A Model-Centric Approach to Test Case Shrinking: utilising k-shortest path in model-based testing

Susan van den Broek
susan.vandenbroek@student.uva.nl

August 1, 2024, 60 pages

Academic supervisor: Dr. Ana Oprescu, a.m.oprescu@uva.nl
Daily supervisor: Dr. Ir. Theo C. Ruys theo.ruys@axini.com
Host organisation: Axini B.V. https://www.axini.com/nl/

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Abstract

In Model-Based Testing (MBT), long traces to errors complicate bug localisation. Shrinking of these long traces, aimed at finding smaller counterexamples from existing ones, traditionally operates at the trace-level. However, leveraging the detailed behavior captured by MBT models suggests potential for more effective shrinking methods. We propose a model-centric shrinking approach, centered around the idea that finding shortest traces in MBT is similar to finding shortest paths in graphs. The goal of this thesis is therefore to 1) research how a model-centric shortest path algorithm can be leveraged in shrinking, and 2) compare the quality and performance of the proposed model-centric shrinking method to existing trace-centric shrinkers. To this end, a proof-of-concept of a model-centric shrinking method was developed. First, the concept of shrinking test cases was mapped against the shortest path problem, in order to find an appropriate shortest path algorithm for our shrinking context. It was concluded that this should be a solution to the k-shortest path problem, but no such state-of-the-art the algorithm that meets all assumptions and requirements of our shrinking context could be identified. We then therefore created our k-shortest path algorithm that utilises Breadth-First-Search and priority queues: the *SPS_AllPaths*. Two variants of this model-centric shrinker were developed, that each contain a heuristic to reduce the search space of the algorithm by means of partial order reductions: *SPS_POR_LabelCount* and *SPS_POR_Subpaths*. We tested our shrinkers in three controlled experiments. The results show that the model-centric algorithms yield smaller or comparable traces to existing trace-centric methods, in a faster fashion depending on the type of bug. We explain the differences between the results of the model-centric and trace-centric shrinkers, and give recommendations for industry practitioners as to which type of shrinker to use when. Future work is discussed.

Contents

Chapter 1

Introduction

Software should work when it matters. In order to achieve this, rigorous testing is necessary. Model-Based Testing (MBT) [1] is an automated testing approach with an academic origin that has since been adopted by industry practitioners. In MBT, a model of the System Under Test (SUT) is created. This model is then used to algorithmically generate test cases. A test case is a linear list of actions that happened at the SUT, but also correspond to a trace of transitions in the model from one state to the other. Whenever the behaviour of the SUT does not adhere to that of the model, a potential bug is found. The power of MBT lies in that the test traces are automatically generated, which means that intricate bugs can be found that would be harder to catch using unit testing [2]. The downside to this, is that the traces can get very long, and practitioners mention that failing test traces of 1000 steps are not uncommon. While these traces are automatically generated, the analysis of their failure is not automated. Bug localisation is done by the human developer, and not by the MBT suite.

Bug localisation is one of the most tedious and costly activities of the debugging process [3], and one of the challenges of debugging is cognitive overload [4]. It would thus be beneficial to developers and testers if a shorter trace to the same error could be produced, as this makes the analysis of the traces and therefore bug localisation easier. This procedure is called *shrinking* and it enables to find smaller counterexamples based on an already found counterexample [5].

Several techniques to achieve shrinking exist, and most of these methods utilise an iterative approach to step-wise shrink the failing test trace. At each iteration, the shrinking algorithm uses its heuristic or bug localisation method to decide what part of the test trace to remove. Then, the shrunk trace is put through the MBT engine again in order to see if it produces the sought-for error. Because these methods depend entirely on the original failing trace, they are limited in the alternative shorter traces that they can produce. Alternatively, because the SUT is invoked at each iteration, these shrinking methods can be slow when they require many iterations to get to the smallest trace [6]. Industry practitioners would be helped by a shrinking method that not only produces small traces, but does so in a time efficient manner.

Since the model in MBT is supposed to capture the behaviour of the SUT exactly, it could contain valuable information for finding shorter test traces. Shrinking in MBT could therefore benefit from an approach that bases its heuristics on the model instead of the original failing trace. Traces in MBT can be seen as traversals over the model; they are like paths to the bug. This means that when a trace fails, a state in which a bug can occur can be identified. Path finding algorithms can then help us to find the shortest path to that state, and with that also the shortest path to the bug. This research aims to create a *model-centric* shrinking method that is based around this concept of shortest path.

1.1 Research Questions

We formulate the following research questions:

1. How can we leverage a model-centric shortest path algorithm to shrink test cases?
2. How does the proposed model-centric shrinking method compare to traditional trace-based shrinking methods with regards to quality and performance?

1.2 Research method

This thesis utilises a mix of literature study, technical action Research, and controlled experiments to answer its research questions. This section elaborates on the different research phases that this thesis has undergone.

First, we conduct a literature study was by consulting *Google Scholar*, *UvA CataloguePlus*, *IEEE Xplore* and other academic research databases. The goal of this literature study is to find an appropriate shortest path algorithm to help answer RQ1. This means that we are mapping shrinking test cases to the shortest path problem. A summary of this study is given in Chapter 3

The next phase of the research is to create the Proof of Concept (PoC): *SPS_AllPaths*. This research artefact consists of a Ruby program capable of shrinking test traces and is integrated to a commercial MBT suite, the Axini Modeling Platform (AMP) [7]. The PoC consists of a main model-centric shrinking algorithm, the *SPS_AllPaths*, as well as two alternative, optimization variants of this algorithm, *SPS_POR_LabelCount* and *SPS_POR_Subpaths*. The PoC is described in detail in Chapter 4.

To validate the PoC, controlled experiments are conducted on synthetic or simulated SUTs and their corresponding models. These SUTs have known bugs in them, and AMP was used to generate test cases revealing them. The PoC is then used to shrink these test cases, creating smaller traces to those same errors. The effectiveness of the PoC is determined by using different metrics on the shrinking data from the controlled experiments. These metrics are in the quality and performance realms, including the shrinking percentage and the shrinking time.

In order to answer RQ2 and thus to compare the proposed model-centric approach with existing trace-centric methods, the same controlled experiments are ran with trace-centric methods as well. Meijer [6] implemented multiple trace-centric shrinkers and integrated it with AMP [6]. Meijer's work is therefore used as a validation benchmark.

1.3 Contributions

Our research makes the following contributions:

1. **A mapping of test case shrinking to the shortest path problem** aimed to find a state-of-the-art shortest path solution most suited for our shrinking context.
2. **A theoretical description of a methodology** to shrink failing test traces derived from model based testing. This methodology is model-centric and make use of a k-shortest path algorithm in order to reach the error in less steps than the original failing trace.
3. **An implementation of a proof-of-concept** for the defined methodology. This research artifact is integrated in AMP, a commercial MBT suite developed by Axini. The PoC consists of a main model-centric shrinking algorithm, and two alternative, optimization variants of that algorithm.
4. **Validation and evaluation of the proof-of-concept**. The artifact is tested on synthetic SUTs and models. The model-centric approach is also compared against trace-centric methods, based on different metrics assessing performance and quality.

1.4 Outline

This thesis is structured as follows. In Chapter 2 we describe the necessary background that is needed to understand and contextualise the research. Chapter 3 describes the mapping of shrinking to the shortest path problem that was done. This chapter helps define the shortest path algorithm that our model-centric shrinking method will utilise. The shrinkers themselves are then described in great detail in Chapter 4, which goes into the design and implementation. The experimental setup and the validation method of the thesis is described in Chapter 5. Results are shown in Chapter 6 and discussed in Chapter 7. This discussion also includes an overview of the potential threats to validity that this thesis is subjected to. Related work is then addressed in Chapter 8. Finally, we present our concluding remarks in Chapter 9 together with future work.

Chapter 2

Background

This chapter presents the necessary background information to give context to this thesis. First, we define MBT and we address how it uses Labelled Transition System (LTS)s to generate test traces. We adress non-determinism in MBT model, and explain the two different types of bugs that can occur in MBT. Then, we discuss how trace-centric shrinking methods have previously been applied to MBT.

2.1 Model-Based Testing

For our definition of MBT, we follow the tradition of conformance testing [8], where the SUT has to conform to a formal model. For our notion of MBT, the SUT is treated as a black-box without knowledge about its internal structure. More specifically, we follow Tretmans [9], who uses *input-output conformance* (ioco) as the conformance relation, and a Labelled Transition System (LTS) as the formal model. An LTS is an unweighted edge-labelled directed and possibly cyclic graph [10], where each node in the graph represents a state of the SUT, and each edge is a transition from one state to the other. This thesis uses the terms *node* and *state* interchangeably. The transitions of the LTS are labelled with the external actions of the SUT; either input, output, or quiescent actions. Inputs to the system are called *stimuli* and are denoted with a question mark, whereas outputs are called *responses* and are denoted with an exclamation point. *Quiescence* is the absence of output, and is denoted with δ. An LTS has a start state, indicated by an otherwise unconnected incoming arrow.

Formally, an LTS is defined as a 4-tuple $\langle Q, L, T, q_0 \rangle$ where: [9]:

- Q is a countable, non-empty set of states;
- L is a countable set of labels;
- $T \subseteq Q \times (L \cup \{\tau\}) \times Q$, with $\tau \notin L$, is the transition relation;
- $q_0 \in Q$ is the initial state.

In this definition, τ is a special label which represents internal behaviour of the system that is unobservable for the system's environment. If an LTS q can perform the sequence of actions `?a` $\rightarrow \tau \rightarrow \tau \rightarrow$ `?b` $\rightarrow$ `!c` $\rightarrow \tau$, then the τ-abstracted sequence of observable actions would be `?a` $\rightarrow$ `?b` $\rightarrow$ `!c`.

An LTS of a simple vending machine is displayed in Figure 2.1. This vending machine can produce `!tea`, `!coffee`, and `!soda`. If a product button is pressed, such as `?b_coffee`, the machine dispenses that product.

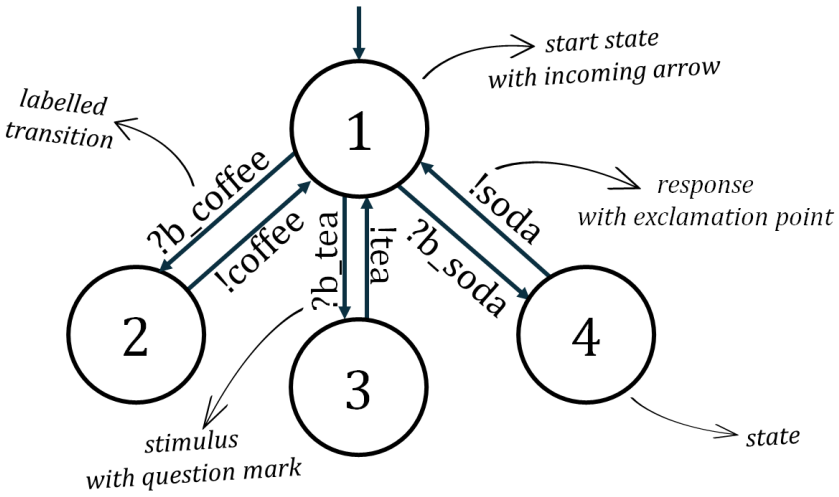


Figure 2.1: LTS of a simple vending machine

An LTS captures the behaviour of a SUT and can be used to test its correctness using the principle of the ioco relation. A SUT i is in ioco with model m when the two following rules hold:

1. If i produces output `x` after trace σ, then m can also produce `x` after σ.
2. If i cannot produce any output after trace σ, then neither can m.

Output `x` can both be a response as well as the absence of outputs: quiescence. This means that if the model expects quiescence (i.e. no response), and the SUT does produce a response, then the ioco relation is violated.

The core concept of MBT as defined by Tretmans is to generate test cases and use these to assess whether the SUT and its model are in ioco. This is done by passing the same trace of stimuli to the model and the SUT, and observe whether the SUT's response is the same as the model's. If it is not, the test trace fails and a potential bug is found. MBT is therefore a type of black-box oracle testing, in which the behaviour of the SUT is observed and compared against a ground truth (i.e. the model). The test traces that MBT uses can be automatically generated using graph traversal techniques on the model. The traversal strategy can be entirely random, or it can aim to achieve a metric such as transition coverage. Traversal strategies can also aim to hit certain scenarios, which are subtraces that describe certain test goals or SUT specifications.

While this thesis focuses on Tretmans' MBT specification using LTSs, the principles of our model-centric shrinker can be used in other MBT methods. Many different types of behaviour models have been used in MBT, such as Unified Modeling Language (UML)s [11], Finite State Machines [12], Extended State Machines [5], and Symbolic Transition Systems [6, 7]. Most of these models either by themselves are graphs, or can be represented as such. The graph properties of the model can be utilised in shrinking by applying shortest path solutions.

2.1.1 Non-deterministic Models

The models in MBT can be *non-deterministic*. This means that the model can exhibit different behaviours or outcomes from the same state under the same conditions. Non-determinism can occur due to a stimulus, but also because of a response. An example of a non-deterministic model is shown in Figure 2.2. In *State 3*, a *choice* occurs: there are multiple transitions with the same transition label. `!b` goes to *State 5*, but also back to *State 2*. Whenever this occurs, the model is in both those states simultaneously, until the choice is resolved. A choice is resolved when the model is back to being in one state instead of multiple. In our example, this could happen after the stimulus `?z`: if the SUT responses with `!c`, then the model can only be in *State 6*. However, if the response is `!a`, then the model is still in two states at the same time, namely *State 6* and *State 2*. In that case, the choice can be resolved by the next stimulus and response. If that is `?z` $\rightarrow$ `!c`, then the mode is no longer in multiple states at the same time, but only in *State 6*.

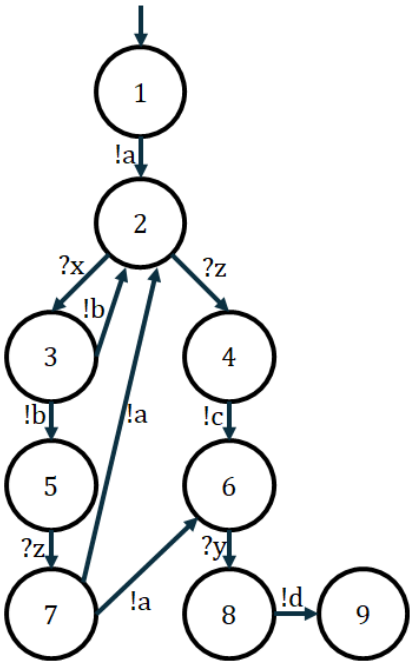


Figure 2.2: A non-deterministic model

2.1.2 Bug Types in MBT

We categorise the bugs caught by MBT engines into two types: *state bugs* and *trace bugs*. A state bug is a type of defect that consistently manifests whenever the model reaches a particular state. Consider for instance an application that uses feature toggles to enable or disable features. The model of this system has specific states that represent the configuration of which toggle switches are on and which are off. Enabling a particular feature through its toggle switch causes the application to misbehave or crash. The problem is consistently reproducible whenever that specific feature is enabled; i.e., whenever the model transitions into that specific state.

A trace bug, on the other hand, manifests whenever a certain sequence of events or inputs (i.e., a *trace*) has happened. Trace bugs are therefore not state-bound. They exist because the model is an abstraction of the specification of the SUT. While the model tracks the necessary variables and parameterised states required by the specification, it might not capture the exact behavior of the SUT, which can be more complex and include additional parameterised states. To illustrate trace bugs, we consider a webpage that crashes only after the user performs a specific sequence of actions: logging in, navigating through several specific pages in a particular order, and then submitting a certain form. In this case, the model might not capture the exact order of page visits because the specification does not require tracking this information. The SUT, however, does keep track of the order in which the pages were visited, leading to the crash after the specific sequence. Reproducing a trace bug therefore requires traversing a specific trace, and not a specific state.

2.2 Shrinking in Model-Based-Testing

Shrinking methods that were previously implemented in MBT are based on heuristics and fault localization methods operating at the *trace* level. Their input data consists of the original failed trace, and they shrink it by systematically removing labels from it. Kanstrén and Checkik [13], for instance, designed an MBT shrinking algorithm that iteratively removes duplicate occurrences of the same transition from a test trace. First, an MBT engine finds test traces (i.e. *scenarios*) that lead to an error. For every step in the scenario, the shrinking algorithm gradually reduces the number of occurrences allowed within that scenario. The newly created scenario is then fed to the SUT again via the MBT engine. If the new scenario results in an error, it iteratively undergoes the same reduction process until further reductions are no longer possible. This ensures that only the critical steps contributing to the bug remain in the scenario. The model of the SUT is not used in this process.

Willemsen [14] applied spectrum-based fault localisation [15] to MBT. In this method the root cause

of a bug is found by comparing passing traces with failing ones. If a certain step occurs in a failing trace but not in a passing one, then it is likely to contribute to the error. A similar technique to this is Delta-Debugging [16, 17]: an automated approach used to isolate the minimal set of changes that causes a test to fail. This root-cause analysis method systematically divides the set of changes applied to a program into smaller subsets. Next, each subset is tested to see if it contains the error. The search is iteratively refined until the minimal failure-inducing set of changes is found. Prior research has shown that Delta-Debugging can be used to shrink MBT test traces [6].

Koopman *et al.* [5] also proposed several MBT shrinking techniques. They include shrinking by removing single transitions; by removing chunks of transitions; removing cycles; or by performing binary search. Meijer [6] applied Delta-Debugging as well as the techniques suggested by Koopman *et al.* to a commercial MBT engine. The work of Koopman *et al.* and Meijer will be used to validate our proposed model-centric shrinker.

