1), ?coin(v=2)~~], [?coffee, ?reset], [?cancel, ?coin(v=2), ?coffee, ?reset], ?confirm
   - 1 $\xrightarrow{?coffee}$
   - The last step failed, now the first quarter of the list of inputs is removed. The test stops after the ?coffee input, since there is no ?coffee input available at location 1 with a balance of 0. The shrink step in unsuccessful.
4. [[~~?coin(v=1)~~], [?coin(v=2)], [?coffee, ?reset], [?cancel, ?coin(v=2), ?coffee, ?reset], ?confirm
   - 1 $\xrightarrow{?coin(v=2)}$ 1 (bal=2) $\xrightarrow{?coffee}$ 4 $\xrightarrow{!coffee}$ 1 (bal=0) $\xrightarrow{?reset}$ 2 $\xrightarrow{?cancel}$ 1 $\xrightarrow{?coin(v=2)}$ 1 (bal=2) $\xrightarrow{?coffee}$ 4 $\xrightarrow{!coffee}$ 1 (bal=0) $\xrightarrow{?reset}$ 2 $\xrightarrow{?confirm}$ 3 $\xrightarrow{!change(c=0)}$ 1
   - The last step failed, only the first input is removed. This test passes because there is no remaining balance to return. The shrink step was unsuccessful.
5. [[?coin(v=1)], [~~?coin(v=2)~~], [?coffee, ?reset], [?cancel, ?coin(v=2), ?coffee, ?reset], ?confirm
   - 1 $\xrightarrow{?coin(v=1)}$ 1 (bal=1) $\xrightarrow{?coffee}$ 1
   - The last shrink step failed, the list cannot be further split, so the algorithm will move on to the next part of the list of inputs, and remove this. The test stops after the ?coffee input, since there is no ?coffee input available at location 1 with a balance of 1. The shrink step is unsuccessful.

Again, the shrinking algorithm is not done yet. The next step would be to remove the [?coffee, ?reset] inputs from the list of inputs, and run a test again.

6.4.3 Delta Debugging Shrinking algorithm

The Delta Debugging Shrinking algorithm is based on the ddmin algorithm introduced in [1]. Like the Binary Elimination Shrinking algorithm, it tries to remove larger parts of the list of inputs at the same time. Just like in the Binary Elimination Shrinking algorithm, the last input ?confirm is always added to the list of inputs to test.

1. [[~~?coin(v=1), ?coin(v=2), ?coffee, ?reset~~], [?cancel, ?coin(v=2), ?coffee, ?reset]], ?confirm
   - 1 $\xrightarrow{?cancel}$
   - The Delta Debugging Shrinking algorithm will start by splitting the test into two equal subsets, and removing the first one. The test cannot continue since the ?cancel input is not available in location 1. The shrink step is unsuccessful.
2. [[?coin(v=1), ?coin(v=2), ?coffee, ?reset], [~~?cancel, ?coin(v=2), ?coffee, ?reset~~]], ?confirm
   - 1 $\xrightarrow{?coin(v=1)}$ 1 (bal=1) $\xrightarrow{?coin(v=2)}$ 1 (bal=3) $\xrightarrow{?coffee}$ 4 $\xrightarrow{!coffee}$ 1 (bal=1) $\xrightarrow{?reset}$ 2 $\xrightarrow{?confirm}$ 3 $\xrightarrow{!change(c=0)}$
   - Next, delta debugging tries to remove the second subset. The new test fails because a coffee was ordered and the remaining balance was not returned. The shrink step was successful.
3. [[~~?coin(v=1), ?coin(v=2)~~], [?coffee, ?reset], [~~?cancel, ?coin(v=2), ?coffee, ?reset~~]], ?confirm
   - 1 $\xrightarrow{coffee}$
   - Since the previous test failed, the algorithm will do a 'reduce to subset' step, it will continue with just the subset that made the last test fail, and start splitting that into new subsets. Of the new subset, the first one is removed again. The test cannot continue since the ?coffee input is not available in location 1 with balance 0. The shrink step is unsuccessful.

The Delta Debugging Shrinking algorithm will continue by trying to remove the next subset [?coffee, ?reset], this will also be unsuccessful. Afterwards the algorithm will do an 'increase granularity' step, this will leave four subsets ([[?coin(v=1)], [?coin(v=2], [?coffee], [?reset]]) on which the algorithm will continue.

6.4.4 Cycle Shrinking algorithm

The Cycle shrinking algorithm is relatively straight forward. It searches for cycles within the failing test case. A cycle is a sub trace after which the location within the model, as well as the valuation is the same. The algorithm will try to remove the cycles from longest to shortest, until there are no cycles left to remove.

1. ?coin(v=1), ~~?coin(v=2), ?coffee, ?reset, ?cancel, ?coin(v=2), ?coffee~~ ?reset, ?confirm
   - 1 $\xrightarrow{?coin(v=1)}$ 1 (bal = 1) $\xrightarrow{?reset}$ 2 $\xrightarrow{?confirm}$ 3 $\xrightarrow{!change(c=1)}$ 1
   - The Cycle Shrinking algorithm tries to remove the largest cycle first, this is from step 3 until 10 in Table 6.1, since after both the valuation and location before step 3 is the same as after step 10. The test passes since the correct balance is returned. The shrink step was unsuccessful.
2. ?coin(v=1), ~~?coin(v=2), ?coffee, ?reset, ?cancel~~, ?coin(v=2), ?coffee ?reset, ?confirm
   - 1 $\xrightarrow{?coin(v=1)}$ 1 (bal=1) $\xrightarrow{?coin(v=2)}$ 1 (bal=3) $\xrightarrow{?coffee}$ 4 $\xrightarrow{!coffee}$ 1 (bal=1) $\xrightarrow{?reset}$ 2 $\xrightarrow{?confirm}$ 3 $\xrightarrow{!change(c=1)}$

- Since the largest cycle can't be removed, the algorithm will move on to the second largest cycle, this is from step 3 to 7. The test fails, since the remaining balance was not returned. The shrink step was successful. This is also the final step in this process, since there are no more cycles that can be removed.

6.4.5 Location Cycle Shrinking algorithm

The Location Cycle Shrinking algorithm is similar to the Cycle Shrinking algorithm above. Instead of looking for cycles by looking at both the location and the valuation, the Location Cycle Shrinking algorithm will only look at the location to determine cycles. These cycles will again be removed from longest to shortest.

1. ~~?coin(v=1), ?coin(v=2), ?coffee, ?reset, ?cancel, ?coin(v=2), ?coffee,~~ ?reset, ?confirm
   - 1 $\xrightarrow{?reset}$, 2 $\xrightarrow{?confirm}$ 3 $\xrightarrow{!change(c=0)}$ 1
   - The Location Cycle Shrinking algorithm tries to remove the largest location cycle first, this is from step 2 to 10 in Table 6.1, since the location before step 2 is the same as after step 10. The test passes since there's no balance to be returned. The shrink step was unsuccessful.
2. ?coin(v=1), ~~?coin(v=2), ?coffee, ?reset, ?cancel, ?coin(v=2), ?coffee,~~ ?reset, ?confirm
   - 1 $\xrightarrow{?coin(v=1)}$ 1 (bal=1) $\xrightarrow{?reset}$, 2 $\xrightarrow{?confirm}$ 3 $\xrightarrow{!change(c=1)}$ 1
   - Since the largest cycle can't be removed, the algorithm will move on to the second largest location cycle, this is from step 3 to 10. The test passes, the shrink step was unsuccessful.
3. ~~?coin(v=1), ?coin(v=2), ?coffee, ?reset, ?cancel,~~ ?coin(v=2), ?coffee ?reset, ?confirm
   - 1 $\xrightarrow{?coin(v=2)}$ 1 (bal=2) $\xrightarrow{?coffee}$, 4 $\xrightarrow{!coffee}$ 1 (bal=0) $\xrightarrow{?reset}$, 2 $\xrightarrow{?confirm}$ 3 $\xrightarrow{!change(c=0)}$ 1
   - The algorithm will move on to the third largest location cycle, this is from step 2 to 8. The test passes since there was no balance to return. The shrink step was unsuccessful.

The algorithm will keep trying to remove location cycles, from longest to shortest, until all cycles that can be removed are removed.