Our research uses a generic interface of shrinking in MBT, as provided by Koopman *et al.* [5] and Meijer [6] to allow for easy experimentation with various shrinking algorithms. The shrinking procedure works as follows. First, an MBT engine generates a trace leading to bug. Then, this trace is fed to a shrinking algorithm. The shrinker produces a new and smaller trace, which is fed back to the MBT engine. The MBT engine gives a verdict on this new trace: *pass* or *fail*. The shrinking algorithm then uses this verdict and its internal logic to decide whether it should keep shrinking or not. The *CycleShrink2* algorithm [6] for instance keeps shrinking until no more cycles can be removed from the trace. If the algorithm decides to keep shrinking, the process of generating and testing new traces continues. This interface also allows shrinkers to be chained, meaning that the result of one shrinker will be used as the input for another shrinker. Our research utilises the shrinking interface developed by Meijer [6], meaning that our shrinking algorithm requires a `trace_generation` function, and additional logic to determine whether to keep shrinking.

Chapter 3

Modelling Shrinking to the Shortest Path Problem

The sought-after failed traces in MBT are counterexamples of the conformance relation. Shrinking is the procedure of finding smaller counterexamples based on an already found counterexample. Because we know the state in which the initial counterexample occurred, a sensible strategy to find a smaller counterexample would be to build the shortest path to that state. The first part of this thesis is therefore concerned with selecting a shortest path solution that is most suited for our shrinking context. There are many different versions to the shortest path problem, each with their corresponding algorithms to solve them. This chapter summarises the literature study and problem modelling that was done in order to select an appropriate solution to our shrinking problem. First, we discuss the different variants to the shortest path problem that are out there, and select the problem most suited to our context. Then, we refine our context by listing the assumptions and specifications that the shortest path algorithm has to comply to. Next, we give an overview of the different shortest path algorithms that were researched and we discuss how they fit into our context. We conclude by selecting a shortest path algorithm most suited for our problem.

3.1 The Shortest Path Landscape

Madkour *et al.* [18] mapped out the shortest path landscape and gave an overview of the different sub-problems and their respective solutions by means of a taxonomy. In this taxonomy, there are two main branches of shortest-path algorithms: static and dynamic. For static problems, the nodes and their weights are known upfront, while for dynamic problems, nodes can be added, deleted or changed over time. The problem described in this thesis is a static one. Then, there are two main variants of the shortest-path problem: single-source shortest-path (SSSP), and all-pairs shortest-path (APSP). In SSSP, the goal is to find the shortest paths from one source node to all other nodes in the graph. The goal in APSP is to find the shortest paths between all pairs of nodes in the graph. SSSP is most appropriate for this thesis, because all test traces start from the same source node.

Additionally, our shrinking should be able to produce more than one single shortest path. Depending on the type of bug, the trace corresponding to the shortest path may *not* lead to an error. This can happen whenever a bug is not guaranteed to occur in a specific state, but requires a certain sequence of actions to happen first. In other words, the bug is a trace bug. In order to still be able to find such bugs, our shrinker has to also produce alternative shortest paths that can be of longer length. This is exactly the situation sketched in the k-shortest path problem. Algorithms solving this problem find not just the shortest path, but also the second shortest path, ..., k-shortest path. There are two variants of the k-shortest path problem. In first variant loops and duplicated states are not allowed on the path; the path is said to be *simple*.

Take for instance the example of the simple vending machine in Figure 2.1. A SUT of this model has a bug that causes it to sometimes dispense coffee or tea when the soda button was pressed. This bug only occurs if the soda button was pressed after the tea or coffee button was pressed earlier in that session. To illustrate, the following trace contains an error: `?b_soda` → `!soda` → `?b_tea` → `!tea` → `?b_soda` → `!tea`, while this trace does not: `?b_soda` → `!soda` → `?b_soda` → `!soda` → `?b_coffee` → `!coffee` → `?b_tea` → `!tea`. The shortest path to the failing state, i.e., *State 4*, is a trace of length

one: `?b_soda`. This trace does not lead to the bug, though, because `?b_tea` or `?b_coffee` did not precede `?b_soda`.

3.2 *k*-Shortest Path Algorithm Requirements

In order to select the appropriate algorithm to solve our version of the k-shortest path problem, we need to identify what exactly our context is. We identify the following assumptions and requirements for our problem:

- *Unweighted graph.* The k-shortest path algorithm has to be able to operate on unweighted graphs, as opposed to weighted ones. In our shrinking problem, we define the shortest path as the path with the least number of transitions taken. This means that each edge of the graph that is traversed, adds another length of 1 to the path. Thus, each edge in the graph has the same weight. The graph that we use, the LTS, is also an unweighted graph.
- *Non-simple paths.* The paths we produce are allowed to contain loops, and additionally we assume that no pre-processing is done on the graphs that the path-finding algorithm operates on. This means that for our requirements, the same node can be visited multiple times by a shortest path. We do not know, nor do we want to assume the nature of the bugs that are found. Non-simple paths therefore cannot be excluded from the solution space. Looking back at the vending machine of Figure 2.1, we could imagine a bug that occurred only after pressing `?button_tea` ten consecutive times. This means that the shortest path to this bug contains loops. An algorithm that generates simple paths only will not be able to produce the shortest path to this bug.
- *Dynamic k.* We do not know in advance how many shortest path variants our algorithm has to produce. Whenever a shortest path is found, the trace corresponding to this path will be fed to the MBT engine. If the trace results in a *pass*, more paths need to be generated, while if the trace results in a *fail*, we have found the shortest path to an error and the algorithm can stop producing paths. The number of attempts it takes to generate a trace that results in a *fail*, i.e. the value of k, is unknown to us. The algorithm therefore has to be able to handle a dynamic k.
- *One source node, multiple target nodes.* The shortest path always departures from the same node, namely the starting node of the LTS. However, the target node of the shortest path depends on the original failed trace. As was explained in Section 2.1.1, a trace traversing a non-deterministic model can end up in multiple states *at the same time*. When the same transition label can lead to different states, a choice occurs and it is unknown to the model what state it is in. In fact, it is in both states *simultaneously*. If the choice is not resolved before the end of the original failing trace, there are multiple destination nodes for the shortest path algorithm. This means that the algorithm has to be able to generate paths to multiple target nodes.

3.3 Candidate Algorithms

We have reviewed numerous state-of-the-art solutions to the k-shortest path problem, as summarised in Table 3.1. Below, we provide discussions on some popular algorithms. It is important to note that the majority of the papers on k-shortest path algorithms, as well as reviews and surveys on this topic (e.g. [19–21]), focus on the loopless variant instead of the non-simple variant of the problem. We also identified only one algorithm [22] specifically designed for unweighted graphs; all others are intended for weighted graphs. While weighted graph algorithms can often be applied to unweighted graphs, they may involve unnecessarily complex procedures to account for varying path lengths. In the context of this thesis in which unweighted graphs are used, these complexities are unnecessary because all path lengths are uniformly zero.

Table 3.1: Evaluation of k-shortest path algorithms

	Unweighted	Non-simple	Dynamic k	One source, multiple targets
Yen's [23]	✓*	×	∼	×
(lazy) [24] Eppstein's [25]	✓*	✓	∼	×
Roddity & Zwick's [22]	✓	×	∼	×
Hoffman & Pavley's [26]	✓*	✓	∼	×
Bellman & Kalaba's [27]	✓*	✓	∼	×
Sakarovitch's [28]	✓*	✓	∼	×
Shier's Double-Sweep [29]	✓*	×	∼	×
Schier's Label-Setting [29]	✓*	✓	∼	×
Martins' [30]	✓*	✓	∼	×

*optimised for weighted graphs, but can also handle unweighted ones
✓ Yes; × No; ∼ Possible with simple modifications

Yen's algorithm [23] is a well-known solution to the k-shortest path problem. It starts by computing the shortest path using a standard shortest path algorithm like Dijkstra's for weighted graphs or Breadth-First-Search (BFS) for unweighted graphs. This initial path serves as the first k-shortest path. The algorithm then systematically explores alternative paths by iteratively modifying the $(k-1)$-shortest path. The algorithm removes one edge at a time from the previously found k-shortest path and calculates a new shortest path around the removed edge using standard shortest path algorithms. This process continues until k valid k-shortest paths are found or until no more paths can be generated.

Eppstein's algorithm finds the k-shortest non-simple paths by leveraging efficient data structures such as trees and heaps. Initially, it computes the shortest path from the source to the target node using Dijkstra's algorithm for weighted graphs or BFS for unweighted graphs. The algorithm then explores alternative paths that branch off from the initial shortest path. These paths are managed within a shortest path tree structure, where each path is stored using an implicit representation, facilitating efficient path construction. Additionally, Eppstein's algorithm employs a heap-based approach to handle, prioritise, and explore potential paths effectively.

While Eppstein's implicit representation makes for fast look-ups, building this graph takes considerable time. This is why Jiménez and Marzal [24] propose a modified version of the algorithm in which only the parts of the graph are built that are necessary for the selection of the k-shortest paths. This improves Eppstein's worst-case running time and makes it more practical to use. They call it the *lazy version* of Eppstein's k-shortest path algorithm.

The k-shortest simple path algorithm developed by Roditty and Zwick [22] is specifically designed to work on unweighted graphs or graphs with lower weights. Because of this, it outperforms Yen's algorithm in those cases. The algorithms starts by adding the second shortest path to a priority queue that is sorted by path length. Per iteration k, an element is popped off this queue and alternative paths based of this path are constructed. The logic to find these paths are the same as with Yen's algorithm, but it is optimised to avoid redundant computations. Paths are decomposed into their deviation edge and subpaths and stored as such. A deviation edge is defined as the edge where the path diverges from existing shortest paths in the output list.

Hoffman and Pavley describe a method for determining the kth best routes from an origin node to a destination node. Their approach involves the construction of a *shortest route tree*. To construct this tree, the distances between the origin node and all other nodes in the graph needs to be computed. The tree itself is defined as the set of egdes that uniquely connects the origin node to each of the other nodes, such that the route from the origin to each node is the shortest route. This datastructure is then used to compute deviations of the shortest path. To get the second shortest path, all deviations to the shortest path are computed and the deviation with the smallest length is selected as the second shortest path. To get the third shortest path, all deviations of the second shortest path are computed. The third shortest path is then either the best deviation of the second shortest path, or the second best deviation of the first shortest path. This goes on until k-shortest paths have been found. Pollack [21] concludes from Hoffman and Pavley's work that no general estimate can be made beforehand as to the number of computations required when generating k-shortest paths.

3.4 Selecting a Shortest Path Algorithm

As can be seen in Table 3.1, we are unable to identify a state-of-the-art k-shortest path algorithm that can be applied to our shrinking context without major modifications. This is mostly due to the fact that our algorithm has to be able to deal with multiple target nodes for its shortest paths, and that it has to work efficiently on unweighted graphs. Additionally, we want our algorithm to be easy to understand and well documented, which is not the case for many of the algorithms discussed in this chapter.

Because a fitting algorithm cannot be found, we decide to create one ourselves. Our k-shortest path algorithm is centralised around Breadth-First Search (BFS). BFS is guaranteed to find the first shortest path and does so in an optimal way for unweighted graphs [31]. It can be modified slightly from the traditional implementation in order to produce the k-shortest paths instead of one single one. It imposes no prerequisites on the paths it generates and can therefore capture any type of bug. BFS is also part of many of the state-of-the-art k-shortest path solvers for unweighted graphs. The exact implementation of our BFS-centered algorithm is discussed in Chapter 4. One should be aware that this algorithm is not optimal, nor is it necessarily better in terms of time and space complexity than the ones described in this chapter. Its primary benefit is that it actually fits all assumptions and requirements necessary.

Chapter 4

Design and Implementation: developing model-centric shrinking algorithms

Our proposed model-centric approach is centered around the idea that shrinking test cases is similar to finding shorter paths in graphs. Whenever an MBT engine produces a failed test case, that fail happened in a specific state of the model (i.e. the *failing state*). A shortest path algorithm can be applied to find the shortest path from the start state of the model to that failing state. The trace corresponding to the shortest path can then be used as the input for the MBT engine in order to see if finds the error. If it does, already the first iteration of our shrinking algorithm found the shortest trace. For other shrinkers this typically takes multiple iterations.

It is however also possible that the shortest path to the failing state does not lead to the bug. This can happen when the SUT keeps track of data that the model is unaware of. If the shortest path does not lead to the error, alternative shortest paths can be constructed and tested against the SUT until the error is found. The traversal strategy that our method uses to generate these paths is a modified BFS using priority queues. The state-of-the-art k-shortest path algorithms that we considered did not match all the assumptions of our problem. We therefore implement our own a algorithm, the *SPS_AllPaths*, and two variants of this algorithm that use heuristics to reduce the search space: *SPS_POR_LabelCount* and *SPS_POR_Subpaths*.

Our algorithms assume that the SUT behaves deterministically, meaning that running the same trace multiple times always yields the same results. The model itself does not have to be deterministic, though. Because the model is an LTS, it does not contain variables. Parameterised information, such as for instance the chosen product in a vending machine, has to be captured on the state level. At the end of the shrinking process, the shrinker returns an array of labels consisting of a shortest path to the failing state. This array only contains the stimuli labels on that path, as the responses are to be observed at the deterministic SUT in the MBT procedure.

This chapter guides the reader through the design and implementation of the aforementioned model-centric shrinkers. First we address how the failing state is determined in Section 4.1. Then, we discuss in Section 4.2 the design decision about pruning, which determines the potential restrictions that our algorithms will have in building their paths. Next, we give a detailed overview of the *SPS_AllPaths* in Section 4.3. Section 4.4 describes two search space heuristic variants of the main algorithm that are implemented in order to improve its performance. We conclude this chapter in Section 4.5 by giving some illustrative examples to the different shrinking procedures.

4.1 The Destination Node(s)

As was already mentioned in Chapter 3, non-determinism in the model may cause test cases to end up in multiple states at the same time. Thus, the shrinker must find paths to multiple destination states. Before these paths can be constructed however, the destination states first have to be determined. The MBT suite that is used in this thesis, does not provide this necessary information about the original trace out of the box. We therefore implemented the `trace_reconstruction` function that, given a test trace and a model, reconstructs the path that the test has walked through that model. This way we

can determine the destination state(s) for the k-shortest path algorithm, as well as some additional information such as the states and transitions traversed by the original trace. This information is used by optimisation variants of the main algorithm.

The `trace_reconstruction` function is displayed in Algorithm 1. We explain it by the means of Figure 4.1, which shows a non-deterministic model with an error. The corresponding failed trace is: `?a` → `?b` → `!x` → `!error`. The `trace_reconstruction` function will follow this trace in the model by traversing it and keeping trace of the different route options it has. A `route_option` is defined as a struct with three attributes: an array of states that were visited by this route in order of visitation; a set of transitions that was taken by this route, and the status of this route. The status determines whether this route option is *impossible*, or still to-be-determined (*tbd*).

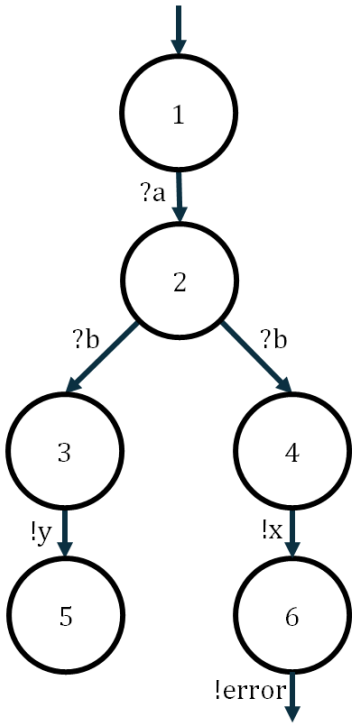


Figure 4.1: A non-deterministic model with an error trace

The `trace_reconstruction` function starts off by adding the first `route_option` to the array of `route_options`. This `route_option` contains the start state of the model, has not traversed any transitions yet, and its status is *tbd*. Then, for each step in the original trace except for the last one, the function will iterate over the `route_option`s with the *tbd* status in the `route_options` array. Per `route_option`, the function looks at its most recently added state and matches this state with the current transition label. In our example, this would be *state 1* and the label `?a`. The model is then used to find all transitions that departure from that state, using that label. In our example, this is the transition between *state 1* and *state 2*.

If there is only one transition with that label departing from that state, the target state of the transition is added to the transition array of this `route_option`. The transition itself is added to the transition set of the `route_option`. If there are multiple transitions possible with the current label and state, a new`route_option` is added to the array of `route_options`. This would be the case in the next iteration of our example. We then look at *state 2* and the label `?b`. There are two different transitions: one goes to *state 3* and one goes to *state 4*. We add new `route_option`s for each of these, and delete the original `route_option` that was currently being investigated. One of the new `route_option`s will add *state 3* to its array or states and its corresponding transition to the transition set, and the other will add *state 4* and its transition.

Whenever there are no possible transitions for the current state and trace label combination, we set the status of that `route_option` to *impossible*. This will happen in the next iteration of our example. The trace label is `!x`, and the current state is *state 3*. This state does not have that transition, so its status becomes *impossible*. The other `route_option`, the one with *state 4*, does have a transition with label `!x`, so that path continues to be expanded.

When the all labels but the last one have been processed, this means that all those `route_option`s with status *tbd* are in fact possible routes, and no more route options are being explored. The last label is not added to the route because it was the response of the model that led to the error in the SUT. This means that this label per definition does not have a corresponding transition. The `trace_reconstruction` function then determines the destination states by adding the last state of each *possible* route option state array to a set. The set of states visited and transitions traversed is the union of these arrays and sets from all *possible* route options. The latter two sets are not used by the basic implementation of the *SPS_AllPaths*, but are used by its search space reducing variants.

Algorithm 1: trace_reconstruction

Data:

- LTS model with nodes and transitions
- original test trace consisting of LTS labels

Result:

- A set of destination nodes
- A set of states visited by the original test trace
- A set of transitions traversed in the original test trace

```
1  route_options = [(states = [model.start_state], transitions = {}, status = tbd])];
2  for i in (length(original_trace) - 2) do
3      label = original_trace[i];
4      for route_option with status tbd in route_options do
5          state = route_option.states.last;
6          possible_transitions = array of transitions from state with label;
7          if length(possible_transitions) == 0 then
8              route_option.status = impossible;
9          if length(possible_transitions) == 1 then
10             route_option.states += possible_transitions.first.target_state;
11             route_option.transitions += possible_transitions.first
12         if length(possible_transitions > 1) then
13             for possible_transition in possible_transitions do
14                 new_route_option = (states = route_option.states +
                       possible_transition.target_state, transitions = route_option.transitions +
                       possible_transition, status = tbd);
15 destination_nodes = {};
16 states = {};
17 transitions = {};
18 for route in route_options with status tbd do
19     destination_nodes += route.states.last;
20     states = states.union(set(route.states));
21     transitions = transitions.union(route.transitions);
22 return destination_nodes, states, transitions;
```

4.2 Considering Model Pruning

MBT is black box testing, which means that the inner workings of the SUT are not known. We can only observe its behaviour and cannot force it to go in a certain direction. This could restrict the paths generated by the algorithms. If the only ground truths we have are the original test trace and the model, is the shortest path then allowed to assume behaviour that was not previously observed? In other words, can our algorithms generate paths that touch upon parts of the model that were not touched upon in the original test trace?

During this research, different views on this topic were explored. Depending on the viewpoint, *pruning* of the model might be necessary. The shrinking algorithm then only has access to those parts of the model that were not pruned away. The different viewpoints are:

- *Pruning by transitions.* Paths should be generated on a pruned model, based on the transitions taken by the original test case.
- *Pruning by labels.* Paths should be generated on a pruned model, based on the labels used by the original test case.
- *No pruning.* A model-centric shrinking approach should have access to the entire model.

This section explains these different viewpoints by taking a look at the model displayed in Figure 4.2. The figure contains a condensed LTS. The states 4 till 20 have been generalised; this part of the LTS contains many loops. Additionally, not all labels are shown. For this example, only the displayed labels are relevant, but please note that the other transitions have labels different than the displayed labels.

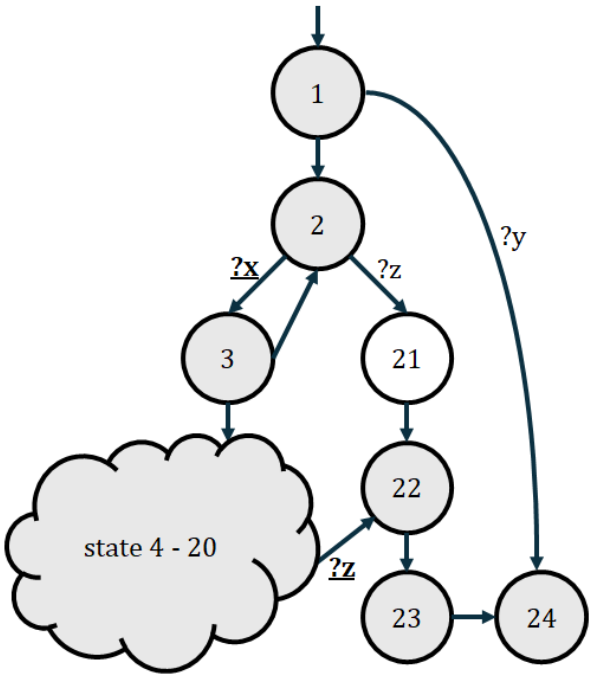


Figure 4.2: The case for pruning

Imagine that a failed trace on this model traversed the gray states, using the transitions underlined and marked in bold (i.e. the transitions from *State 2* to *State 3* and *State 4-20* to *State 22*). This trace eventually failed in *State 24*, meaning that it is now our goal to find the shortest path to this state. If we do not want to assume SUT behaviour, we could say that we only want to take those transitions that the original trace took as well. This way we know for sure that whenever we give the SUT an input at a state, we have already observed its behaviour for that input at that state before and we are therefore not forcing or assuming anything. This means that our shrinker only has access to those transitions that the original trace also used; we prune away all other transitions of the model. Thus the shrinker does not have access to the `?z` transition at *State 2* and will therefore reach *State 24* via the shortest path through *State 4-20*.

We can also prune by label. This means that our shrinker only has access to all transitions using labels that the original trace also used. With this viewpoint, we are only constructing paths using simuli that the original trace also used, but we are allowing ourselves to press these buttons at a different point in time (or at a different stage in the model) than was done in the original trace. We know that the SUT accepts this stimulus and gives a response to it in at least some states, and we also know that it was part of the original error. A model-centric shrinker following this logic would have access to the `?z` transition at *State 2*, because this label was part of the original trace. The `?y` transition from *state 1* to *State 24* is pruned away. The shortest path then skips over *state 3-20* by taking a shortcut from *State 2* to *State 22*, via the `?z` transition to *State 21*.

When no pruning is done, the shrinker has access to the entire model. This means that directly from *state 1*, a path can be built to the failing state, *State 24*. This path uses the label `?y`, which was not part of the original trace at all.

In this thesis, no pruning on the model was done. This is because we believe that a model-centric shrinker should have access to the entire model. The model in MBT is supposed to serve as a ground-truth; it is meant to capture the behaviour of the SUT exactly. We should therefore be able to utilise it to the fullest. The pruning viewpoints are opposed to this idea, because touching other parts of the model than the original trace means that you now assume SUT behaviour. We do not think that this matters, because the model is supposed to be correct. If the SUT responds in different ways than the model expected, this only means that we have found a new path to another potential bug. We also want to give our shrinkers access to the entire model, because this allows our method to be more model-centric, which is the goal of this thesis.

4.3 SPS_AllPaths; trace generation

We now know how to get the destination nodes for our shrinking algorithm, and we have also defined the part of the model that the shrinker is allowed to operate on. The next step is creating the actual algorithm itself; the *Shortest Path Shrinker - All Paths (SPS_AllPaths)*.

The algorithm takes as input an LTS model and the final state(s) of the model where the failed trace ended. The source for the k-shortest path algorithm is always the starting node of the model, but a test trace may end up in multiple destination-nodes *at the same time*, due to the possible non-determinism in the model.

The main component of the *SPS_AllPaths*, the `trace_generation` function, is described in Algorithm 2. This function builds paths to the destination nodes and turns those paths into traces that can be tested by the MBT engine.

In order to find k-shortest non-simple paths, two queues are being used: the `main_queue`, and the `k_queue`. The `k_queue` contains paths with at least one loop more than those in the `main_queue`. Utilising two queues with different priority helps keeping queues small, mitigating the explosion of the number of paths and with that the number of queue elements that our algorithm has to consider. The queues are implemented as dictionaries, with as key the path size of the paths in that bucket, and as value an array holding the actual queue. The node elements in this array consist of both the most recently added state, as well as the path to that state. The `main_queue` is a global variable for the shrinker, and is initialised containing only the start state of the model. The `k_queue` is initialised empty with each call of the `trace_generation` function. This function also uses a set, the `visited_set`, to keep track of which nodes have been visited by the `main_queue`.

Algorithm 2: SPS_AllPaths **trace_generation**

Data:
- LTS model with nodes and transitions
- destination_nodes: set of target nodes
- global variable main_queue: dictionary which holds the BFS queues

Result: Stimuli of shortest path to destination_nodes

```
1  visited_set = [];
2  k_queue = {};
3  path_size = the smallest key of main_queue;
4  found = false;
5  while not found do
6      current_queue = main_queue[path_size];
7      while not found && current_queue is not empty do
8          node = current_queue.pop;
9          if node not in visited_set then
10             for transition in node.transitions do
11                 target_node = transition.target_node;
12                 target_node.path = node.path + transition;
13                 main_queue[path_size + 1] += target_node;
14             visited_set += node;
15         else
16             k_queue[path_size] += node;
17         if node in destination_nodes then
18             found = true;
19     path size += 1;
20 merge k_queue into main_queue for a possible next call of trace_generation;
21 return the stimuli of node.path;
```

From line 5 in Algorithm 2, the algorithm iterates over the `main_queue`, starting with the smallest `path_size` and working its way up to larger `path_size`s. At each iteration, all nodes with paths of `path_size` are stored in `current_queue`. Each `node` of this queue is then visited. If the `node` is not yet present in the `visited_set`, all of the neighbours of this `node` are added to the right `path_size` bucket

of the `main_queue`. The `node` is then added to the `visited_set`. If the `node` had already been visited, it is added to the *`k`*`_queue` instead and its neighbours will be visited at a later time. It is then checked whether the `node` is a destination node. If it is, a shortest path has been found and algorithm stops exploring nodes. This check is performed after processing the `node`, as this ensures that alternative paths are added to the queue. This is necessary in case the shortest path that was just discovered, does not lead to the bug.

Once a shortest path has been found, the MBT engine uses it as the input for a test case. The `verdict` of this test is then returned to the *SPS_AllPaths*. The shrinking process stops once `trace_generation` produces a trace that yields the verdict *`failed`*. As long as the verdict is *`passed`*, the `trace_generation` function is called again and a new (but possibly longer) shortest path is generated.

4.4 Partial Order Reduction Optimisations

Initial testing with the *SPS_AllPaths*, showed that it can take this shrinker many attempts to find to correct path to a trace bug. Whenever the shortest path does not directly lead to the bug, the *SPS_AllPaths* will always produce *all* possible paths to the state where the bug was found, until it happens to stumble upon one that contains the error. This means that the algorithm produces a lot of different but very similar shortest path variants.

We might not always be interested in all paths, though; sometimes a subset of these paths is enough. Take for instance the trace `?A` → `?B` → `?A` → `?B` , and the trace `?A` → `?A` → `?B` → `?B` , and `?B` → `?B` → `?A` → `?A` . These traces are of the same length and they end up in the same state, but they are seen as different paths by our shrinker. Because the order in which stimuli occur are relevant to the *SPS_AllPaths* queue explosion may and does occur. On top of that, it also results in the generation of many different shortest paths. Because each of these paths will have to be tested by the MBT engine, the total shrinking time is higher.

This is why we built two Partial Order Reduction (POR) optimisations for our total order shrinker: *Shortest Path Shrinker - Partial Order Reduction - Label Count (SPS_POR_LabelCount)*, and *Shortest Path Shrinker - Partial Order Reduction - Subpaths (SPS_POR_Subpaths)*. These are variants of the *SPS_AllPaths* that use a heuristic to determine whether a (sub)trace is equal to another (sub)trace. If they are deemed equal, only one of them is explored and the other is discarded. This resembles the partial order reduction methods found in model checking [32, 33]. The two new shrinkers have internal logic to determine whether subtrees are equal are not, and when they are these shrinkers only add one of them to their queues. They optimise our algorithm in two ways. On one hand, the resulting queue is smaller, meaning less searching is needed. On the other hand, these shrinkers will find fewer shortest paths which means that a smaller number of traces are being fed to the MBT engine for testing.

4.4.1 SPS_POR_LabelCount; trace generation

Early evaluation of the *SPS_AllPaths* shows that the paths it generates include many different but similar ways to get to the failing state. The difference between paths of the same length is often only the order in which stimuli occurred, while this order is irrelevant for the bug to occur. This is why the first POR variant we propose, *SPS_POR_LabelCount*, counts labels to determine when some traces are effectively equivalent.

Imagine for instance a website that contains a bug when visiting webpage `W`. This bug only occurs when `W` is visited after first visiting the three other webpages of the website, page `A`, `B`, and `C`, regardless of the order. This means that four clicks are needed to reproduce the bug: a click to the first webpage, to the second, to the third, and finally the fourth click to webpage `W` itself. The *SPS_AllPaths* detects this bug by systematically exploring paths of increasing length. Initially, it generates paths of one click, then paths of two clicks, then three clicks, and finally paths of four clicks. The number of different paths grows factorially with the number of clicks. The *SPS_POR_LabelCount* algorithm only produces one of these variants per path length.

SPS_POR_LabelCount keeps track of how many times each label occurs in a constructed path and adds this data to the queue alongside the visiting node and the path. Whenever a path is being explored, its configuration in added to a global set: the `queue_composition`. A configuration of a path consists of a dictionary, with as key the label name, and as value its count. When going over all the neighbours of the `node`, this algorithm only adds those neighbours that can be reached via a configuration that is not present in the `queue_composition` set yet.

This shrinker sees the aforementioned `A-B-C` traces as the same, as they all have the same configuration: `configuration: "A": 1, "?B": 1, "?C": 1`. The *SPS_POR_LabelCount* algorithm will therefore only explore one of these traces, resulting in smaller queues and less traces that need to be tested. The downside of this approach, is that *SPS_POR_LabelCount* is not able to catch all types of bugs. If the order of inputs is relevant for a bug to show itself, this algorithm is not guaranteed to find the shortest failing path to that bug. For instance, if the aforementioned website bug only occurs if page `C` is directly followed by page `A`, then *SPS_POR_LabelCount* might not be able to find a path to this bug. See Algorithm 3 for `trace_generation` function of *SPS_POR_LabelCount*.

Algorithm 3: SPS_POR_LabelCount `trace_generation`

Data:
- LTS model with nodes and transitions
- destination_nodes: set of target nodes
- global variable main_queue: dictionary which holds the BFS queues
- global variable queue_composition: dictionary which holds the compositions of explored paths

Result: Stimuli of shortest path to destination_nodes

```
visited_set = [];
k_queue = {};
path_size = the smallest key of main_queue;
found = false;
while not found do
    current_queue = main_queue[path_size];
    while not found && current_queue is not empty do
        node = current_queue.pop;
        if node not in visited_set then
            for transition in node.transitions do
                new_composition = node.composition;
                new_composition[transition.label] += 1;
                if new_composition not in queue_composition then
                    target_node = transition.target_node;
                    target_node.path = node.path + transition;
                    target_node.composition = new_composition;
                    main_queue[path_size + 1] += target_node;
                    queue_composition += new_composition;
            visited_set += node;
        else
            k_queue[path_size] += node;
        if node in destination_nodes then
            found = true;
    path size += 1;
merge k_queue into main_queue for a possible next call of trace_generation;
return the stimuli of node.path;
```

4.4.2 SPS_POR_Subpaths; trace generation

The first POR algorithm reduces the search space in one particular case, i.e. when many different paths consisting of the same labels arranged in different orders lead to the failing state. It therefore does not work on bugs for which the order of labels does matter. It also does not take the model's structure into consideration; only the structure of the trace. We therefore set out to create a more model-centric POR variant that is applicable on all types of bugs: the *SPS_POR_Subpaths*. This algorithm looks at subpaths, and takes the failing trace into account during path construction. Instead of generating all possible subpaths between nodes, only subpaths which occurred during the failing trace are included. If there is no such original subpath, only one of the other non-original paths is included. We explain the logic of the *SPS_POR_Subpaths* by means of an illustrative example, shown in Figure 4.3.

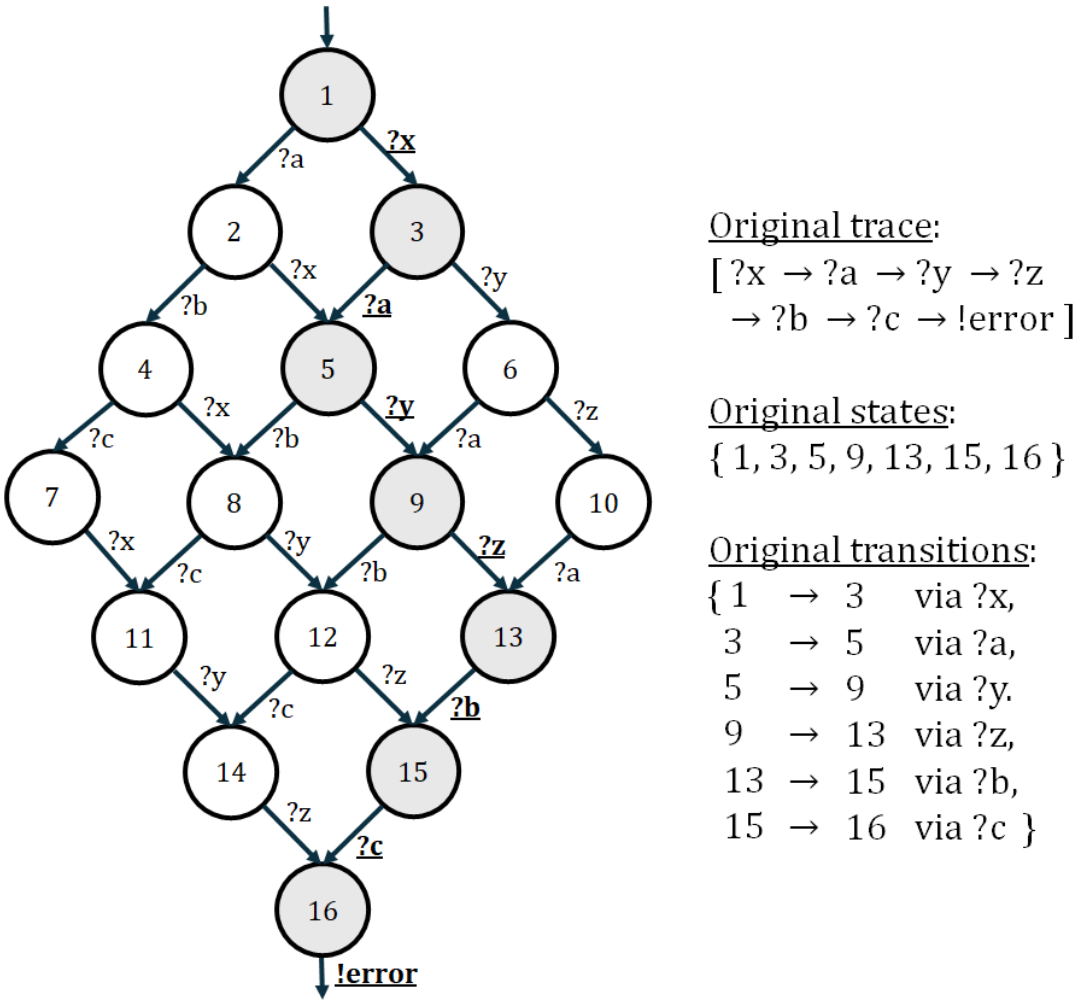


Figure 4.3: An LTS with many similar subpaths illustrating SPS_POR_Subpaths

Figure 4.3 shows and LTS in which states can be reached in many different but similar ways. For simplicity, only the stimuli labels are shown; however, it should be understood that between each stimulus, a corresponding response and state transition occurs. This scenario is common in real-life models. For example, at a vending machine, a balance of €3 can be achieved by first inserting €1 and then €2, or by inserting €2 first and then €1. Similarly, in a paint mixing machine, adding red first and then blue, or blue first and then red, will always result in a purple mixture. It is also common for the same state to be reachable through multiple pathways, as is also depicted in Figure 4.3. A total bank balance of €100 can be achieved by depositing or withdrawing various amounts, depending on the starting balance. Likewise, an elevator can arrive at a particular floor from any other floor.