7 RESULTS AND DISCUSSION

This chapter will show and discuss the results of the two experiments. The experiments show that the two systems have different characteristics when it comes to shrinking failing test cases. On the vending machine, the Cycle Shrinking algorithm was much more effective than on the real-world system PPR-AAP. In contrast, the shrinking algorithm based on Delta Debugging was surprisingly effective on the real-world system, but in most cases not as effective as the Location Cycle Shrinking algorithm.

This chapter will first show results of the first experiment in Section 7.1 and second experiment in Section 7.2 by showing averages for the metrics described in Section 5.4: Shrinking percentage, number of interactions with the SUT during the shrinking process, the number of test cases executed during the shrinking process and shrinking time.

The discussion in Section 7.3 will give an answer to the sub questions, while discussing the results in more detail. At the end of this chapter the research question *"What method can be used to shrink traces of failing tests in test cases derived from Symbolic Transition Systems?"* will be answered. In short, the algorithms that used the model, the Cycle Shrinking algorithm and the Location Cycle Shrinking algorithm performed best. To minimise the final length of the test cases, these algorithms can be followed up by the Delta Debugging Shrinking algorithm.

The raw data from the experiments can be found on GitHub[1].

7.1 Results Experiment 1: Vending Machine

The first experiment was done on a simple vending machine. This vending machine was also used by Koopman et al. [2]. They included 10 mutants, that each contain exactly one bug. For each mutant 10 failing test cases were generated and shrunk by the different algorithms. A summary of the data presented in this section can be found in Section 7.1.1 at the end of this section.

We begin by looking at the average shrinking percentage for each algorithm. The shrinking percentage is the percentage that traces after shrinking are shorter on average than the traces before shrinking, as a percentage of the length of the initially found traces. This gives an indication of how effective the shrinking algorithms are in reducing the size of the traces.

As an example, if there are two traces of failing tests, one with an initial length of 30 and a final length of 5 and one with an initial length of 50 and a final length of 7, the average shrinking percentage would be $\frac{(30+50)-(5+7)}{30+50} \cdot 100 = 85\%$.

In all bar plots the shrinking algorithms are shown on the horizontal axis using an abbreviated name. The name 'cycle' means the Cycle Shrinking algorithm. 'Cycle, Delta' means the Cycle Shrinking algorithm was used first, followed by the Delta Debugging Shrinking algorithm.

[1] `https://github.com/Lars-Meijer/sts-shrinking-results/`

The plot in Figure 7.1 shows the shrinking percentages for the different algorithms in the vending machine experiment. It shows that on average, the shrinkers have a very similar effectiveness, except the Cycle Shrinking algorithm. The shrinking algorithms have a shrinking percentage of 81.5% to 81.0%. The Cycle Shrinking algorithm has an average shrinking percentage of 73.0%.

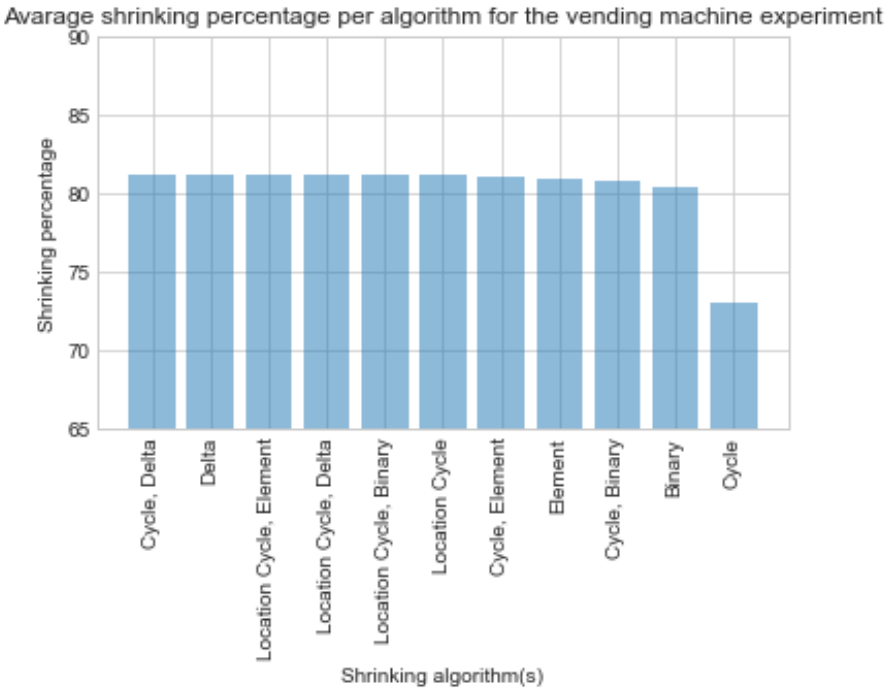

Figure 7.1: The percentage each shrinking algorithm shrunk a failing test case on average for the experiment on the vending machine.

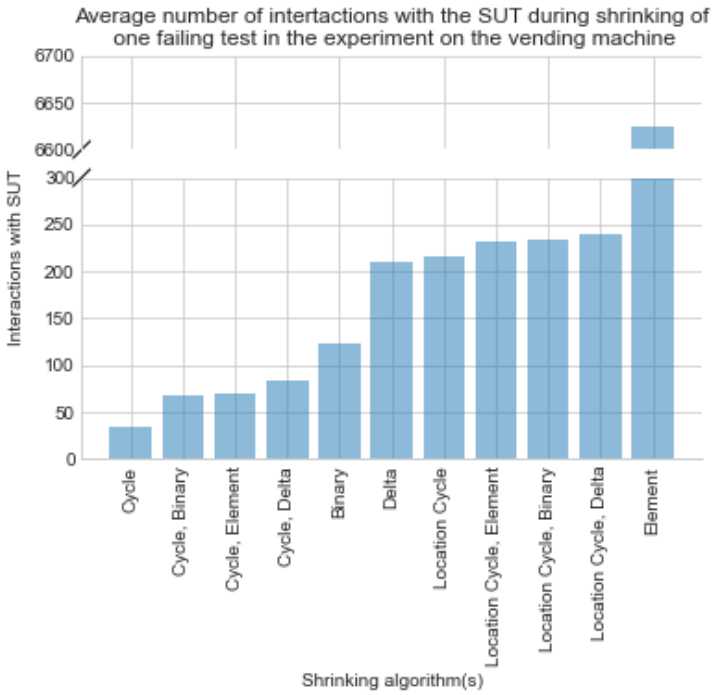

Figure 7.2: Number of interactions with the SUT a shrinking algorithm needs on average during the vending machine experiment.

Next we will look at the the number of interactions with the SUT each shrinking algorithm needs on average, to shrink a single failing test case. An interaction is one input and all following output. This is shown in Figure 7.2.

This plot shows clearly that the Element Elimination Shrinking algorithm uses a very high number of interactions with the SUT, compared to all other algorithms. This is the main reason it was not taken into account in the second experiment. The Cycle Shrinking algorithm uses the fewest interactions with the SUT. Combinations that start with the Cycle Shrinking algorithm also use fewer interactions than the other algorithms. Making them good candidates for shrinking test cases.

Now we will look at the number of test cases that was needed to shrink a single test case on average. This is shown in Figure 7.3.

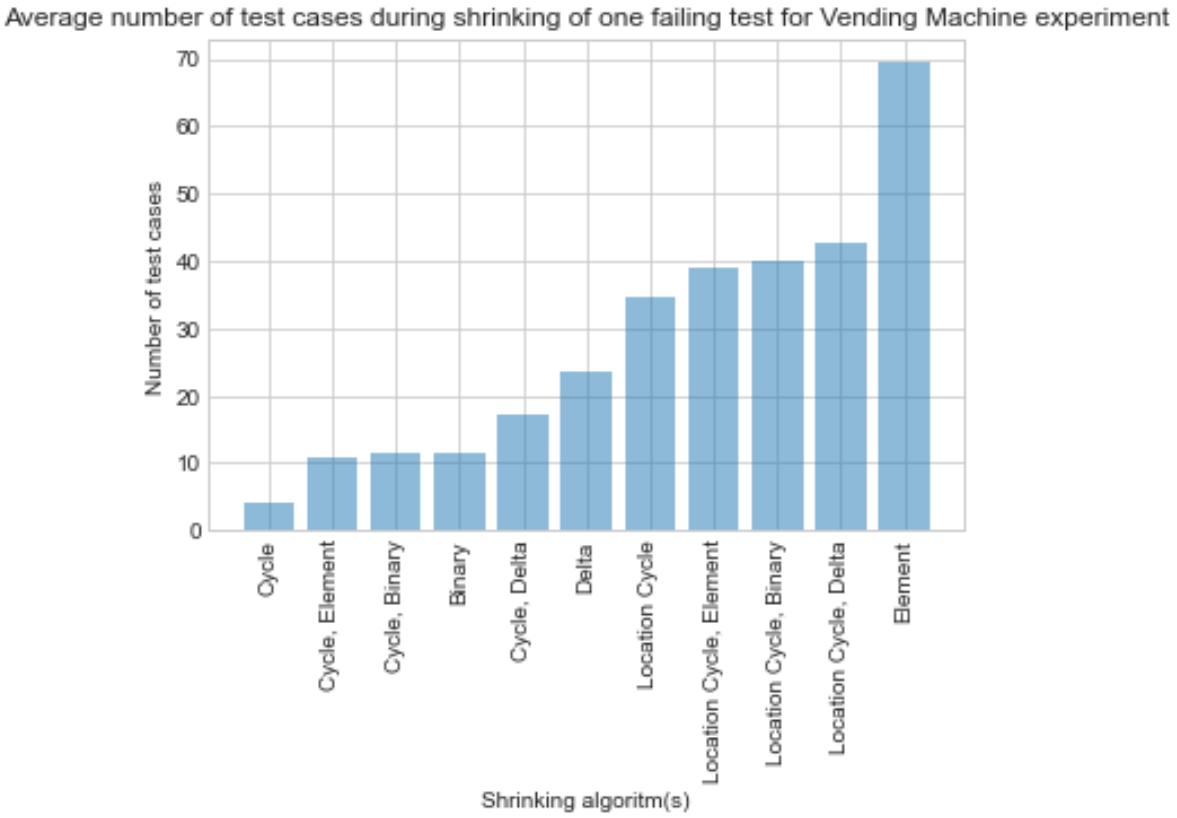

Figure 7.3: The number of test cases needed to shrink a single failing test on average during the experiment on the vending machine.

By comparing this plot to the plot in Figure 7.2, we can see a correlation between the number of test cases needed to shrink a test case, and the total number of interactions with the SUT that is needed to shrink a failing test. The shrinking algorithms that had the highest number of interactions with the SUT, also have the largest number of test cases. In fact, the order of the shrinking algorithms is the exact same.

Finally, we look at the shrinking time. The plot in Figure 7.4 shows the time that it took on average to shrink a single test case.

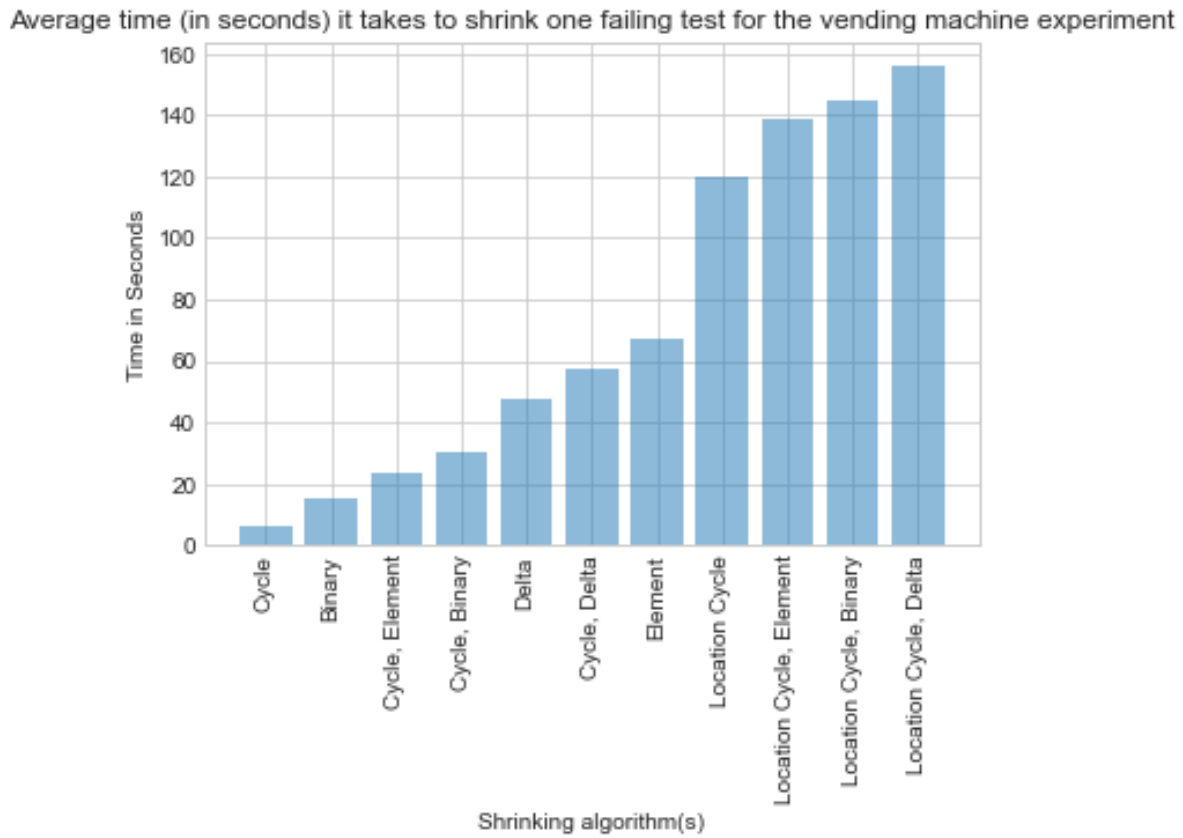

Figure 7.4: The average time it took to shrink a single test case using each method during the experiment on the vending machine.

In a real-world system we expect the interactions with the SUT to be the most time consuming part of a test case. Koopman et al. made the following assumption: “Testing an actual transition is more time-consuming than some simple calculations to select the next transition to be tested” [2]. This is also the experience of the experts at Axini.

Real-world systems are often complex, operations in such systems are non-trivial. Additionally, testing on a real-world system uses a form of inter-process communication to interact with the SUT. Often this takes on the form of middleware, such as a message queue over a network. This adds additional delay to each interaction with the SUT.

In the experiment on the vending machine, there is no clear correlation between the time it takes to shrink a test case and the number of interactions with the SUT. The vending machine was directly integrated into AMP, there was only in-process communication, in the form of method calls. Additionally, the operations of the vending machine are very simple, such as adding one or two to the balance of the vending machine.

In the next experiment we do clearly see a correlation between the number of interactions with the SUT in Figure 7.6 and time in Figure 7.8. Based on the next experiment and the information above, we assume that the number of interactions with the SUT is a more fair metric than time to measure the effort it takes to shrink a test case.

7.1.1 Experiment Summary

The following table shows an overview of the data from the experiment on the vending machine, presented in this section. These are the same numbers as seen in the plots contained in this

section. For each shrinking algorithm it shows the average shrinking percentage, the average number of interactions with the SUT, the average number of test cases and the time that was needed to shrink a single failing test case.

	Element	Binary	Delta	Cycle	Location Cycle	Cycle, Element	Cycle, Binary	Cycle, Delta	Location Cycle, Element	Location Cycle, Binary	Location Cycle, Delta
Percentage	80.9%	80.4%	81.2%	73.0%	81.2%	81.1%	80.8%	81.2%	81.2%	81.2%	81.2%
Nr. interactions	6624	123	210	35	215	70	69	84	231	234	240
Nr. test cases	70	12	24	4	35	11	12	17	39	40	43
Time	67s	15s	50s	6s	120s	24s	30s	57s	138s	144s	155s

Table 7.1: An overview of the data presented in this section.

7.2 Experiment 2: PPR-AAP

The second experiment was on PPR-AAP, a real world system used by ProRail to manage parts of the Dutch rail network. PPR-AAP can be used to subscribe to the train schedule and receive updates in real time. The model for this system contains around 150 locations, and it has about 10 location variables.

For this system fifteen mutants with one bug each were created. To do this two experts created a list of common bugs in similar systems, similar bugs were introduced in the mutants and then verified to be realistic by these experts. The full process is explained in section 5.2. For each of the fifteen bugs, five failing test cases were generated and shrunk. We will look at the same plots as shown for the experiment on the vending machine shown in the previous section.