Turning again to Figure 4.3, the path of the original test case is shown by gray states and underlined transitions in bold; we can see that the trace eventually failed in *state 16*. The *SPS_POR_Subpaths* will construct a shortest path to this state, but it will not consider all possible paths. Instead, it will prioritise those subpaths that were present in the original trace over those that were not traversed in the original trace. This increases the likelihood that the paths constructed by this shrinker are able to capture trace bugs earlier on.

When the *SPS_POR_Subpaths* encounters a state that can be reached via multiple paths of the same length, it will, by default, add only one of these paths to the queue. If a state can be reached through a transition that was also part of the original trace (an *original transition*), this transition takes priority over non-original transitions. For example, *State 5* can be reached in two ways: via the `?x` transition from *State 2*, and via the `?a` transition from *State 3*. Since the `?a` transition from *State 3* is an *original transition*, the *SPS_POR_Subpaths* will prioritise it and only add the `?x` → `?a` path to the queue, not the `?a` → `?x` path.

If there are multiple subpaths of the same length that reach the same state using different original transitions, all of these subpaths are added. However, if a state cannot be reached via an original

transition, such as *State 8*, all incoming paths to this state are considered equal. Consequently, the *SPS_POR_Subpaths* will add only one of these subpaths to the queue.

The advantage of this shrinker, is that it reduces the search space of the algorithm effectively and in a model-centric manner, while remaining the ability to find any type of bug. The downside is that the path that it finds might not be the shortest possible path to the failed state of a trace bug. This is because the shortest path can be hidden in those parts of the model that were not touched by the original trace. The *SPS_POR_Subpaths* only explores some of the subpaths in this area of the model, meaning that the actual shortest path could be discarded along the way. The *SPS_AllPaths* does have this guarantee; its final paths are always the shortest possible path to the bug in the failed state. However, the *SPS_AllPaths* might need many interactions with the MBT engine in order to find trace bugs, as all paths are generated and thus tested against the SUT. In contrast, the *SPS_POR_Subpaths* is designed to filter out many of these paths, and should therefore be able to generate its final shrunk trace with less SUT-interactions.

The *SPS_POR_Subpaths* uses several datastructures: the `subpaths_hash`, `subpath_to_node`, and `node_of_subpath`. The `subpaths_hash` is a dictionary with as key a `path_size`, and as value a `subpath_to_node`. The `subpath_to_node` is also a dictionary, with as key the `node` that can be reached in a `path_size`-number of transitions, and as value a `node_of_subpath`. The `node_of_subpath` datastructure is a struct with three attributes: a `node` that was traversed in path construction, a `transition` that was used to reach the `node`, and the `queue_element` that was put into the `main_queue` when the `node` was added to this queue during path construction. A schematic overview of these datastructes and their relations is found in Figure 4.4.

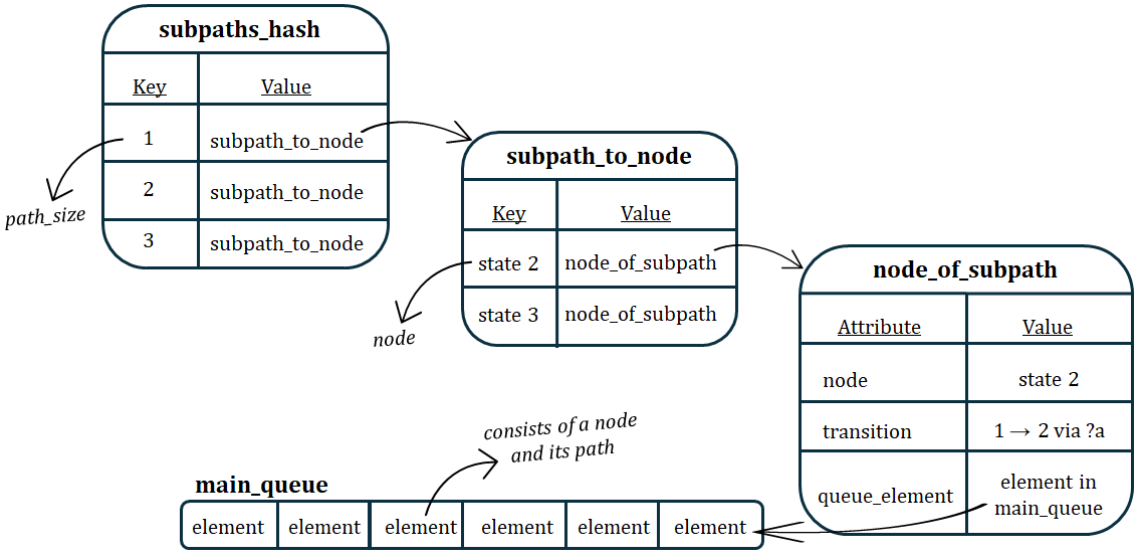


Figure 4.4: The datastructures of the SPS_POR_Subpaths

The pseudocode of the *SPS_POR_Subpaths* `trace_generation` function is found in Algorithm 4. When iterating over the `transitions` of the `node`, the *SPS_POR_Subpaths* checks at Line 11 whether the `target_node` of that `transition` is present in the `subpaths_hash`. If it is not, then this `node` has not been explored by the algorithm before and it will be added to the `main_queue` and the `subpaths_hash`. If the `target_node` was already explored before, the algorithm checks whether the `transition` currently being considered is an `original_transition`. If it is, it should be added to the `main_queue` despite being explored before.

The algorithm finds the `node_of_subpath` belonging to the oldest member of this `subpath_to_node` on Line 18. If this `node_of_subpath` does not contain an `original_transition`, its `queue_element` is edited by changing its path into the path of the current `node`, and adding the `transition` to it. The `node_of_subpath.transition` is also changed to the `transition`. In a sense, the algorithm has now swapped out the old `queue_element` by this new `queue_element` that does contain an `original_transition`.

If the `node_of_subpath` of Line 18 already contains an `original_transition`, it is left alone and the current `transition` is added to the `main_queue` alongside it. In both cases, the new `node_of_subpath` is not added to the `subpaths_hash`, as the `node_of_subpath` currently residing there already contains the necessary information.

Algorithm 4: SPS_POR_Subpaths **trace_generation**

Data:
- LTS model with nodes and transitions
- destination_nodes: set of target nodes
- original_states: set of states visited by the original test trace
- original_transitions: set of transitions traversed by the original test trace
- global variable main_queue: dictionary which holds the BFS queues
- global variable subpaths_hash: dictionary with as key path sizes and as value a node that can be reached via a path of that size

Result: Stimuli of shortest path to destination_nodes

```
1  visited_set = [];
2  k_queue = {};
3  path_size = the smallest key of main_queue;
4  found = false;
5  while not found do
6      current_queue = main_queue[path_size];
7      while not found && current_queue is not empty do
8          node = current_queue.pop;
9          if node not in visited_set then
10             for transition in node.transitions do
11                 if transition.target_node not in subpaths_hash[path_size + 1] then
12                     target_node = transition.target_node;
13                     target_node.path = node.path + transition;
14                     main_queue[path_size + 1] += target_node;
15                     subpaths_hash[path_size + 1] += (target_node, transition);
16                 else
17                     if transition in original_transitions then
18                         other_node_of_subpath = subpaths_hash[path_size +
                            1][transition.target_node];
19                         if other_node_of_subpath.transition not in original_transitions then
20                             other_node_of_subpath.queue_element.path = node.path + [transition];
21                             other_node_of_subpath.transition = transition;
22                         else
23                             target_node = transition.target_node;
24                             target_node.path = node.path + transition;
25                             main_queue[path_size + 1] += target_node;
26             visited_set += node;
27         else
28             k_queue[path_size] += node;
29         if node in destination_nodes then
30             found = true;
31     path size += 1;
32 merge k_queue into main_queue for a possible next call of trace_generation;
33 return the stimuli of node.path;
```

4.5 Shrinking Procedure Examples

This section elaborates on the three different model-centric shrinking algorithms by means of an illustrative example. Take the model displayed in Figure 4.5. Like the example of Figure 4.3, it only shows stimuli and the original trace is marked with gray states and bold, underlined transitions. The SUT that this model was tested against, contains a trace bug: *State 14* will produce `!error` if somewhere along the trace, the `?s` stimulus was chosen.

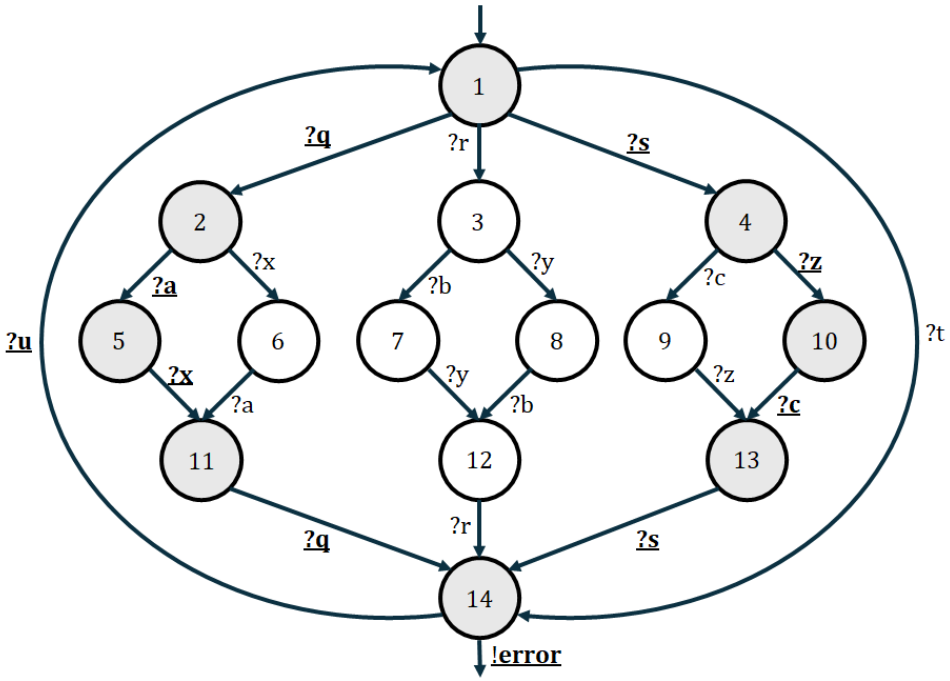


Figure 4.5: A simplified LTS with failed trace

Original trace

The original trace that our three algorithms will shrink in this example, is the following:
`?q` → `?a` → `?x` → `?q` → `?u` → `?s` → `?z` → `?c` → `?s` → `!error`.

SPS_AllPaths

SPS_AllPaths will produce all paths to the failed state, i.e. *State 14*, until it encounters the error. Those paths are of increasing length:

1. `?t`.
2. `?t` → `?u` → `?t`.
3. `?q` → `?a` → `?x` → `?q`.
4. `?q` → `?x` → `?a` → `?q`.
5. `?r` → `?b` → `?y` → `?r`.
6. `?r` → `?y` → `?b` → `?r`.
7. `?s` → `?c` → `?z` → `?s` → `!error`.

SPS_POR_LabelCount

SPS_POR_LabelCount generates all paths to *State 14*, but disregards those paths that contain a trace configuration that it has already explored previously. The trace configuration is defined by the number of occurrences of each transition label. This means that only the first subpath of the diamond-shapes will be explored. For instance: `?a` → `?x` is included, and therefore `?x` → `?a` is not. This results in the following traces:

1. `?t`.
2. `?t` → `?u` → `?t`.
3. `?q` → `?a` → `?x` → `?q`.
4. `?r` → `?b` → `?y` → `?r`.
5. `?s` → `?c` → `?z` → `?s` → `!error`.

SPS_POR_Subpaths

When the *SPS_POR_Subpaths* encounters a state that it previously reached with the same path size, it prioritises those transitions that were also part of the original trace. In our example, *State 14* can be reached in via a path of size 4 with three different transitions: `?q`, `?r` and `?s`. Of those transitions, `?r` was not part of the original trace. This means that this path will not be included. Additionally, *State 11* and *State 13* can also be reached via multiple transitions. Here, the same logic is followed: in the original transition, *State 11* was reached via transition `?x`, and *State 13* via `?c`. The other paths are discarded. Therefore, the following traces are generated:

1. `?t`.
2. `?t` → `?u` → `?t`.
3. `?q` → `?a` → `?x` → `?q`.
4. `?s` → `?z` → `?c` → `?s` → `!error`.

As you can see, trace 4 of the *SPS_POR_Subpaths* is slightly different to trace 5 of *SPS_POR_LabelCount*. This is because *SPS_POR_LabelCount* uses the first trace configuration it encounters, while *SPS_POR_Subpaths* includes those transitions that are part of the original trace.

The number of traces produced by *SPS_POR_Subpaths*, depends upon the original trace. If the original trace contains a small number of transitions, less paths will be explored. Meanwhile, if the original trace contains a large number of transitions, more paths will be explored. This is illustrated in the following two examples:

Example 1

Original trace with few transitions:
`?t` → `?u` → `?t` → `?u` → `?s` → `?z` → `?c` → `?s` → `!error`.

Results in the traces:

1. `?t`.
2. `?t` → `?u` → `?t`.
3. `?s` → `?z` → `?c` → `?s` → `!error`.

Example 2

Original trace with many transitions:
`?t` → `?u` → `?q` → `?a` → `?x` → `?q` → `?u` → `?q` → `?x` → `?a` → `?q` → `?u` → `?r` → `?y` → `?b` → `?r` → `?u` → `?s` → `?z` → `?c` → `?s` → `!error`.

Results in the traces:

1. `?t`.
2. `?t` → `?u` → `?t`.
3. `?q` → `?a` → `?x` → `?q`.
4. `?q` → `?x` → `?a` → `?q`.
5. `?r` → `?y` → `?b` → `?r`.
6. `?s` → `?z` → `?c` → `?s` → `!error`.

Chapter 5

Experiments and Validation

We answer RQ2 by means of controlled experiments. To evaluate the effectiveness of the proposed shrinking algorithms and validate their results, we integrate them into an existing shrinking interface. This shrinking interface is a research artefact developed by Meijer [6], built on top of a commercial MBT suite; the Axini Modeling Platform [7]. We conducted three controlled experiments, for which we developed multiple SUTs in Ruby and added their corresponding models to the MBT suite. The suite was then used to generate test cases for these systems. The shrinking interface was subsequently employed to reduce these test cases using both the proposed model-centric shrinkers and traditional trace-centric shrinkers.

The following section will delve into the details of the MBT suite and the shrinking interface. Next, we will provide a brief overview of the trace-centric shrinkers used for validation, followed by an in-depth discussion of each of the three experiments.

5.1 Shrinking in the Axini Modeling Platform

The MBT suite used in this thesis is the Axini Modeling Platform (AMP), developed by Dutch company Axini. AMP approaches MBT as defined by Tretmans [9], which means that the ioco relation is used to define conformance. In AMP, users can create models of the SUT using a DSL, the Axini Modeling Language (AML). The AML model is then translated to an internal Ruby representation of a Symbolic Transition System (STS). An STS is like an LTS, with the extra addition of variables. This means that any STS that does not use variables, is an LTS.

To configure a test, the user sets the number of test cases they want to run, as well as the maximum trace length of each test, and the testing strategy that AMP should follow when traversing the model. A strategy could for instance pick its next transition at random, or it could try to optimise a certain metric, such as transition coverage. A strategy aiming for a high transition coverage will prioritise transitions it has not tried before in that test run.

During the execution of a test, AMP traverses through the AML model by selecting transitions. When a stimulus is chosen, the same stimulus is sent to the SUT. At all times, AMP checks whether the behaviour of the SUT is in ioco with the behaviour of the model. If it is not, a test case results in a fail. A test case passes if the maximum trace length has been reached without encountering nonconformant behaviour, or when the model ends up in a state in which no more transitions can be chosen.

The shrinking algorithms themselves were built on top of an existing shrinking framework within AMP, created by Meijer [6]. It acts like an interface and works as follows. First, AMP generates the original test traces as per the user's test configuration. Once this process is completed, the shrinking framework feeds all of the failed traces to the shrinking algorithms, one shrinker at a time. Once one shrinker has completed its procedure on all of the original traces, its results are stored and the original traces are passed on again to the next shrinker. The framework can be configured to use shrinkers in sequence. This means that the shrunk traces of one shrinker are used as the input trace for another shrinker. Combining shrinkers this way can lead to different and often improved results. A schematic overview of AMP and Meijer's framework can be found in Figure 5.1.