Again, we will start with the the shrinking percentage, this is shown in Figure 7.5.

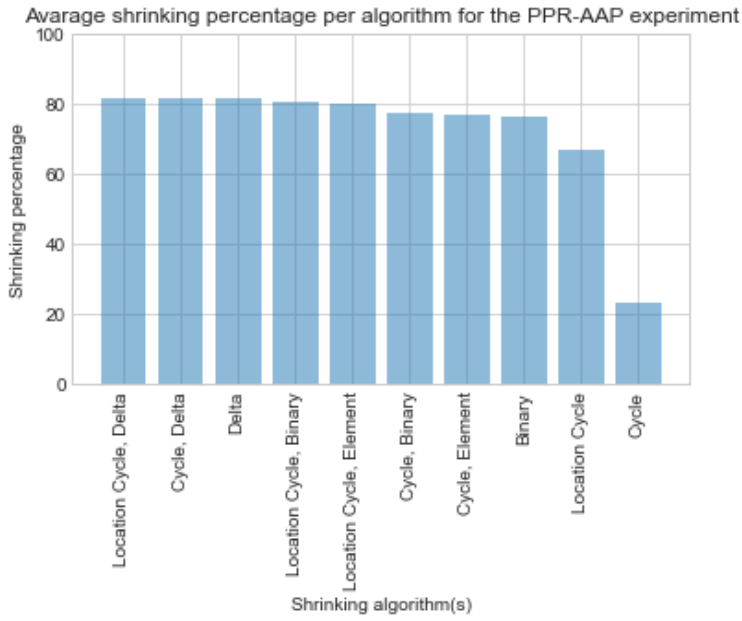

Figure 7.5: The percentage that each shrinking algorithm shrunk the failing traces on average for the PPR-AAP experiment.

We see that the effectiveness on PPR-AAP is somewhat lower than on the vending machine, but overall the shrinking algorithms remained effective. Contrary to the previous experiment, the Location Cycle Shrinking algorithm is less effective than the other algorithms. The normal Cycle Shrinking algorithm has also lost most of it's effectiveness, managing to only remove about 25% of the original failing traces. This is because the algorithm finds very few cycles and the cycles that are found are generally fairly short. Section 7.3.2 will go into more depth on the performance of the Cycle Shrinking algorithms on PPR-AAP.

Next, we will look at the number of interactions with the SUT that were needed to shrink a single failing test case for PPR-AAP on average. This is shown in figure 7.6.

We can see that delta debugging shrinking algorithm has surprisingly few interactions with the SUT while shrinking bugs in this system.

It is clear that Cycle Shrinking algorithm, followed by element shrinking algorithm is by far the least efficient while shrinking. As Figure 7.5 shows, the cycle shrinking algorithm by itself is not very effective, this means that there's quite a lot of work still to be done by the element shrinking algorithm.

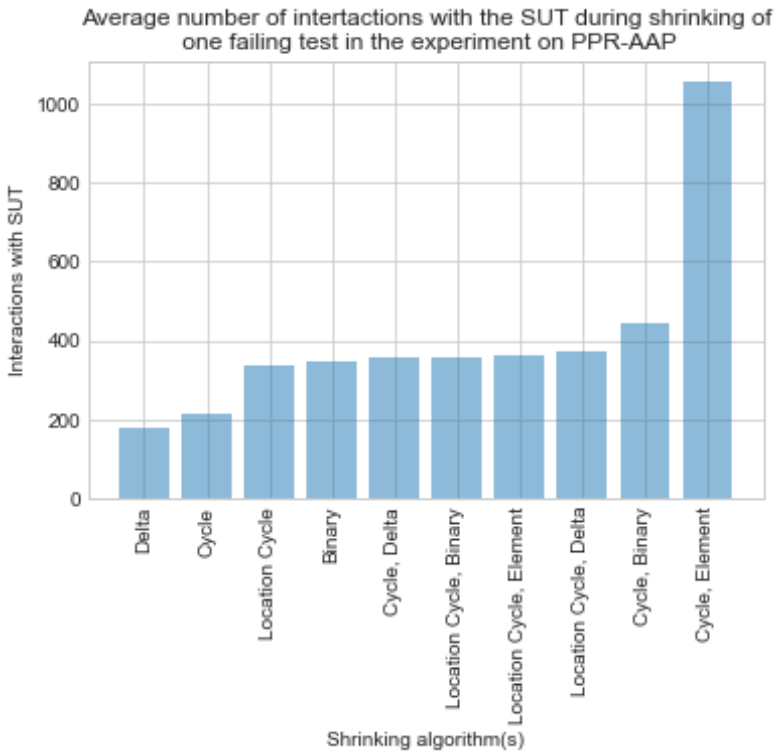

Figure 7.6: The number of interactions with the SUT that was needed on average to shrink a single failing test case during the experiment on PPR-AAP.

Figure 7.7 shows the average number of test cases that was needed to shrink a single failing test case on average during the second experiment.

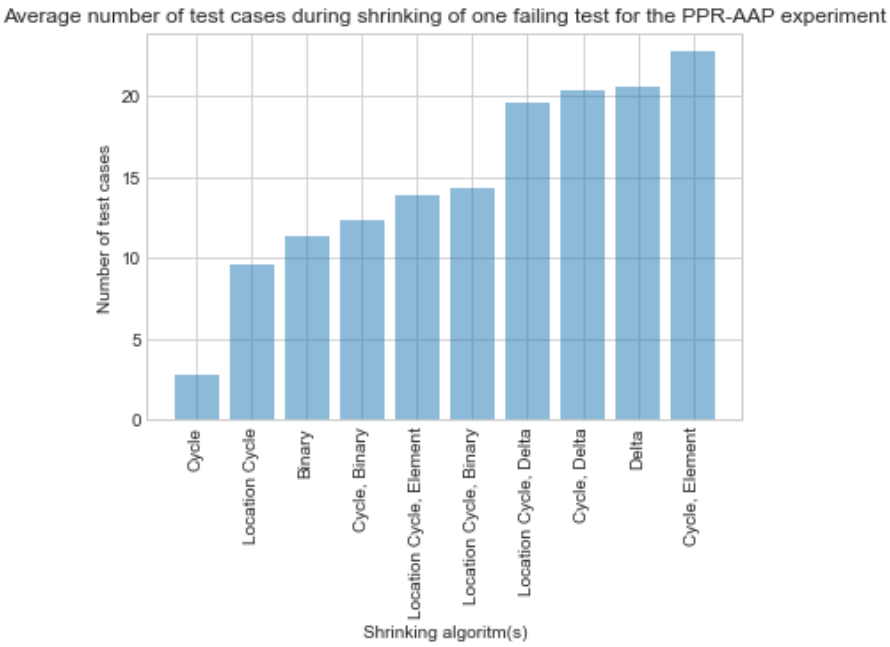

Figure 7.7: The number of executed tests-cases with the that was needed on average to shrink a single failing test case during the experiment on PPR-AAP.

The graph in Figure 7.7 shows that the Cycle Shrinking algorithm used very few test cases. Because the Cycle Shrinking algorithm finds few cycles, it also does few test cases to try and remove inputs from the traces. Compared to the other algorithms the Location cycle Shrinking algorithm used fewer test cases than in the previous experiment.

Finally, we look at the time it takes to shrink a single failing test case using each of the shrinking algorithms.

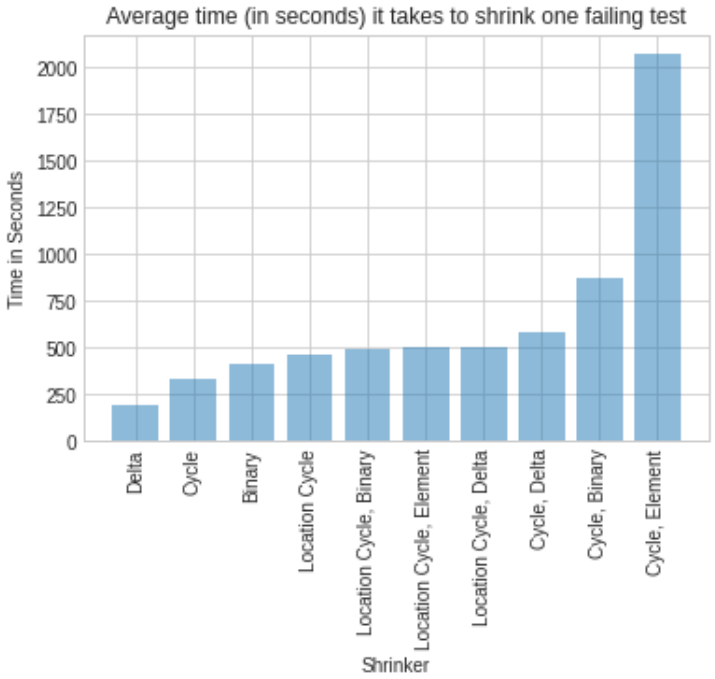

Figure 7.8: Average time in seconds that it took on average to shrink a single failing test case during the experiment on PPR-AAP.

If the graph in Figure 7.8 is compared to the graph of the number of interactions with the SUT during the shrinking process in Figure 7.6, it becomes clear that the more interactions an algorithm has with the SUT, the longer it takes to shrink a test case.

7.2.1 Experiment Summary

The following table shows an overview of the data from the experiment on PPR-AAP, presented in this section. These are the same numbers as seen in the plots contained in this section. For each shrinking algorithm it shows the average shrinking percentage, the average number of interactions with the SUT, the average number of test cases and the time that was needed to shrink a single failing test case.

	Binary	Delta	Cycle	Location Cycle	Cycle, Element	Cycle, Binary	Cycle, Delta	Location Cycle, Element	Location Cycle, Binary	Location Cycle, Delta
Percentage	76.3%	81.4%	23.1%	66.7%	76.9%	77.3%	81.7%	80.1%	80.5%	81.8%
Average nr. of interactions	349	178	216	337	1055	444	358	363	359	372
Average nr of test cases	11	21	3	10	23	12	20	14	14	20
Average time to shrink	420s	192s	335s	468s	2078s	873s	581s	500s	498s	505s

Table 7.2: An overview of the data presented in this section.

7.3 Discussion

7.3.1 Koopman's Algorithms on STSs

First, we will compare the results of our experiment, with the results from Koopman et al. [2]. This will answer the first sub question: "Can the results of Koopmans et al. be reproduced on Symbolic Transition Systems?".

We tried to reproduce the results by Koopman et al. [2] on test cases derived from STSs. In general however, AMP found shorter traces than GAST, the tool Koopman et al. used. This means that the number of steps needed to shrink a single test case cannot be directly compared. Since a shorter start trace requires fewer interactions to shrink.

It is however to be expected that the final trace length after shrinking is similar. Table 7.3 shows the average size of the traces after shrinking (measured as the number of inputs in the test case since in ESMs the inputs and outputs are on the same transition). We do indeed see that the results are very similar.

Algorithm		Bug 1	Bug 2	Bug 3	Bug 4	Bug 5	Bug 6	Bug 7	Bug 8	Bug 9	Bug 10
Element	Koopman	7	6	2	5	6	4	4	2	2	7
	STS	5.8	5.8	2.3	4.8	6.0	4.1	3.7	2.0	2.0	6.4
Binary	Koopman	7	6	3	5	6	4	4	2	2	7
	STS	6.1	6.6	2.6	5.0	6.0	4.1	3.7	2.0	2.0	6.5
Cycle	Koopman	13	14	9	10	14	12	10	7	4	13
	STS	11.6	13.6	5.4	7.1	12.1	6.3	6.3	3.3	3.6	9.2
Cycle + Element	Koopman	7	6	2	5	5	4	3	2	2	7
	STS	6.0	5.8	2.2	4.7	4.9	4.0	3.7	2.0	2.0	6.6
Cycle + Binary	Koopman	7	6	2	5	5	4	3	2	2	7
	STS	6.0	6.0	2.3	4.7	4.8	4.0	3.7	2.0	2.0	6.6

Table 7.3: A comparison of the results from Koopman et al. in [2] and the results obtained in the first experiment based on the length of traces after the shrinking process. The table shows the average length of the traces after shrinking for each mutant for both experiments.

In Table 7.3, we do indeed see that the results from the first experiment and the experiment by Koopman et al. [2] are very similar. The largest difference are the results of the cycle shrinking algorithm. It seems that the implementation of the cycle shrinking algorithm for STSs performs significantly better here. This can however be explained by the fact that AMP found shorter failing traces on which the shrinking process was done. In general, only the insertion of coins between the final reset and the occurrence of the bug cannot be removed, since these transitions change the state. A longer starting trace means there are more cycles that cannot be removed.

The first experiment shows that it is indeed possible to apply the algorithms of Koopman et al. to STSs, at least in some cases. Experiment two shows that the algorithms can also be applied to real-world systems. Of course, the limitations that apply to the original algorithms still apply: The algorithms only work on SUTs that behave deterministically.

During the first experiment two new algorithms were also tested: The Location Cycle Shrinking algorithm and the Delta Debugging Shrinking algorithm. We will discuss these here, and compare the results of the results of these algorithms to the algorithms by Koopman et al.

Of particular interest is the comparison between the Delta Debugging Shrinking algorithm and the Binary Elimination Shrinking algorithm, and the comparison between the Cycle Shrinking algorithm and Location Cycle Shrinking algorithm since these pairs of algorithms are quite similar to each other.

Binary Elimination versus Delta Debugging

Both the Binary Elimination Shrinking algorithm and and the Delta Debugging Shrinking algorithm are algorithms that try to minimise a test by a sort of binary search for the shortest test.

Figure 7.1 shows that Cycle Shrink followed by the Delta Debugging Shrinking algorithm have the highest average shrinking percentage. The Delta Debugging Shrinking algorithm with an average shrinking percentage of 81.5% scores marginally better than the Binary Elimination Shrinking algorithm with a shrinking percentage of 81.3%.

The number of interactions with the SUT for Binary Elimination Shrinking algorithm is lower than for the Delta Debugging Shrinking algorithm as shown in Figure 7.2. This means that on

a system where interactions cost take more time than in this synthetic experiment, the Binary Elimination Shrinking will be faster.

In general, it seems that the Binary Elimination Shrinking algorithm is a better choice to shrink traces than the Delta Debugging Shrinking algorithm in the vending machine experiment.

Cycle shrinking versus Location Cycle Shrinking

A cycle is a sub trace of the complete trace of the failing test where the model is in the same state before and after executing this sub trace. Both the location and the valuation of the variables should be the same for a sub-trace to be considered a cycle. The Cycle Shrinking algorithm tries to remove such cycles from the trace as explained in Section 4.1.

A location cycle is, similar to a normal cycle. For the Location Cycle Shrinking algorithm we do however not care about the valuation of the variables. In other words, we only care about the location being the same.

The Location Cycle Shrinking algorithm finds all cycles, including the ones where the valuation is the same that would be found by the normal Cycle Shrinking algorithm. Because of this, it is expected that the Location Cycle Shrinking algorithm is more effective in shrinking than the normal Cycle Shrinking algorithm. Figure 7.5 confirms this.

In fact, the Location Cycle Shrinking algorithm performs just as well as the other shrinking algorithms. This is because the model is so simple, after each input the it returns to the initial location immediately, or it does a single output, after which it will return to the initial location. This means that every input will make a Location Cycle.

If we look at the number of interactions with the SUT in Figure 7.2, the cycle shrinking algorithm is very efficient. In fact, the Cycle Shrinking algorithm has less than a quarter of the number of interactions on average than the next best single algorithm, the Binary Elimination Shrinking algorithm. In general, the Cycle Shrinking algorithm managed to remove the largest or second largest cycle it found without issue, which makes it very efficient.

The Location Cycle Shrinking algorithm uses significantly more interactions with the SUT, using around the same as the Delta Debugging Shrinking algorithm.

Because cycle shrinking is very efficient, it makes a good candidate for combining it with other shrinkers. The combinations of Cycle Shrinking with Element, Binary and Delta Debugging Shrinking algorithms all have less interactions with the SUT than all other algorithms and combinations thereof. These three combinations also have almost identical shrinking percentages.

Conclusion

In the experiment on the vending machine the Cycle Shrinking algorithm, followed by the Binary Elimination Shrinking algorithm is the best choice to shrink failing test cases. It gives the shortest traces on average, while it performs very few interactions with the SUT, second only to just the Cycle Shrinking algorithm.

On a similar system, where interactions with the SUT are very expensive, just the Cycle Shrinking algorithm results in reasonably short traces after shrinking, using the fewest interactions with the SUT.