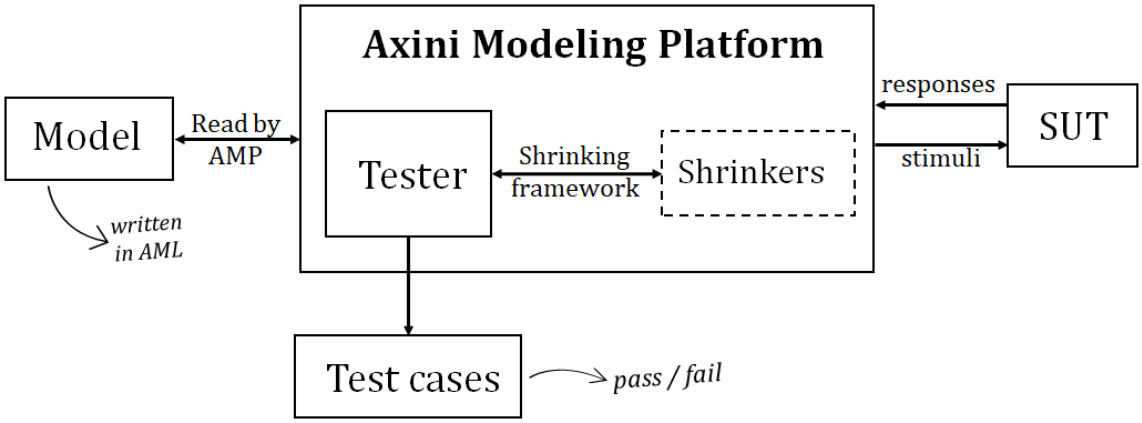


Figure 5.1: Shrinking in the Axini Modeling Platform

5.2 Validation with Trace-Centric Shrinkers

To validate the results of our model-centric shrinking algorithms, we compared them with traditional trace-centric shrinkers. The latter shrinkers were developed by Koopman *et al.* [5], and translated to Ruby and integrated within AMP by Meijer [6]. Not all of Meijer's shrinkers are included in this thesis. The shrinkers that we selected to be compared with our model-centric shrinkers demonstrated the best balance of quality and performance. An overview of these shrinking algorithms is provided in Table 5.1.

Table 5.1: Trace-centric Shrinking Algorithms

Algorithm	Description
CycleShrink2	Shrinks traces by removing cycles, starting with the biggest one.
BinaryShrink2	Uses binary search to shrink traces.
ElemShrink2	Shrinks traces by removing one stimulus at a time.
CycleShrink2, ElemShrink2	Performs *CycleShrink2* and uses its output as the input for *ElemShrink2*.
CycleShrink2, BinaryShrink2	Performs *CycleShrink2* and uses its output as the input for *BinaryShrink2*.

5.3 Experiment One: SmartDoor

SmartDoor is IoT door that locks, unlocks, opens and closes using an app. Locking the door is done with a four-digit passcode, and unlocking it requires that same user-defined passcode. An `?invalid_passcode` would be any passcode that is not a positive 4-digit number. Whenever the app sends a message (`?open`, `?close`, `?lock`, `?unlock`), the door sends a response back (such as `!opened` or `!closed`). Commands are only accepted when they are possible and are otherwise met with the `!invalid_command` response. For instance, the door cannot `?unlock` if it is `!open`, and it cannot `?close` if it is already `!locked`. If an `?incorrect_passcode` has been entered three times in a row without entering a `?correct_passcode`, SmartDoor shuts off (i.e. `!shut_off`). In this shut-off state, the door can receive up to one command from the app. Afterwards, the app is no longer able to send messages to the door.

Because our shrinking algorithms work with LTSs and therefore do not have access to variables, the passcode cannot be passed along with the `?lock` and `?unlock` stimuli as a parameter. We therefore introduce different `?lock` and `?unlock` variants: `?lock_correct_passcode`, `?lock_invald_passcode`, `?unlock_correct_passcode`, `?unlock_incorrect_passcode` and `?unlock_invald_passcode`. The different passcode attempts are encoded in the states. This means that there are three variants of the locked state: *locked* (no incorrect attempts), *locked_1* (one incorrect attempt), and *locked_2* (two incorrect attempts; will `!shut_off` if another incorrect passcode is entered). Different implementations of the SmartDoor were created, each with a different bug. These mutants of the system are summarised in

Table 5.2. Our model has 25 states and 58 transitions. It accepts 7 different stimuli, and has 8 unique responses. For access to the model, please contact the author of this thesis.

Table 5.2: SmartDoor Mutants

Mutant	Bug Type	Description
m01	State	The door cannot be closed in an opened state.
m02	State	The door accepts `?open` command when opened and `?closed` in a closed stte.
m03	Trace	The door can be locked twice.
m04	State	The door can be unlocked with an incorrect passcode.
m05	State	The door responds with `!invalid_command` after a lock or unlock command with an valid passcode.
m06	State	The door accepts lock and unlock commands with an invalid passcode.
m07	State	The door never shuts off.
m08	Trace	The door counts invalid passcodes as an unlock attempt.

5.4 Experiment Two: Koopman's Vending Machine

Koopman *et al.* [5] describe a vending machine SUT that accepts two types of coins (`?Coin1` and `?Coin2`) and keeps track of the balance inserted thus far. The user can choose between five different products (e.g. `Coffee, Espresso, French, Wiener, Double`) by pressing the `?Choice`-button. If there is enough balance, the machine will dispense the chosen product after pressing the `?Go`-button. All products cost 3, except for coffee which costs 2. The `?Info`-button can be pressed to get the current balance and chosen product, and the `?Reset`-button can be used in order to reset the choice to `None` and the balance to zero. If there was any remaining balance after a reset, this will be returned to the user in the form of change (with responses like `!Change_2` and `!Change_7`). The same vending machine was also used by Meijer [6] to validate his results.

We translated the machine specification to an LTS by making the different states encode the chosen product and the current balance (e.g. with states such as *None_4*, *Coffee_11*, and *Espresso_0*). A maximum balance of 15 was also added to the machine in order to have a finite number of states. Users can insert a coin from any state, as long as it does not result in an exceeded maximum balance. Users can also always press the `?Choice_[product]`-button (e.g. `?Choice_Wiener` or `?Choice_Double`), which will set the chosen product. Users can also at any time press the `?Go`-button, which does nothing when there is insufficient balance or no chosen product, and will otherwise dispense the chosen product and subtract its price from the balance. The model has 273 states and 1119 transitions. It accepts 106 different stimuli, and has 117 unique responses. Different implementations of this system were created, each with a different bug. These mutants are discussed in Table 5.3. and the AML model can be found in Appendix A.1.

Table 5.3: Koopman Mutants

Mutant	Bug Type	Description
m01	Trace	Remembers the product chosen after dispensing a cup.
m02	Trace	Loses any remaining balance after dispensing a product.
m03	State	Increments the balance with 1 when inserting a Coin2.
m04	Trace	Does not set the product to None but keeps it as is upon a ?Reset.
m05	Trace	Uses a uniform price of 3 for all products.
m06	State	Dispenses coffee independent of the chosen product.
m07	Trace	Inserting a Coin2 in this machine sets the chosen product to None.
m08	Trace	Inserting a Coin2 sets the chosen product to coffee.
m09	Trace	Does not perform a balance check when pressing ?Go.
m10	Trace	Remembers the chosen product if there is remaining balance after dispensing a product.

5.5 Experiment Three: ATM

The final experiment involves an Automatic Teller Machine (Automatic Teller Machine (ATM)). Users can interact with the ATM to `?withdraw` from and `?deposit` money to their bank card. The ATM has different bills that it can give out (i.e. `!5`, `!10`, `!20`, `!50` and `!100`). Users start by inserting their bank card into the ATM. If it is a `!valid_card`, the user now has to enter their Personal Identification Number (PIN). If the `?PIN` is `!correct`, the user gains access to their balance and can start depositing and withdrawing money, one bill at a time. After three incorrect PIN attempts, the ATM `!blocks` the bank card. Users can hold a maximum balance of 500 and a minimum balance of zero. If a user tries to withdraw money that would lead to a negative balance, the ATM returns `!insufficient_balance`. When a user successfully withdraws money, the ATM returns the value of the bill withdrawn. Upon a deposit, the ATM returns the new `!balance: <balance>`. Users can also request their current balance (i.e. `?balance`). When the user is done interacting with the ATM, they can `?exit` the system to get their `!card` back. The ATM will then sit idle until a new card is inserted. Our model of the ATM has 1502 states and 3080 transitions. It accepts 416 different stimuli, and has 213 unique responses. Different implementations of the ATM were created, each with a different bug. These mutants of the system are discussed in Table 5.4, whereas the AML model can be found in Appendix A.1.

Table 5.4: ATM Mutants

Mutant	Bug Type	Description
m01	State	ATM does not perform the insufficient balance check.
m02	Trace	Withdrawing a bill does not decrease the balance.
m03	State	Withdrawing a 100 bill withdraws a 10 bill instead.
m04	State	Accepts incorrect PIN codes.
m05	Trace	Remembers the most recently deposited bill and will return that instead of the actual balance when the balance is asked.

Chapter 6

Results

This chapter presents the individual results of the three different experiments. For each experiment, we report on the following metrics, broken down per bug type:

- *Shrinking Percentage.* This is the average amount of shrinking done by a shrinking algorithm, expressed as a percentage of the length of the original trace. If the original trace consists of 10 steps, and the final shrunk trace consists of 2 steps, then the shrinking percentage is 80%.
- *Test Cases per Shrink.* This is the average number of test cases the shrinking algorithm generated in order to get to its final shrunk test trace.
- *Time per Shrink.* This is the average time in seconds it takes a shrinking algorithm to shrink one original trace.

In some cases, the original failing trace already was the shortest possible path to that bug. These instances are not included when computing the shrinking percentage, but would yield a percentage of zero. They are however included when computing the test cases per shrink and the time per shrink, as resources were spent to establish that the original trace is indeed the shortest path. Including these cases does not change the overall conclusion; the general pattern and subsequent findings remain consistent.

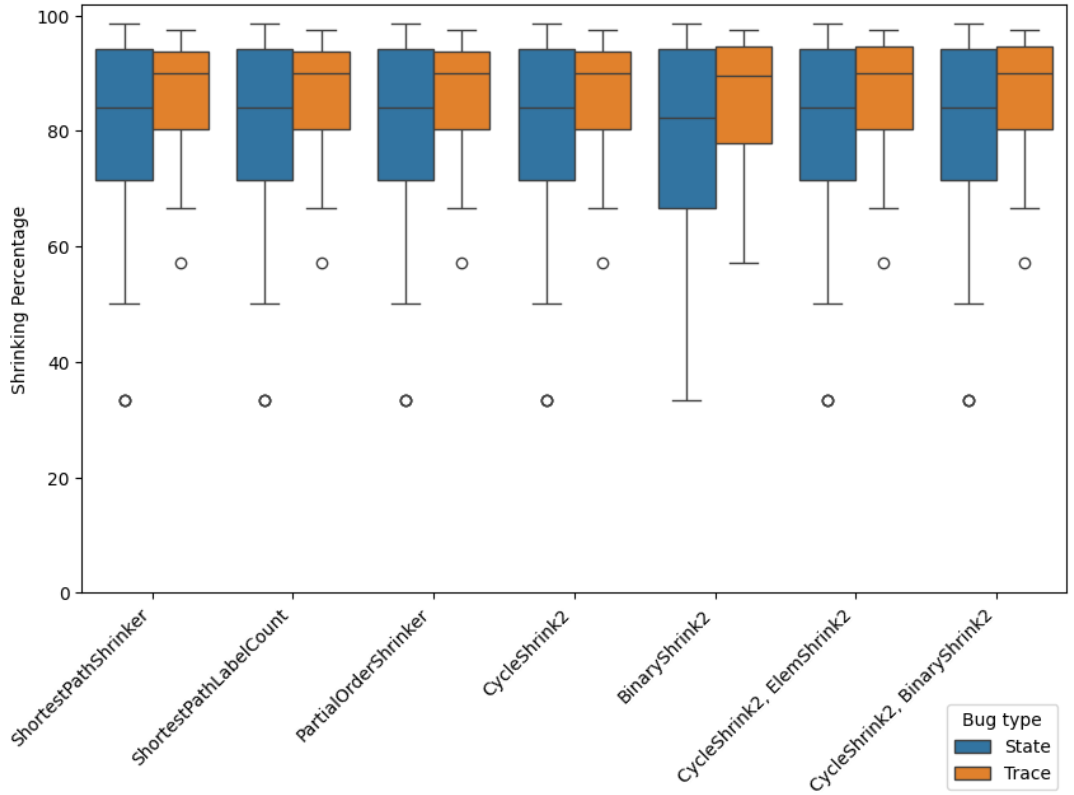
The raw data for each of the experiments can be found in Appendix A.2. The results of the experiments in which all test cases are included are shown in Appendix B.1. The statistical significance of our data can be found in Appendix B.2.

6.1 Experiment One: SmartDoor

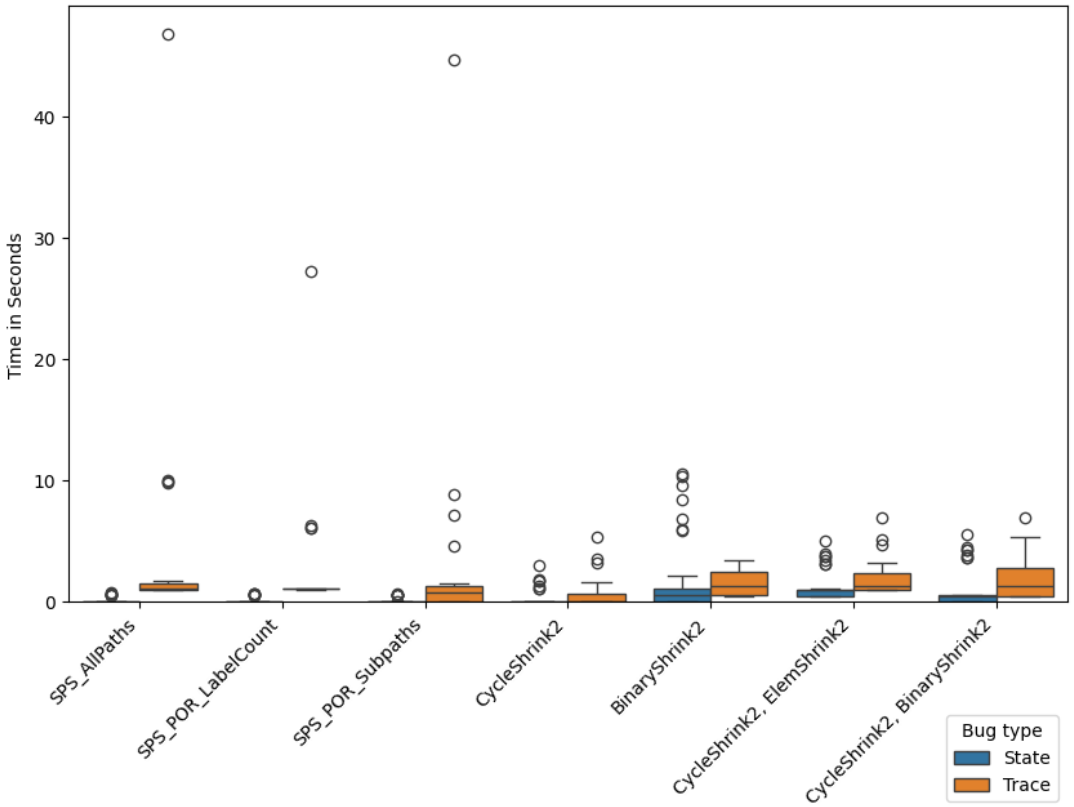
AMP produced 10 random test cases for each of the 8 different SmartDoor mutations. 7 of these test cases resulted in a pass, thus 73 different original failing traces were shrunk by the different algorithms. 6 out of the 73 original failing traces already were the shortest possible path to that failed state.