7.3.2 Algorithms on a real-world system

The second experiment looks at the performance of the shrinking algorithms on the real-world system PPR-AAP. Section 7.2 gave an overview of the results of this experiment. In this section we will discuss the results of this experiment in more depth, and answer the sub-question *"How do the algorithms by Koopman et al. [2] perform when they are algorithms applied to real-world systems?"*.

First, we can make the observation that most algorithms were fairly effective on the real-world system PPR-AAP. Figure 7.5 shows that most shrinking algorithms have a shrinking percentage of just below 80%. The exception to this is the cycle shrinking algorithm, which only has a shrinking percentage of 25%.

In PPR-AAP we also see for the first time that different algorithms have significantly different performance on different bugs. Figure 7.9 shows per bug how many interactions with the SUT were needed on average to shrink a single failing test per shrinking algorithm.

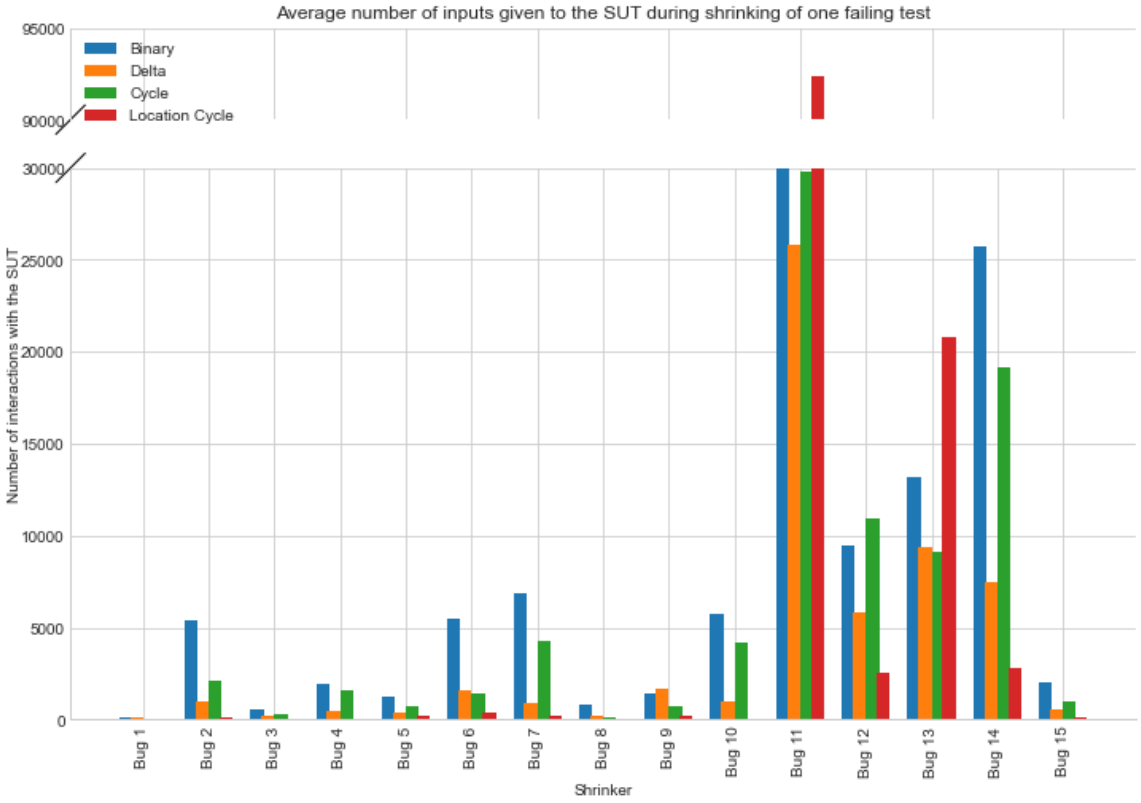

Figure 7.9: The number of interactions with the SUT that was needed on average to shrink a single failing test case on PPR-AAP, for each bug that was introduced.

We see that, for almost every bug, the Location Cycle Shrinking algorithm has the fewest interactions with the SUT. But for Bug 11 and Bug 13 this algorithm had more interactions with the SUT than all other algorithms.

Bug 11 and 13 are both bugs where the cycle is part of the bug, this means that a certain state needs to be reach n times before the bug will trigger.

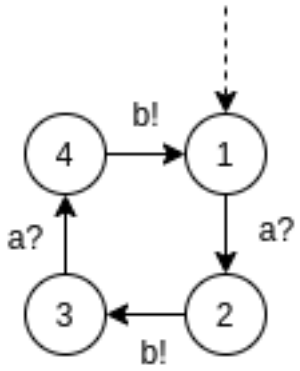

Figure 7.10: A simple LTS with a cycle.

Consider the simple LTS in Figure 7.10, imagine a bug in state 2 triggers after 4 cycles, and the output is incorrect. This means there is a location cycle from the first time state 1 was visited, to every other time location 1 was visited. This is also the case for all other locations. Removing any of these cycles would not cause the bug to trigger, so the shrinking fails. This means that the Location Cycle Shrink algorithm will try to remove a lot of cycles, to no avail. Thus the algorithm has to do a lot of interactions with the SUT, which explains the high number of interactions for the Location Cycle Shrinking algorithm for bug 11 and 13.

The next section will compare Delta Debugging with Binary Elimination. Section 7.3.2 will compare the Cycle Shrinking algorithm with the Location Cycle Shrinking algorithm.

Delta Debugging and Binary Elimination

The experiment on PPR-AAP shows that Delta Debugging and combinations of shrinkers with Delta Debugging creates the shortest traces after the shrinking process. Figure 7.5 shows that the differences between the algorithms are more pronounced than in the first experiment.

Figure 7.9 shows that delta debugging is has fewer interactions with the SUT than binary elimination. In the first experiment, binary elimination had fewer interactions with the SUT on average. This difference can be explained by the properties of the shrinking algorithm and the SUT.

If the Binary Elimination Shrinking algorithm doesn't manage to remove a certain part of the input, it will split the part of the input it was trying to remove in two parts, and it will try to remove the first halve again, until it tries to remove just one input.

In contrast, if the Delta Debugging Shrinking algorithm doesn't manage to eliminate a certain part of the input, it will move on to the next part and try that, until all parts have been tried, at which point all parts will be split, and the smaller parts, and their complements will be tested one by one.

The result is that Delta Debugging Shrinking algorithm ends up doing more tests, but those tests are on average shorter, because larger parts of the inputs are removed sooner. The fact that Delta Debugging uses more test cases was also shown in Figure 7.7.

In PPR-AAP almost any test case has to start by setting up a connection, the first step of the test case can not be removed. After establishing a connection, at some point a valid Protocol Request has to be made, to establish a communication protocol. This is another transition that cannot be removed from the trace. Single non-removable inputs like these are unfavourable for the Binary Elimination Shrinking algorithm, while the Delta Debugging Shrinking algorithm manages to deal with them better.

Cycle shrinking and Location Cycle Shrinking

As in the previous experiment the Location Cycle Shrinking algorithm has a higher average shrinking percentage than plain Cycle Shrinking.

The shrinking effectiveness of the Cycle Shrinking algorithm on PPR-AAP is only 25%. It performs badly with regards to the average shrinking percentage when compared to the other algorithms in Figure 7.5. Furthermore, it is not even a particularly fast algorithm if we look at the number of interactions with the SUT in figure 7.6 and the time it takes to shrink an average test case in figure 7.8.

Compared to the vending machine from the first experiment, PPR-AAP is a relatively complex system. PPR-AAP uses variables to keep track of several attributes related to the system, such as the type of the last request and the time of this request. Moreover, unlike the vending machine, PPR-AAP does not have a trivial sequence that will return to the initial state, including valuation of variables.

This means that the Cycle Shrinking algorithm does not find many cycles. The cycles that are found are generally relatively short. Consider a trace that has 100 inputs, the cycle shrinking might find an cycle of six inputs, two of four inputs, and ten self loops. Even if all these cycles could be removed, it would still require testing a traces of 94 inputs, 90 inputs, 86 inputs and the 10 traces with the self loops removed. So even after removing these inputs, the traces that are tested are long.

In conclusion, the complexity of the system makes it unlikely for cycles long to be found, cycles that are found are generally short, which results in bad shrinking performance and a relatively long shrinking process for the Cycle Shrinking algorithm.

In most cases the Location Cycle Shrinking algorithm uses the fewest number of interactions with the SUT, as seen in Figure 7.9. It also results in shorter final traces than the Cycle Shrinking algorithm. But the resulting traces in general aren't as short as some of the other algorithms.

To shorten them even further, just like in the previous experiment, a combination of Location Cycle Shrink and another algorithm can be used to shrink the test cases even further. The Delta Debugging Shrinking algorithm is the best choice for this.

Conclusion

In general, the Location Cycle Shrinking, algorithm uses very few interactions with the SUT to shrink a test case. It does however not provide the shortest traces after shrinking. To get short traces in a relatively short time, the Location Cycle Shrinking algorithm can be followed up by the Delta Debugging Shrinking algorithm.

In some cases, the Location Cycle Shrinking algorithm does not perform well. Namely in cases where the cycle is part of the bug, so in cases where a certain action has to be performed multiple times before the bug triggers. When this is the case, Delta Debugging provides very short end traces, using relatively few interactions with the SUT.

If there is no information available about the kind of bug that caused a failing test case that needs to be shrunk, the Delta Debugging Shrinking algorithm can be used. The Delta Debugging Shrinking algorithm will result in short traces, while using relatively few interactions with the SUT to get to that result.

7.3.3 Modifications to existing algorithms

This section answers the sub question *"Can modifications be made to these three algorithms to get shorter traces or complete the shrinking process faster?"*.

As seen in the second experiment, the Cycle Shrinking algorithm was not as effective on failing test cases of PPR-AAP. Because PPR-AAP has a lot of location variables, only few cycles were found. Cycles that were found were generally short.

This lead to the introduction of the Location Cycle Shrinking algorithm. An algorithm that works similar to the regular Cycle Shrinking algorithm, except it detect cycles only based on locations, instead of locations and the valuations of the variables.

As expected, the Location Cycle Shrinking algorithm resulted in a higher shrinking percentage in both the first and second experiment. In the first experiment it used more transitions to shrink a trace than the regular Cycle Shrinking algorithm. While in the second experiment it used the fewest interactions with the SUT in most cases, while resulting in reasonably short traces.

There is one exception to this: If a cycle is part of the bug, so in cases where a certain action has to be performed multiple times before the bug triggers. If this is the case, the Location Cycle Shrinking algorithm will have more interactions with the SUT than other algorithms, and will be very slow.

It is likely that the normal Cycle Shrinking algorithm becomes less effective as the model and SUT become more complex. It is however also possible that the Location Cycle Shrinking algorithm might not be ideal for some systems. If a model has many strict guards on transitions for example, it could be possible that removing location cycles becomes less and less effective. A solution for this might be an option where the user selects which location variables should be taken into account during the shrinking process, and which should be ignored to have a middle ground between the two variants.

7.3.4 Effectiveness of Delta Debugging

In the two experiments the delta-debugging minimisation algorithm was tested to shrink traces. In this section the sub question "How effective is the delta-debugging minimisation algorithm for shrinking traces?" will be answered.

The effectiveness of the Delta Debugging Shrinking algorithm varied between the two experiments. In both experiments it resulted in a high shrinking percentage. In the first experiment on the vending machine, delta debugging used relatively many interactions with the SUT to achieve its result.

In the second experiment, the Delta Debugging Shrinking algorithm used the fewest interactions with the SUT in general. If we look at the individual bugs however, we see that in most cases the Location Cycle Shrinking algorithm used much fewer interactions with the SUT. This is because Location Cycle Shrinking performs very badly if executing cycle a certain number of times is the reason a bug triggers.

This makes the Delta Debugging Shrinking algorithm a good candidate if nothing is known about the type bug that caused the test to fail in the first place, or if it is known that some kind of cycle causes the failing test. Delta Debugging is also a good candidate to shrink the remainder of the test case after the Cycle Shrinking or Location Cycle Shrinking algorithm, since it generally resulted in the shortest traces.

7.3.5 Other shrinking algorithms

During the course of the research, the last sub question “What other techniques, such as shortest path algorithms, can be used to shrink traces and how effective are they?” was investigated, but there are no experimental results at this moment. This will be left as future work.

7.4 Research Question

This section will answer the research question originally laid out In Chapter 3: *“What method can be used to shrink traces of failing tests in test cases derived from Symbolic Transition Systems?”*.

The experiment shows that there is no single best shrinking algorithm. The Element Elimination algorithm is very simple to implement and produces short traces, but is very expensive to execute if we look at the number of interactions with the SUT.

The Binary Elimination Shrinking algorithm was just as effective in producing short traces in the vending machine experiment as the Delta Debugging Shrinking algorithm, while using fewer interactions with the SUT to do so. In the experiment on the real-world system, the Delta Debugging Shrinking algorithm produces the shortest traces. Delta Debugging also uses fewer interactions with the SUT than the Binary Elimination Shrinking algorithm to shrink traces on average.

However, the algorithms that used information from the model to shrink traces used significantly fewer interactions with the SUT than the other algorithms. During the experiment on the vending machine, the Cycle Shrinking algorithm performed the best in this regard. While during the experiment on PPR-AAP the Location Cycle Shrinking algorithm used the fewest number of interactions with the SUT, except in cases where a cycle was part of the reason of the bug occurring.

While the Cycle Shrinking algorithm in the first experiment and the Location Cycle Shrinking algorithm in experiment two, used the fewest interaction with the SUT, they had a lower shrinking percentage than the other algorithms. It makes sense to follow up these algorithms with either the Binary Elimination Shrinking algorithm or the Delta Debugging Shrinking algorithm. Which results in the shortest failing tests, while also using comparatively few steps.

In general, the best combination to shrink failing tests is either Cycle Shrinking followed up by Delta Debugging or Location Cycle Shrinking followed by Delta Debugging depending on the system. If nothing is known about the bug that might have caused a failing test in the first place, the Delta Debugging Shrinking algorithm might be a good choice. It produces short traces after shrinking, while, on average using few interactions with the SUT compared to other algorithms, especially in the experiment on the real world system.

8 CONCLUSION

Test case shrinking is the process of reducing the size of failing test cases to make them easier to analyse and debug. The shorter the test case the easier it is to debug. One way to achieve shorter failing test cases is by trying to reduce the size of a failing test case. This is called test case shrinking. Shrinking of failing test cases can be done using a shrinking algorithm.

This thesis looked into the applicability of shrinking algorithms on failing test cases derived from STS models. The thesis was centred around the research question:

"What method can be used to shrink traces of failing tests in test cases derived from Symbolic Transition Systems?"

To answer this question, five shrinking algorithms were evaluated evaluated. Three algorithms originally introduced by Koopman et al. in [2]: The Element Elimination shrinking algorithm, the Binary Elimination shrinking algorithm and the Cycle Shrinking algorithm. In this thesis we also introduced two algorithms that, as far as we know, have not yet been applied to state-based models such as ESMs and STSs. The Delta Debugging Shrinking algorithm is an adaptation of Zeller's Delta Debugging minimisation [1] algorithm. The Location Cycle shrinking algorithm is an adaptation of the Cycle Shrinking that only looks at the location, and not the valuation of the variables when searching for cycles.

These five algorithms and the combinations of Cycle Shrinking or Location Cycle Shrinking with another algorithm were evaluated in two experiments on two different systems. In the first experiment a simple vending machine was used. Each algorithm shrunk ten failing test cases for each of the ten bugs introduced in the vending machine.

So far the algorithms have only been tested on a small mock system, namely a simple vending machine. Thus the second experiment looked into the performance of all five algorithms on a real-world system called PPR-AAP. In this experiment each algorithm was tested on fifteen bugs, for each bug five failing test cases were generated and shrunk.

In the first experiment the Cycle Shrinking algorithm had a reasonable shrinking percentage, while using very few interactions with the SUT. This makes this algorithm a very good candidate for shrinking traces for this system. To maximise the shrinking percentage, the Cycle Shrinking algorithm can be followed up by another shrinking algorithm. Both the Binary Elimination shrinking algorithm and the Delta Debugging shrinking algorithm could be used for this.

The second experiment showed that Cycle Shrinking was not as effective on the real-world system. Because of the complexity of the system and number of variables, few cycles were found and the ones that were found were in general very short.