Table 6.1: SmartDoor Results - Means per Bug Type

	Test Cases per Shrink		Time per Shrink		Shrinking Percentage	
	State	Trace	State	Trace	State	Trace
SPS_AllPaths	1.00	9.75	0.09	4.74	79.22%	85.49%
SPS_POR_LabelCount	1.00	7.05	0.09	3.16	79.22%	85.49%
SPS_POR_Subpaths	1.00	8.05	0.09	3.68	79.22%	85.49%
CycleShrink2	1.42	4.50	0.23	0.84	79.22%	85.49%
BinaryShrink2	6.22	9.30	1.55	1.58	78.87%	84.36%
CycleShrink2, ElemShrink2	3.34	6.55	1.63	2.14	79.25%	85.61%
CycleShrink2, BinaryShrink2	2.98	6.60	0.79	2.04	79.25%	85.61%



(a) Shrinking Percentage per Bug Type



(b) Shrinking Time per Bug Type

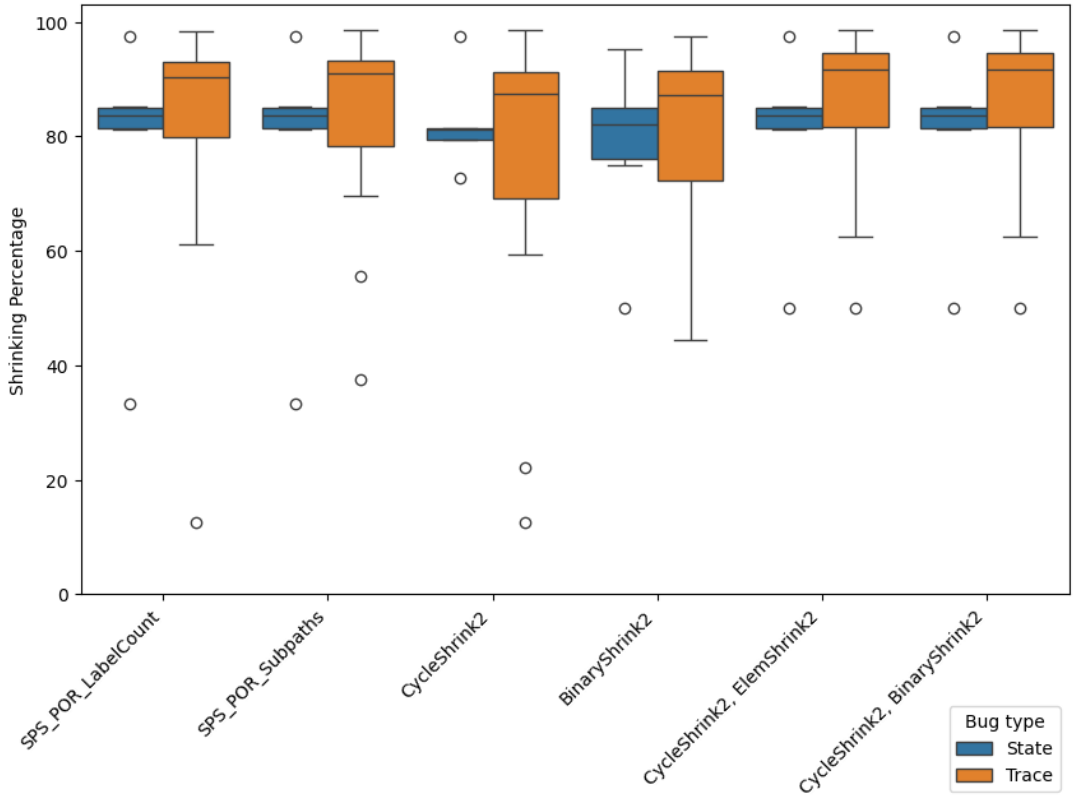
Figure 6.1: SmartDoor Results - Percentiles

6.2 Experiment Two: Koopman's Vending Machine

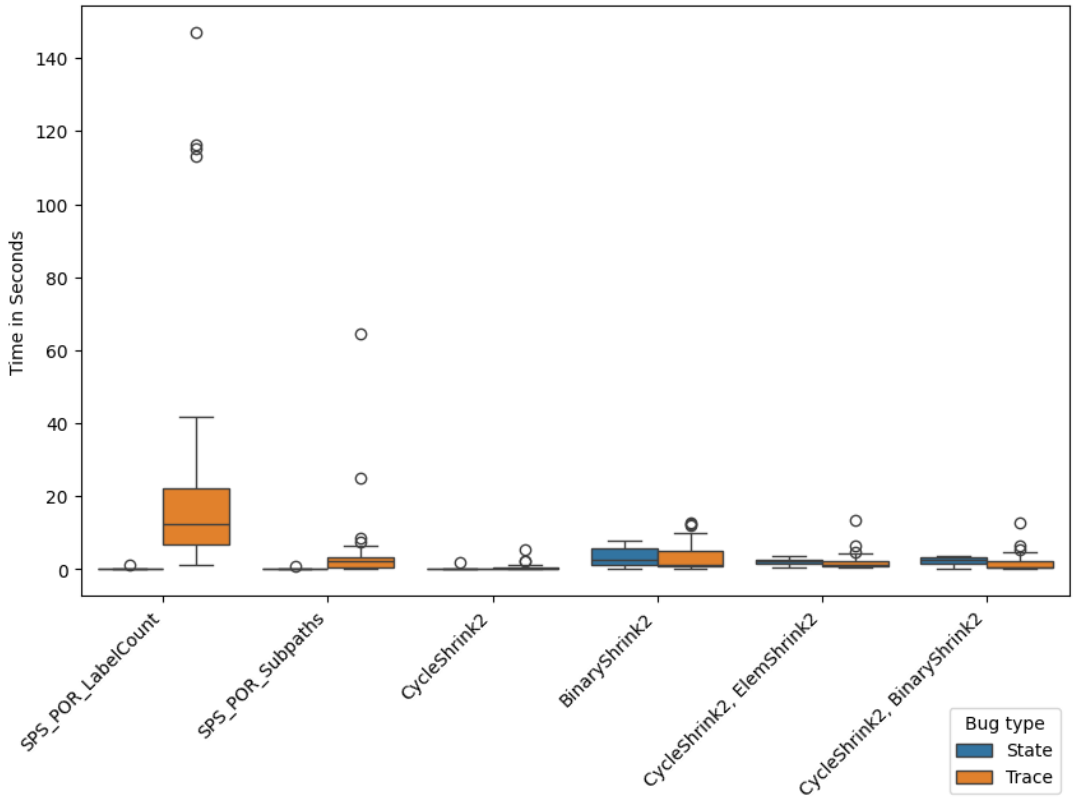
This experiment consists of many trace bugs, and the model is rather dense, meaning that there are relatively many transitions for a smaller number of states. The consequence of this is that there are many different alternative paths for our algorithms to consider. These bug and graph properties result in a larger number of test cases needed per shrink. In fact, there were so many different paths, that the runtime of *SPS_AllPaths* was too long for it to be included in this experiment, resulting in an out-of-time (o.o.t.). This is also why AMP produced only 3 instead of 10 random test cases for each of the 10 different Koopman mutations. None of these test cases resulted in a pass, thus 30 different original failing traces were shrunk by the different algorithms. All of these traces were longer than the shortest possible path to their failed state.

Table 6.2: Koopman Results - Means per Bug Type

	Test Cases per Shrink		Time per Shrink		Shrinking Percentage	
	State	Trace	State	Trace	State	Trace
SPS_AllPaths	o.o.t.	o.o.t.	o.o.t.	o.o.t.	o.o.t.	o.o.t.
SPS_POR_LabelCount	1.00	41.46	0.30	30.86	77.40%	84.27%
SPS_POR_Subpaths	1.00	11.17	0.20	5.92	77.40%	83.99%
CycleShrink2	2.33	3.17	0.42	0.67	68.71%	77.77%
BinaryShrink2	12.50	23.83	3.44	3.55	78.30%	80.75%
CycleShrink2, ElemShrink2	7.33	11.12	2.12	2.32	80.17%	85.59%
CycleShrink2, BinaryShrink2	7.50	12.71	2.35	2.04	80.17%	85.59%



(a) Shrinking Percentage per Bug Type



(b) Shrinking Time per Bug Type

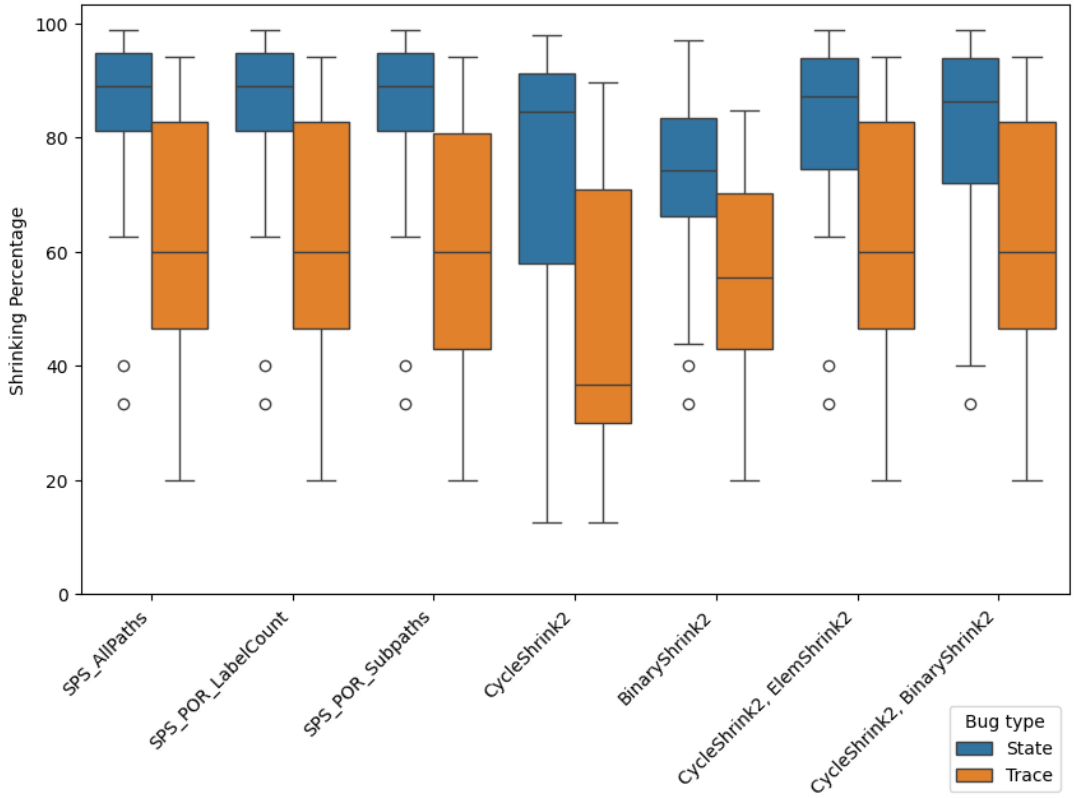
Figure 6.2: Koopman Results - Percentiles

6.3 Experiment Three: ATM

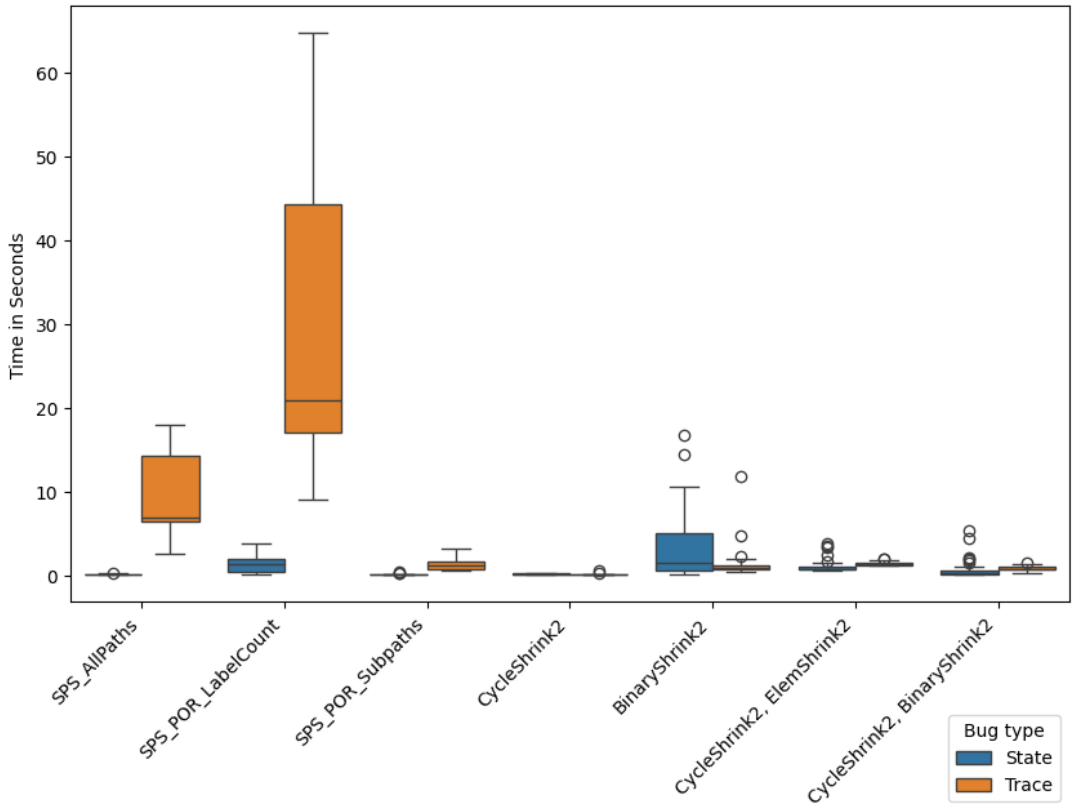
AMP produced 10 random test cases for each of the 5 different ATM mutations. None of these test cases resulted in a pass, thus 50 different original failing traces were shrunk by the different algorithms. 4 out of the 50 original failing traces already were the shortest possible path to that bug.

Table 6.3: ATM Results - Means per Bug Type

	Test Cases per Shrink		Time per Shrink		Shrinking Percentage	
	State	Trace	State	Trace	State	Trace
SPS_AllPaths	1.00	17.45	0.10	9.78	83.90%	61.94%
SPS_POR_LabelCount	1.00	17.45	1.29	30.83	83.90%	61.94%
SPS_POR_Subpaths	1.00	3.20	0.09	1.33	83.90%	59.33%
CycleShrink2	1.43	1.40	0.10	0.09	69.07%	39.81%
BinaryShrink2	20.57%	13.25	3.59	1.71	71.65%	55.38%
CycleShrink2, ElemShrink2	7.03	8.15	1.11	1.39	81.93%	61.94%
CycleShrink2, BinaryShrink2	7.03	8.85	0.77	0.78	81.27%	61.94%



(a) Shrinking Percentage per Bug Type



(b) Shrinking Time per Bug Type

Figure 6.3: ATM Results - Percentiles

Chapter 7

Discussion

In this chapter we discuss our results. We start by briefly addressing the problem modelling of test case shrinking on the shortest path problem. Then, we individually address the results of each of the three experiments. Next, we note some additional observations that were made while conducting the different experiments. We conclude this chapter by answering the research questions and by giving recommendations to industry practitioners. Threats of validity are discussed. Please note that this discussion takes into account the statistical significance of the results, found in Appendix B.2.

7.1 Modelling Shrinking to the Shortest Path Problem

As has been explained in Chapter 3, the most appropriate shortest path algorithm for our shrinking context, is k-shortest path. However, to the best of our abilities we were unable to identify a state-of-the-art k-shortest path algorithm that meets all the assumptions and requirements of that same shrinking context. We therefore created our own algorithm.

Finding 1: The shortest path problem most suited to our shrinking context, is k-shortest path. A state-of-the-art k-shortest path algorithm that meets all of our assumptions and requirements could not be identified.

7.2 Experiment One: SmartDoor

The first experiment, SmartDoor, contains the smallest and least complicated model of the three experiments in this thesis. The shrinking percentages of the model-centric shrinkers are not statistically different from those of the trace-centric shrinkers, for both state bugs and trace bugs. This means that both type shrinkers perform the same in terms of trace reduction. However, the model-centric shrinkers take a smaller number of cases and less time to shrink state bugs than the trace-centric shrinkers do. In fact, the model-centric shrinkers only require one test case in order to identify the shortest path to a state bug. This is to be expected, as per definition state bugs occur on the state-level.

Finding 2: Model-centric shrinkers only require one test case to find the shortest path to state bugs.

Meanwhile, most model-centric shrinkers take more time and more test cases than most trace-centric shrinkers to find the shortest path to trace-bugs. There is one exception: the *SPS_POR_LabelCount* algorithm takes fewer test cases than *BinaryShrink2*. It however does take more time, suggesting that its computations are more expensive.

Finding 3: Model-centric shrinkers generally require more time and more test cases to find the shortest path to trace bugs.

Finding 3 was also expected, because for trace bugs a certain specific sequence of actions needs to happen for them to show. The *SPS_AllPaths* and *SPS_POR_LabelCount* are given no guidance as

to what this sequence might be, and therefore try all possible paths until they finds the right one. The *SPS_POR_Subpaths* does have some guidance, as it prioritises transitions that were taken in the original trace. Meanwhile, trace-centric shrinking algorithms have this guidance to a higher degree, as the sought-for sequence is hidden within the original trace.

7.3 Experiment Two: Koopman's Vending Machine

The next experiment is that of Koopman *et al.*'s vending machine. This model contains a lot of loops, and many stimuli are not immediately followed by a response. This means that it can take some time to realise that an error has occurred earlier on. Because this model is very dense, and because there were mostly trace bugs instead of state bugs, the *SPS_AllPaths* was not ran on this experiment. It simply took too much time to let it find the trace bugs. The smaller sample size regretfully also means that no statistically significant difference could be found in the shrinking percentages between the model-centric and trace-centric bugs.

A remark however can still be made. We notice the hint that the chained trace-shrinkers, i.e. *CycleShrink2, ElemShrink2* and *CycleShrink2, BinaryShrink2*, have higher shrinking percentages. This is because these shrinkers can "escape" the failed state and end their trace in a different failed state that is reachable in a smaller number transitions.

> **Finding 4:** Trace-centric shrinkers can 'escape' the failing state, resulting in a higher shrinking percentage.

The escaping behaviour can be shown via the following failed trace. It was ran on m01, which is a SUT of the Koopman vending machine that remembers the product chosen after dispensing a cup. The trace is 14 steps long.
`?choice_coffee` → `?coin1` → `?go` → `?coin2` → `?choice_espresso` → `?info` → `!info_espresso_3` → `?coin2` → `?go` → `!cup_espresso` → `?coin2` → `?coin1` →`?info` → `!espresso_5`

A model-centric shrinker looking for the shortest path, will always construct its paths to the state *info_none_5*, as this is the state the model was in at the moment the error occurred. The shortest possible path to that state that contains the trace bug contains 9 steps:
`?coin2` → `?choice_coffee` → `?go` → `!cup_coffee` → `?coin2` → `?coin2` → `?coin1` → `?info` → `!espresso_5`

Meanwhile, the chained shrinkers can end up in another state by removing individual steps from the original trace. This results in a trace of only 6 steps that ends in the state *info_none_0*:
`?choice_coffee` → `?coin2` → `?go` → `!cup_coffee` → `?info` → `!coffee_0`

This behaviour is not limited to trace bugs; it can happen for state bugs as well. For instance, m03 increments the balance with 1 instead of 2 whenever a `?coin2` is inserted. If `?coin2` is inserted for the first time at a higher balance, the model-centric shrinkers will construct paths to that state, whereas trace-centric shrinkers can show the behaviour immediately without first entering `?coin1`s to increase the balance.

We also see that the *SPS_POR_Subpaths* requires a lot less test cases than the *SPS_POR_LabelCount* to find trace bugs, and consequently takes a lot less time as well.

> **Finding 5:** The *SPS_POR_Subpaths* can greatly outperform *SPS_POR_LabelCount* in terms of the number of test cases and time it needs to shrink a trace bug.

7.4 Experiment Three: ATM

The results of the ATM experiment show that all three model-centric shrinkers have a higher shrinking percentage for state bugs than the trace-centric shrinkers, except for *CycleShrink2, ElemShrink2*, which has the same shrinking percentage. For trace bugs, the shrinking percentage of the model-centric shrinkers was the same as the chained trace-centric shrinkers, and higher than that of the individual ones. This means that for state bugs, our shrinkers generally reduce the test cases more than Meijer's shrinkers did, whereas for trace bugs they reduce them to the same degree or more. We also note that the

SPS_POR_Subpaths has a smaller shrinking percentage than the other model-centric shrinkers, though this difference is not significant.