In this experiment, the Location Cycle shrinking algorithm proved much more efficient than the other algorithms. On average this algorithm used the fewest interactions with the SUT while resulting in relatively short test cases. There is one exception: Bugs where the a cycle was part of the bug. In this case the Location Cycle Shrinking algorithm did not perform well, it used

much more interactions with the SUT than all other algorithms. In cases like these, the Delta Debugging shrinking algorithm is the best choice to shrink failing test cases.

Because the Location Cycle Shrinking algorithm, just like the Cycle Shrinking algorithm in the first experiment did not produce the shortest traces, this algorithm can also be followed up by another shrinking algorithm. The best candidate for this is the Delta Debugging shrinking algorithm.

In conclusion, in some cases, namely those where a cycle is the reason a bug occurs, the Delta Debugging algorithm is the best choice to shrink failing test cases derived from symbolic transition systems. In all other cases, depending on the system, either the Cycle Shrinking algorithm or the Location Cycle Shrinking algorithm, followed by the Delta Debugging Shrinking algorithm is the best choice to shrink failing test cases.

9 FUTURE WORK

This thesis provided a lot of answers on the applicability and performance of shrinking algorithms on test cases derived from STSs. There are however several questions that remain.

First, how can these algorithms be used for non-deterministic SUTs? And by extension for bugs related to timing? It would be worthwhile to investigate if these algorithms could be adopted to work on non-deterministic systems.

If the same test case only fails one out of 100 times due to non-determinism, shrinking this test case would using the algorithms from this thesis would likely not be very effective. A solution could be to run the same test multiple time if it passes during the shrinking process. This would however not prove the test case never fails, each time it passes would only increase the confidence that it will not fail. Of course, having to do this for every test case that passes could be very expensive, depending on the number of times a passing test would be repeated.

The issue of non-determinism extends to timing. In practice transitions in an STS have a timeout. If an response is expected, it has to happen before the timeout expires. If a test case fails because the timeout expired, it could happen when the same test case is executed again, the output would be (just) in time. This can be seen as a form of non-determinism.

The sub-question *"What other techniques, such as shortest path algorithms, can be used to shrink traces and how effective are they?"* was left open.

The experiments showed that the two algorithms that used information from the model, namely the Cycle Shrinking algorithm and the Location Cycle shrinking algorithms needed the fewest interactions with the SUT while shrinking test cases. Instead of shrinking a failed test case, it might be possible to construct a new test case based on some hypothesis.

From a set of failing test cases it might be possible to construct a hypothesis about why this set of test cases fails. If there are for example 10 failing tests, it might for example be possible to form a hypothesis like: "The test always fails in location x", "The test always fails in state y", "Label $a?$ always fails", "Location x was always visited exactly 7 times" or "Every failing test case ends with the labels $a? \rightarrow x! \rightarrow b? \rightarrow y! \rightarrow c? \rightarrow fail!$".

It might be possible to create an algorithm that can construct test cases that verifies or rejects a hypothesis. For example, if the hypothesis is that a test case always fails at a certain location in the model, an obvious answer could be to try to use a shortest path algorithm to find a path to this location. However, in STSs transitions can have a transition restrictions. This makes the usage of normal shortest path algorithms infeasible. Instead, a path needs to be constructed such that it can reach the location of the transition that failed, while satisfying all transition restrictions.

REFERENCES

[1] A. Zeller and R. Hildebrandt, “Simplifying and isolating failure-inducing input,” *IEEE Transactions on Software Engineering*, vol. 28, p. 2002, 2002.

[2] P. Koopman, P. Achten, and R. Plasmeijer, “Model-based shrinking for state-based testing,” in *Revised Selected Papers of the 14th International Symposium on Trends in Functional Programming - Volume 8322*, ser. TFP 2013. Berlin, Heidelberg: Springer-Verlag, 2013, p. 107–124. [Online]. Available: https://doi.org/10.1007/978-3-642-45340-3_7

[3] R. Neumann, “Using promela in a fully verified executable ltl model checker,” in *Verified Software: Theories, Tools and Experiments*, D. Giannakopoulou and D. Kroening, Eds. Cham: Springer International Publishing, 2014, pp. 105–114.

[4] L. Frantzen, J. Tretmans, and T. Willemse, “Test generation based on symbolic specifications,” in *Formal Approaches to Software Testing (4th International Workshop, FATES 2004, Linz, Austria, September 21, 2004, Revised selected papers)*, ser. Lecture Notes in Computer Science, J. Grabowski and B. Nielsen, Eds. Germany: Springer, 2004, pp. 1–15.

[5] A. Zeller, “Yesterday, my program worked. today, it does not. why?” *SIGSOFT Softw. Eng. Notes*, vol. 24, no. 6, p. 253–267, Oct. 1999. [Online]. Available: https://doi.org/10.1145/318774.318946

[6] A. Bertolino, “Software testing research: Achievements, challenges, dreams,” in *Future of Software Engineering (FOSE ’07)*, 2007, pp. 85–103.

[7] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[8] P. Hamill, *Unit Test Frameworks: Tools for High-Quality Software Development*, ser. O’Reilly Series. O’Reilly Media, 2004. [Online]. Available: https://books.google.nl/books?id=WvFuyuc5ZAEC

[9] G. Tretmans, *Test Generation with Inputs, Outputs and Repetitive Quiescence*, ser. CTIT technical report series. Netherlands: Centre for Telematics and Information Technology (CTIT), 1996, no. TR-CTIT-96-26, cTIT Tecnnical Report Series 96-26.

[10] J. Tretmans, *Model Based Testing with Labelled Transition Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–38. [Online]. Available: https://doi.org/10.1007/978-3-540-78917-8_1

[11] M. Timmer, H. Brinksma, and M. Stoelinga, *Model-Based Testing*, ser. NATO Science for Peace and Security Series D: Information and Communication Security. Netherlands: IOS Press, 4 2011, no. 30, pp. 1–32, 10.3233/978-1-60750-711-6-1.

[12] T. Kanstrén and M. Chechik, “Trace reduction and pattern analysis to assist debugging in model-based testing,” *Proceedings - IEEE 25th International Symposium on Software Reliability Engineering Workshops, ISSREW 2014*, pp. 238–243, 12 2014.

[13] P. Koopman and R. Plasmeijer, “Fully automatic testing with functions as specifications,” in *Central European Functional Programming School*, Z. Horváth, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 35–61.

[14] Yong Lei and J. H. Andrews, “Minimization of randomized unit test cases,” in *16th IEEE International Symposium on Software Reliability Engineering (ISSRE’05)*, Nov 2005, pp. 10 pp.–276.

[15] M. Weiser, “Programmers use slices when debugging,” *Commun. ACM*, vol. 25, no. 7, p. 446–452, Jul. 1982. [Online]. Available: https://doi.org/10.1145/358557.358577

[16] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer, “Efficient unit test case minimization,” in *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’07. New York, NY, USA: Association for Computing Machinery, 2007, p. 417–420. [Online]. Available: https://doi.org/10.1145/1321631.1321698

[17] C. Artho, A. Biere, and M. Seidl, “Model-based testing for verification back-ends,” in *Tests and Proofs*, M. Veanes and L. Viganò, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 39–55.

[18] A. Niemetz, M. Preiner, and A. Biere, “Model-Based API Testing for SMT Solvers,” in *Proceedings of the 15th International Workshop on Satisfiability Modulo Theories, SMT 2017), affiliated with the 29th International Conference on Computer Aided Verification, CAV 2017, Heidelberg, Germany, July 24-28, 2017*, M. Brain and L. Hadarean, Eds., 2017, p. 10 pages.

[19] L. Clapp, O. Bastani, S. Anand, and A. Aiken, “Minimizing gui event traces,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 422–434. [Online]. Available: https://doi.org/10.1145/2950290.2950342

[20] H. Heiskanen, A. Jääskeläinen, and M. Katara, “Debug support for model-based gui testing,” in *2010 Third International Conference on Software Testing, Verification and Validation*, April 2010, pp. 25–34.

[21] K. Claessen and J. Hughes, “Quickcheck: A lightweight tool for random testing of haskell programs,” *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP*, vol. 46, 01 2000.

[22] J. Hughes, “Quickcheck testing for fun and profit,” in *Practical Aspects of Declarative Languages*, M. Hanus, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 1–32.

A RAW EXPERIMENT DATA

The raw data is from the experiments is in 25 CSV files (10 for Experiment one and 15 for experiment 2, one for each bug). These are hosted on GitHub and can be found at:

`https://github.com/Lars-Meijer/sts-shrinking-results/`.