We are able to produce shorter traces because our method is not limited by the original failing trace. Meijer's shrinkers can only remove stimuli from traces, not introduce them. Meanwhile the *SPS_AllPaths* and its variants can take any transition that is present in the model. This means that if a shortcut to a state exists using a different stimulus than those that were used in the original trace, trace-centric shrinkers will not find this shorter route. The *SPS_POR_Subpaths* however, is limited in its ability to take these shortcuts because it will only add one equivalent subpath to its queue instead of all. This explains why its shrinking percentage is slightly lower than that of the *SPS_AllPaths* and of *SPS_POR_LabelCount*.

Finding 6: Model-centric shrinkers can achieve higher shrinking percentages than trace-centric shrinkers, when the model contains a shortcut to the failing state.

These short-cuts are very prominent in the model of this experiment. For instance for m01 of this experiment, the SUT omits the `insufficient_balance` check. Imagine an original trace like the following, consisting of 26 steps. It fails at state *ilde_25* at a `?withdraw_50` request. The correct response would have been `!insufficient_balance`, but instead `!50` is given:
`?insert_card_valid_300` → `!card_valid` → `?correct_PIN_300` → `!PIN_correct` → `?deposit_50` → `!balance: 350` → `?withdraw_20` → `!20` → `?balance` → `!balance: 330` → `?withdraw_100` → `!100` →`?withdraw_50` → `!50` →`?deposit_5` → `!balance: 185` → `?withdraw_100` → `!100` → `?withdraw_10` → `!10` →`?balance` → `!balance: 75` → `?withdraw_50` → `!50` → `?withdraw_50` → `!50`

A shrinker that is only able to remove parts of this original trace, can at best shrink it down to the following trace of 14 steps, a shrinking percentage of 46.15%:
`?insert_card_valid_300` → `!card_valid` → `?correct_PIN_300` → `!PIN_correct` → `?withdraw_100` → `!100` →`?withdraw_50` → `!50` → `?withdraw_100` → `!100` → `?withdraw_10` → `!10` → `?withdraw_50` → `!50`

The *SPS_POR_Subpaths* can produce this reduced trace instead, by directly inserting a card with a balance of 25. This trace has only 6 steps, a shrinking percentage of 79.92%.
`?insert_card_valid_25` → `!card_valid` → `?correct_PIN_25` → `!PIN_correct` → `?withdraw_50` → `!50`

Another interesting finding of this experiment, is the improvements that the *SPS_POR_Subpaths* can make for shrinking trace bugs in both the number of test cases and the shrinking time. The *SPS_POR_Subpaths* required less test cases to find trace bugs than three out of the four trace-centric shrinkers, *CycleShrink2* being the exception. That was also the only shrinker that could shrink in less time than the *SPS_POR_Subpaths*. This shows that reducing the search space of a k-shortest path algorithm in a guided way can really improve its results in shrinking traces. Please note that Finding 7 can only be partially be concluded from Experiment One and Two. While *SPS_POR_Subpaths* did improve the performance of the other model-centric shrinkers in these experiments, *SPS_POR_Subpaths* did not out-perform the shrinking speed and number of test cases trace-centric shrinkers.

Finding 7: The SPS_POR_Subpaths can drastically improve the performance of model-centric shrinkers for trace bugs, requiring way fewer test cases than most trace-centric shrinkers and less time than some.

We can also now confirm something that was hinted at before: the *SPS_POR_LabelCount* shrinker takes a lot more time to generate paths than the other shrinkers. It required the same number of test cases to find the shortest path to trace bugs as the *SPS_AllPaths*, but it took over three times longer. We suspect that this is due to the computations needed to determine whether a trace configuration is already present in the queue or not. This results in the *SPS_POR_LabelCount* requiring more computation time per node that it pops off the queue. On top of that, the logic behind the *SPS_POR_Subpaths* is more model-centric than that of the *SPS_POR_LabelCount*. We therefore think that the former shrinker is a more solid choice.

Finding 8: The SPS_POR_LabelCount shrinker takes more computation time than other shrinkers to generate traces.

7.5 General Observations

During the analysis of the experiments, some other observations were made. They do not belong to a experiment in particular; they are often times interesting edge cases that were observed by analysing the traces generated by the different algorithms. We list these observations here:

- As far as we can think of, every optimisation of the *SPS_AllPaths* comes with a trade-off between quality and performance. The *SPS_POR_LabelCount* algorithm for instance deems the order of labels in the trace irrelevant. This reduced the search space, but it makes this shrinker unable to catch all types of bugs, increasing its speed but reducing its quality. The *SPS_POR_Subpaths* is able to catch all bugs, but it no longer has the guarantee that the path to the bug that it finds is actually the shortest path to the failing state containing the bug. This also increases performance, but potentially decreases the quality.

  Finding 9: While partial order reductions can greatly improve the performance of a shrinker, they might do so at a slight quality decrease.

- While trace-centric shrinkers can more predictably 'escape' the failing state and find the same bug elsewhere in the model, model-centric shrinkers can achieve the same behaviour. Whenever the first shortest path does not directly lead to the error, our model-centric shrinkers will construct alternative paths to the failing state. These alternative paths can contain a subtrace that directly leads to the same error, but in another state. This happens when the same error can be encountered in different states, as has been explained earlier in this chapter. The model-centric shrinker will build a complete path to the failed state, but during the testing procedure in the MBT engine, the same error is encountered before the end of the shortest path is reached. This is the main difference between the escaping abilities of the two types of shrinkers: with trace-centric shrinkers, escaping the failed state is more predictable, while with model-centric shrinkers, it often happens by chance as they stumble upon a shorter path while testing a longer one.

  Finding 10: Model-centric shrinkers can also 'escape' the failing state, resulting in a higher shrinking percentage.

- Sometimes, the shortest path generated by the model-centric shrinkers ends up being longer than the original trace. This typically occurs when the original trace is already very short, and the SUT does not provide much feedback. By this we mean that not every stimulus is met with a response. Consequently, the model will not immediately know whether it is still in sync with the SUT or not. If the SUT's faulty implementation causes it to deviate from what the model expects after just a few stimuli, these initial errors cascade through the subsequent steps of the trace. Without proper feedback, the state of the model can deviate greatly from the actual state of the SUT. By the time the bug manifests through an ioco violating response, the failing state that the model is in, then no longer represents the actual state of the SUT. This discrepancy means that the shortest path generated to that failing state, is now less related to the bug, resulting in subsequent longer paths.

  Finding 11: Model-centric shrinkers are more effective when the SUT provides frequent feedback, preferably with a response after each stimulus.

- Throughout this research, we have experienced that model-centric shrinkers can be a valuable in the model development process. They aid in verifying the model's correctness. For instance, if the *SPS_AllPaths* consistently identifies state bugs despite the belief that the SUT is correct, it suggests potential issues with the model. The shortest path leading to such a state can provide insights into identifying and rectifying these mistakes.

Finding 12: Model-centric shrinkers can be a useful tool in the model development process.

7.6 Answering the Research Questions

In this section, we address the research questions posed in this thesis. The answers are derived from several key components: the modelling of test case shrinking to the shortest path problem, the design and implementation of the model-centric shrinkers developed as part of this research, and the results and subsequent analysis of the experiments conducted.

RQ1: How can we leverage a model-centric shortest path algorithm to shrink test cases?

The first research question is answered by the problem modelling of Chapter 3 and the design and implementation of the PoC in Chapter 4. Together these chapters describe how by utilising a k-shortest path algorithm, a model-centric shrinker can generate paths to failing states and thus find smaller counter examples by the means of an existing counter example. This way, shrinking is done in a bottom-up fashion: the traces generated by our algorithms start off as small as possible, and slowly become longer. With traditional trace-centric shrinkers, the original trace becomes smaller and smaller with each iteration of the algorithm. Because the *SPS_AllPaths* uses BFS, and because there was initially no guidance given for finding trace bugs, state explosion occurs and many attemps are needed to find trace bugs. To tackle these issues, two variants of the *SPS_AllPaths* were developed. These are the *SPS_POR_LabelCount* and the *SPS_POR_Subpaths*. They each work by reducing the search space of the algorithm. The model-centric shrinkers were integrated into an existing shrinking framework that is incorporated to a commercial MBT suite, AMP.

RQ2: How does the proposed model-centric shrinking method compare to traditional trace-centric methods?

Our model-centric shrinking algorithm can be evaluated in terms of quality and performance. For quality, we look at the shrinking percentage, whereas for performance we look at shrinking time and the number of test cases per shrink. The analysis of the results as described in the previous sections of this chapter, show that the evaluation is a nuanced picture.

On the one hand, our model-centric shrinkers have higher quality for state bugs, as they always find these in their first iteration. In other words, our shrinkers require less test cases and less time to shrink state bugs than Meijer's shrinkers. The quality of the model-centric shrinkers for this type of bug is also higher, or at the same level. This is because the shrinking percentage was higher, or the difference with trace-shrinkers was not statistically significant. The difference in quality is especially prominent when the model contains shortcuts to the failed state.

However, when it comes to trace bugs, the performance of the *SPS_AllPaths*; it takes this algorithm more attempts and therefore also more time to find this type of bug. This is due to the fact that this method always builds *all* paths to the failed state until the correct one is found, suffers from this. Two variants of this shrinker were developed in order to mitigate this problem, of which the *SPS_POR_Subpaths* does so most successfully. This shrinker is able to reduce the search space effectively, resulting in a lot fewer test cases and therefore also less time spent shrinking. In the ATM experiment, *SPS_POR_Subpaths* even has better performance and the same quality than three out of four trace-centric shrinkers.

7.7 Industry Recommendations

Industry practitioners might wonder what type of shrinker they could use best in their unique situation. In this regard, we make the following remarks:

- **The choice between a model-centric or a trace-centric shrinker depends on the type of bug.** Model-centric shrinkers excel when shrinking state bugs, while their performance staggers when dealing with trace bugs. A model-centric shrinker could however be utilised in order to figure

out the nature of the bug: if such a shrinker takes more than one test case to shrink a trace, you know for sure that you are dealing with a trace bug. Trace-centric shrinkers cannot give you that information. Knowing the type of bug can however be valuable information when debugging. Additionaly, model-centric shrinker can be a useful tool in the model development process, as they can aid in verifying the model's correctness.

- **The choice between a model-centric or a trace-centric shrinker depends on the model.** Some models are more suited for model-centric shrinkers, whereas on others a trace-centric shrinker will work better. The *SPS_AllPaths* could not be ran on Koopman's vending machine, because its model is too dense to find its trace bugs. This issue can be mitigated in part by using search space heuristics such as the ones of the *SPS_POR_LabelCount* and the *SPS_POR_Subpaths*. On the other hand, the characteristics of the ATM model made it a good candidate for model-centric shrinkers, as it presented these algorithms with a lot of shortcuts to failed states.
- **The choice between a model-centric or a trace-centric shrinker depends on the trade-off between quality and performance.** In the end, it depends on what the user of the MBT platform finds most important. If speed is of the essence and not much is known about the bug, then a trace-centric shrinker might be the safest choice. The *SPS_POR_Subpaths* is also a valid option when opting for performance. Otherwise, if testing against the SUT can be done quickly and the developer most of all wants a short trace to help with debugging, then model-centric shrinkers can be the better choice. Our results show that model-centric shrinkers never deliver lower quality than trace-centric ones; in the worst case they shrink with the same rate.

7.8 Threats to Validity

Validity refers to the degree to which a concept, conclusion, or measurement accurately reflects or corresponds to the real world. In other words, it helps determine how trustworthy and useful a study's findings are. High validity ensures that the results genuinely represent the phenomena being studied, allowing researchers and practitioners to have confidence in the study's conclusions and their applicability. In this section, we discuss the potential threats to the validity of this thesis. We outline the steps taken to mitigate these threats where possible, and we acknowledge the limitations that remain despite our efforts.

7.8.1 Internal Validity

Internal validity ensures that the results of a study are due to the variables tested and not other factors; it's about the confidence in the cause-effect relationship of the study. During the problem modelling of test case shrinking to the shortest path problem, we could not identify a state-of-the-art k-shortest path solution that meets all of our assumptions and requirements. This however does not mean that such an algorithm does not exist. Factors such as researcher bias and selection bias might have prevented us from finding the right algorithm for our shrinking context. To mitigate these concerns, we adopted a systematic approach in our literature search and selection process. This included utilizing various databases and employing snowballing techniques to discover new papers and expand our search terms.

Another threat to the internal validity of this thesis, is the lack of statistical power of Experiment Two. Due to the characteristics of both the model and the type of bugs, rendering results for this experiment took significantly more time than for the other experiments. Consequently, it was decided that AMP should generate three original traces per bug, instead of the ten original traces used in the other experiments. This smaller sample size makes it harder to draw definitive conclusions from this experiment. Despite these constraints, we included this experiment because it is a significant component in related literature, such as the works of Koopman and Meijer.9

7.8.2 External Validity

External validity refers to the extent to which the findings of a study can be generalised or applied to settings, populations, or times beyond the specific conditions of the study. It assesses whether the results of a study are applicable to real-world scenarios and whether they hold true outside the controlled environment of the research. This relates to our selection of experiments. As previously discussed in this chapter, the effectiveness of shrinking algorithms is heavily influenced by the characteristics of the model and the types of bugs present in the SUT. To provide a comprehensive overview, we conducted three distinct experiments using different models and SUTs, each contributing unique insights.

Experiment One, involving the SmartDoor model, featured a simple structure with straightforward bugs. Experiment Two utilised Koopman's vending machine model, which is dense and predominantly contains trace bugs. Here, our expectation was that trace-centric shrinking algorithms would outperform model-centric approaches. In response, we designed Experiment Three, centered around an ATM model characterised by numerous shortcuts, showcasing the strengths of model-centric shrinkers.

While we deliberately chose models and SUTs that cover a broad range of complexities to encompass various software testing scenarios, the applicability of our results to real-world systems remains uncertain. The SUTs used in this thesis are synthetic—they are small Ruby programs designed to simulate the behaviour of their respective systems. For instance, the SmartDoor SUT is not an actual physical door but a program emulating its functionality. Simulations, by nature, may simplify or abstract away complexities found in real-world systems. Therefore, the behaviour observed in our experiments might differ from what would occur in practical applications. Given the time constraints of this thesis, we focused on synthetic models and SUTs, and analyzing real-world systems was beyond our scope. This aspect remains an area for future investigation. Due to the varying outcomes observed across different models and SUTs, the general applicability of our findings remains uncertain. To address this, we intentionally selected models and SUTs that span a wide spectrum of complexities, aiming to capture the diversity of scenarios encountered in software testing.

7.8.3 Construct Validity

Because we could not identify a suitable k-shortest path algorithm, we designed one ourselves. Due to time constraints, formally proofing the correctness of this algorithm is out-of-scope for this thesis. The lack of a formal proof does however impose a threat to the construct validity of this research. Construct validity refers to whether a method accurately measures what it claims to measure; whether the algorithm accurately performs shrinking as intended. To evaluate its effectiveness, we tested our algorithm against a benchmark with known correctness. These tests were ran on the SUTs and models of the three experiments of this thesis, and our algorithm produced results consistent with the benchmark. Nevertheless, without a formal proof, uncertainties remain regarding its generalizability across different datasets and conditions.

Another threat to construct validity is the choice of trace-centric algorithms, as they are the tools used to measure the effectiveness of our model-centris shrinkers. To address this concern, we draw upon the work of Meijer [6], who implemented 11 different trace-centric shrinkers sourced from various studies such as those by Koopman [5] and Zeller [17]. From these 11 shrinkers, we carefully selected four that demonstrated the best balance between quality and performance. Based on this selection process, we are confident in the validity of our chosen trace-centric shrinkers for comparison. They represent a considerable subset of options known for their effectiveness, as established in prior research.

Chapter 8

Related work

This chapter provides a review of literature that is closely related and therefore relevant to this thesis. We divide the related work in the following categories: shortest path in MBT, graph traversal in model checking, and partial order reduction in model checking.

8.1 Shortest path in Model-Based Testing

Using shortest path algorithms in MBT is not an entirely new concept. Microsoft for instance uses shortest path for test case generation [34]. It is one of the test query types that their tool offers: *Shortest Path* allows testers to specify where they want their test cases to begin and end, and returns the shortest path between those two points. *All Shortest Paths* finds all the possible ways to begin and end a test case, and returns the shortest paths between all of these points. Shortest path has also been researched in the context of test case generation in MBT (e.g. [35], [36] and [37]). The approaches taken by Nguyen and Pretschner both use BFS to compute the shortest path [35, 37].

The difference between these papers and the proposed thesis, is that they apply shortest path at the test case *generation* phase, and not at test case *analysis* and *bug localisation*. A question that could arise, is the following: *If test cases are generated using shortest path, then how come can they still be shrunk (using that exact same approach)*? The answer lies in that not all test traces generated by MBT, use shortest path. In fact, shortest path can only be utilised when the destination of the path is known. This is only the case when a specific requirement is being tested. This is not the power of MBT; the power of MBT is the automated test coverage. Axini for instance does not use shortest path strategies to generate tests; their strategies are more unguided or random. Because of this randomness, test cases become longer and shrinking is possible.

8.2 Graph Traversal in Model Checking

Our proposed shrinking method is similar to model checking, in the sense that both are looking for counter examples within a model by traversing it systematically. Model checking is a formal verification technique used to systematically explore the state space of a system model to ensure that it satisfies certain correctness properties. A survey on directed model checking was done by Edelkamp *et al.* [38]. Directed model checking is a bug hunting technique that mitigates state explosion. In this survey different algorithms to achieve directed model checking are discussed. Wijs [39] proposed an on-the-fly BFS-based method to check the liveness properties of a model. It does so by detecting infinite cycles in the model, which is usually a Depth-First-Search (DFS) job. However, because of the parallelisation of GPU's, Wijs deems BFS a better candidate in this case than DFS. Wu *et al.* [40] also proposed a GPU based solution to speeding up BFS in model checking. They looked at counter example generation in Linear Temporal Logic and utilise dynamic parallelism and memory hierarchy to handle the irregular searching pattern in BFS, which accelerates the process. Using GPU's for faster results is out of scope for this thesis, but could be interesting future work. Similarly to our research, Aljazzar and Leue [41] used a k-shortest path solution to find candidate counter examples. They used an adaptation of Eppstein's algorithm and applied it to continuous- and discrete-time Markov Chains. The diagnostic information collected by their algorithm is helpful in system debugging.

8.3 Partial Order Reduction in Model Checking

Partial order reduction (POR) techniques, first proposed by Godefroid [42], are based on the observation that in concurrent systems, many interleavings of independent actions lead to the same state. By identifying and exploring only a representative subset of these interleavings, model checkers can significantly reduce the state space they need to explore without compromising the verification results. In model checking, just like in our model-centric shrinking algorithms, POR helps mitigate the state explosion problem. Model checkers frequently use a Depth-First-Search (DFS) traversal strategy, leading to many POR strategies being designed for DFS. However, it has been verified [43] that POR is also applicable to the BFS traversal strategy used in this thesis. Flanagan and Godefroid [44] introduced dynamic POR which, unlike traditional POR methods that rely on static analysis, identifies independent actions on-the-fly. This dynamic approach, unlike the method presented in this thesis, was created for state spaces without cycles. Peled's [45] POR created an addition to the model checker SPIN that utilises POR in on-the-fly model checking. Similarly, the POR variants of *SPS_AllPaths* also work on-the-fly. Lluch-Laduente *et al.* [46] applied POR to directed model checking, and found that it can be an effective way to reduce the state space while reducing the runtime despite introducing more overhead. They however also recognise that applying POR can harm the solution quality. By this they mean that POR does not preserve optimiality of the shortest path to a target state. A similar remark has been made in this thesis about the *SPS_POR_Subpaths* algorithm.

Chapter 9

Conclusion

In this thesis, we have introduced and demonstrated a proof-of-concept for a *model-centric* approach to test case shrinking in Model-Based Testing (MBT). Our method, termed the *SPS_AllPaths*, utilises Breadth-First Search (BFS) to identify the shortest path to a failing state. It also explores alternative paths when the shortest path alone does not uncover the bug. This algorithm was developed because existing state-of-the-art k-shortest path algorithms did not fully meet the assumptions and requirements specific to this thesis.

Additionally, we implemented two variants of the *SPS_AllPaths*: *SPS_POR_LabelCount* and *SPS_POR_Subpaths*. These variants incorporate different search space heuristics, which can enhance performance but may trade off some quality aspects. Our experimental results underscore the effectiveness of model-centric shrinking. Across all evaluated metrics, our approach consistently matches or surpasses existing methods when shrinking traces to isolate state bugs. For trace bugs, our algorithms also achieve comparable or better reduction rates compared to trace-centric shrinkers, albeit with varying speeds depending on the specific shrinker used.

Overall, the findings suggest that the model-centric shrinking methods as proposed in this thesis, presents a viable alternative to traditional trace-centric approaches. It offers promising capabilities for enhancing the efficiency and effectiveness of test case minimization in MBT scenarios. This advancement is valuable for enhancing bug analysis and localization efforts within software testing.

9.1 Future Work

Looking ahead from our current findings on model-centric shrinking methods, several avenues for future research and development emerge. In this section, we explore potential opportunities and directions for enhancing our work, as well as developing innovative approaches to address existing challenges.

9.1.1 Improving our Model-Centric Shrinkers

We can think of several ways to improve the three model-centric shrinking algorithms suggested in this thesis. For instance, in the current implementation of the algorithms, queue elements hold the entire path to a node. A common optimization in search algorithms is to store only the path to the parent of a node, with each parent pointing to its predecessor. This approach can significantly reduce memory usage. *SPS_POR_LabelCount* can reduce also improve its memory usage by incorporating a different way to store path configurations, for instance by implementing the configuration as a string instead of a dictionary. The configuration `configuration: "A": 1, "?B": 3, "?C": 10`, would then be stored as `"?A1?B3?C10"`, and pattern-matching could be used to modify it. Of course many other canonical values are possible to represent the path configuration.

SPS_POR_Subpaths can be improved by first checking if a node contains an original transition, and then only add those and never the non-original ones. This saves time by not needing to modify the queue later by replacing the non-original transition by the original one. The algorithms could also be improved by switching its behaviour after the first iteration. If the shrinker requires more than one iteration (i.e. more than one test case) to find the bug, this means that the bug is a trace bug rather than a state bug. In this case, one could opt to only explore paths with original transitions, and never add non-original transitions to the queue. This will likely improve the performance of the algorithm in terms of shrinking time and the number of test cases needed. However, it also leads to a potential reduction of the solution

qualitt, as explained in Finding 9 and by Lluch-Laduente *et al.* [46].

The buckets of the `main_queue` that hold the actual queues could also be sorted in such a way that paths that are more likely to contain the fault, are placed at the front. If done successfully, this would reduce the number of test cases needed to reproduce the error, and with that also reduce the shrinking time. For instance, *SPS_POR_Subpaths* could place nodes that were reached with an original transition at the front. Another way in which the queues can be sorted is by how many of the transitions of a path are original transitions, placing those paths with the most original transitions at the front. Please note that the `main_queue` will still be processed based on its `path_size`, and that the re-ordering happens at the `current_queue`-level, and at `k_queue[path_size]`-level.

Additionally, a different k-shortest path algorithm could be developed that meets all of the requirements and assumptions mentioned in Chapter 3. Most state-of-the-art k-shortest path algorithms utilise clever datastructures that save on both space and time complexity, such as path trees. We did not implement this, because our goal was not to create a new k-shortest path algorithm, it was to use shortest paths as a way to shrink cases. Future research could aim at developing such an improvement to our algorithm.

9.1.2 Analysing and Testing our Model-Centric Shrinkers

As our modelling of shrinking test cases to the shortest path problem showed, no suitable k-shortest path algorithm could be identified. Due to this, we developed our on algorithm. However, its correctness has not been formally proven or validated. This would be excellent future work to strengthen the position of this research. The current algorithms also lack a thorough theoretical analysis of their performance. Future work could determine the time and space complexity using Big O notation.

Since our thesis focused on synthetic models rather than real-world systems, we recommend exploring these areas in future research. Real-world systems are inherently more complex, potentially highlighting even more nuanced differences between model-centric and trace-centric shrinkers. Additionally, conducting experiments on SUT that contain multiple bugs would be beneficial. In our experiments, each mutation of the SUT contained exactly one error. However, real-world systems often have multiple bugs concurrently. Understanding how this scenario impacts the effectiveness of model-centric versus trace-centric shrinkers, and clarifying the distinctions between them, remains an open question.

We also suggest conducting user-testing with the two different type of shrinkers to evaluate their practical effectiveness, for instance by means of case studies. This would involve observing how the shrinkers assist users during the debugging process in real-world scenarios. By doing so, researchers can gather valuable insights into the usability and impact of shrinkers on improving debugging efficiency and accuracy.

9.1.3 New model-Centric Shrinkers

Before settling on the k-shortest path algorithm, we also investigated how the principles of loop-unrolling could be applied to shrink test cases in a model-centric manner. Loop unrolling is a technique used by compilers to enhance the performance of loops in software programs. It achieves this by replacing the loop's control structure with a sequence of repeated loop body instructions. Instead of executing the loop's body once per iteration, loop unrolling executes multiple copies of the loop body per iteration. This reduces the overhead associated with loop control and provides more opportunities for optimization [47]. By replacing the loop body with iterations of the same code, loop unrolling essentially flattens out loops. This is also done by Lee and Ling [48] in their study on *cycle unrolling*, in which they flatten finite cycles to create acyclic graphs from cyclic ones.

Loop unrolling is similar to the problem of the our shrinking approach, in the sense that both aim to find the optimal number of repetitions. In the case of loop unrolling, the number of repetitions (i.e. the unrolling factor) should be large enough to optimise performance by reducing loop control overhead, but also small enough to bloat code size or introduce inefficiencies (e.g. in registers). Determining the optimal loop unrolling factor has been intensively studied (e.g. [49], [50], [51], [52], [53]). In our shrinking problem, we aim to expand the shortest path with enough iterations of loops and/or detours in order to find the error, while keeping the number of unnecessary transitions as minimal as possible. A model-centric approach to shrinking that utilises loop-unrolling principles could therefore yield interesting results.

Another challenge we face is dealing with parametrised information, specifically variables within behavioural models like the STS. Unlike the LTS used in our thesis, which contains no variables and

thus represents the entire search space with its structure alone, models with variables create additional complexities. For instance, a path in an STS where a variable is set to 5 differs from a path where the same variable is set to 7. Future research needs to explore how model-centric shrinking algorithms can effectively navigate these challenges. One potential approach could involve flattening the model, where unique states are created for each distinct configuration of parameters. This approach would simplify the path-building process by ensuring each unique parameter configuration is explicitly represented as a state in the model. Future work could additionally aim to remove our assumption of a deterministic SUT. This would involve developing methods and techniques to handle nondeterministic behaviours, thereby broadening the applicability and robustness of the testing framework.

We also see potential in a hybrid shrinker, that combines elements of the proposed model-centric shrinker and traditional trace-centric shrinkers. A shrinker could for instance first produce the shortest path to the failing state using BFS. If this trace does not directly reproduce the bug, then we are dealing with a trace bug instead of a state bug, and a trace-based approach will likely be faster. Furthermore, trace-centric shrinkers could be chained with model-centric shrinkers. This can prove fruitful when the shortest path found by the model-centric shrinker, is not the theoretical shortest path. Our shrinkers always remain at the mercy of the original failing trace. This trace determines the failing state, and with that also the shortest path. Algorithms like *ElemShrink2* and *BinaryShrink2* can end up in different failing states, which may result in smaller shortest paths. A model-centric shrinker can never purposely 'escape' the failing state, but a trace-centric one can.

9.1.4 Adding Learning to Model-Centric Shrinking

Another research direction would be to see if one could let the model-centric shrinker learn from the traces that did not lead to the bug. The shrinker could for instance prioritise paths with a low similarity to those paths that did not lead to the bug. This approach would be especially interesting in an asynchronous shrinking framework. The shrinker could produce paths while waiting on the feedback from the MBT engine. Based on some heuristic or feedback mechanism, the shrinker could put the new paths in a priority queue, placing those it deems most likely to contain the bug in the front.

Alternatively, Machine Learning (ML) could be utilised to create a system capable of shrinking test traces using information from the MBT model. This resembles the work of Dinella *et al.* [54], who proposed a ML technique that transforms a faulty software program into a non-faulty one. They represent the program as a graph, on which the ML technique then performs graph transformations, such as adding, removing or changing graph nodes. These transformations eventually lead to a non-faulty graph, meaning that the bugs in the program have been resolved. In the case of shrinking, the graph of the fauly program would be that of the original trace, and the non-fauly graph would be the shrunk trace. The MBT model would be used as input.

9.1.5 k-Shortest Path for Model Correctness

Lastly, we recognise different purposes for the k-shortest path algorithm in MBT. Like Dijkstra famously said, "*program testing can be used to show the presence of bugs, but never to show their absence.*" [55]. However, the *SPS_AllPaths* can be used to assess model correctness and potentially also SUT correcetness. Similar to in model checking, a k-shortest path algorithm could traverse to all nodes of the model and generate paths up until a length of k. If none of these paths contain an error, then we know that the nodes of the model contain no state bugs, and that no trace bugs exists with paths of length k.

Acknowledgements

A couple of years ago, I would never have guessed that I would end up writing my Master's thesis on this topic, let alone that it would lead to a published paper. I began this journey as a Business Administration student, and although I enjoyed the subjects, I found myself yearning for something more. This led me to choose Programming as my minor, a decision that sparked a newfound passion and allowed me to learn so much in such a short time. My final project for that programming minor was on train route scheduling, which involved graph traversal to find optimal paths using heuristics like a greedy lookahead. It's quite amusing to think that the project which introduced me to the world of Computer Science is so closely related to the one that concludes my academic journey in the field.

I want to express my gratitude towards my supervisors, Ana and Theo. Ana, your endless enthusiasm made this whole process a lot more enjoyable and motivating. Theo, I really appreciate your incredible attention to detail and all the time you spent reviewing the DEBT paper and meeting with me, even during your free time. I still remember how, even while we were both out of office and on holiday, you kept sending me a stream of emails with suggestions for future work.

I also want to thank the people at Axini, for welcoming me with open arms and providing me with the tools I needed to succeed. A special thanks to Lars for helping me get started and giving me access to his codebase, and to Aleandro, who was always ready to lend a hand. Without Machiel's enthusiasm during the UvA Thesis Fair I never would have choses this subject in the first place.

Finally, I want to thank my friends and family for making these years at uni so much more fun and interesting. Your support is greatly appreciated.

Bibliography

[1] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. Elsevier, 2007, ISBN: 9780123725011. DOI: 10.1016/b978-0-12-372501-1.x5000-5. [Online]. Available: http://dx.doi.org/10.1016/b978-0-12-372501-1.x5000-5.

[2] A. Pretschner *et al.*, "One evaluation of model-based testing and its automation," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05, St. Louis, MO, USA: Association for Computing Machinery, 2005, pp. 392–401, ISBN: 1581139632. DOI: 10.1145/1062455.1062529. [Online]. Available: https://doi.org/10.1145/1062455.1062529.

[3] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016. DOI: 10.1109/TSE.2016.2521368.

[4] I. Zayour, C. Mavromoustakis, B. El-Hajj-Diab, and A. Rahil, "Towards an understanding of the causes of difficulties in debugging," in *2018 International Conference on Computer and Applications (ICCA)*, 2018, pp. 383–389. DOI: 10.1109/COMAPP.2018.8460381.

[5] P. Koopman, P. Achten, and R. Plasmeijer, "Model-based shrinking for state-based testing," in *Trends in Functional Programming*, J. McCarthy, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 107–124, ISBN: 978-3-642-45340-3. DOI: https://doi.org/10.1007/978-3-642-45340-3_7.

[6] L. Meijer, "Test case shrinking for model based testing on symbolic transition systems," M.S. thesis, Feb. 2021. [Online]. Available: http://essay.utwente.nl/85919/.

[7] T. Ruys, *Introduction to model based testing and the axini modeling language*, Axini B.V. Oct. 2021. [Online]. Available: https://www.axini.com.

[8] G. Tretmans, "A formal approach to conformance testing," English, PhD Thesis - Research UT, graduation UT, University of Twente, Netherlands, 1992, ISBN: 90-90-05643-2. DOI: 10.3990/1.9789090056432.

[9] J. Tretmans, "Model based testing with labelled transition systems," in *Formal Methods and Testing*, ser. Lecture Notes in Computer Science (LNCS), R. M. Hierons, J. P. Bowen, and M. Harman, Eds., vol. 4949, Embedded Systems Institute, Eindhoven, and Radboud University, Nijmegen, The Netherlands, Springer-Verlag, 2008, pp. 1–38. DOI: 10.1007/978-3-540-78917-8_1.

[10] R. Gorrieri, "Labeled transition systems," in *Process Algebras for Petri Nets: The Alphabetization of Distributed Systems*. Cham: Springer International Publishing, 2017, pp. 15–34, ISBN: 978-3-319-55559-1. DOI: 10.1007/978-3-319-55559-1_2. [Online]. Available: https://doi.org/10.1007/978-3-319-55559-1_2.

[11] T. Ahmad, J. Iqbal, A. Ashraf, D. Truscan, and I. Porres, "Model-based testing using uml activity diagrams: A systematic mapping study," *Computer Science Review*, vol. 33, pp. 98–112, 2019, ISSN: 1574-0137. DOI: https://doi.org/10.1016/j.cosrev.2019.07.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1574013718302314.

[12] H.-K. MIAO, S.-B. CHEN, and H.-W. ZENG, "Model-based testing for web applications: Model-based testing for web applications," *Chinese Journal of Computers*, vol. 34, no. 6, pp. 1012–1028, Sep. 2011, ISSN: 0254-4164. DOI: 10.3724/sp.j.1016.2011.01012. [Online]. Available: http://dx.doi.org/10.3724/sp.j.1016.2011.01012.

[13] T. Kanstrén and M. Chechik, "Trace reduction and pattern analysis to assist debugging in model-based testing," in *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, IEEE, 2014, pp. 1–11. DOI: 10.1109/ISSREW.2014.9.

[14] M. Willemsen, "Improving diagnosis by grouping test cases to reduce complexity," M.S. thesis, Jul. 2018. [Online]. Available: http://essay.utwente.nl/75676/.

[15] M. Weiglhofer, G. Fraser, and F. Wotawa, "Using spectrum-based fault localization for test case grouping," in *2009 IEEE/ACM International Conference on Automated Software Engineering*, 2009, pp. 630–634. DOI: 10.1109/ASE.2009.78.

[16] A. Zeller, "Yesterday, my program worked. today, it does not. why?" In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-7, Toulouse, France: Springer-Verlag, 1999, pp. 253–267, ISBN: 3540665382.

[17] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002. DOI: 10.1109/32.988498.

[18] A. Madkour, W. G. Aref, F. U. Rehman, M. A. Rahman, and S. Basalamah, "A survey of shortest-path algorithms," *arXiv preprint arXiv:1705.02044*, 2017. DOI: https://doi.org/10.48550/arXiv.1705.02044.

[19] K. Mohanta, B. Poddar, *et al.*, "Comprehensive study on computational methods for k-shortest paths problem," *International Journal of Computer Applications*, vol. 40, no. 14, pp. 22–26, 2012.

[20] A. W. Brander, M. C. Sinclair, *et al.*, "A comparative study of k-shortest path algorithms," in *Proc. of 11th UK Performance Engineering Workshop*, Springer, 1995, pp. 370–379.

[21] M. Pollack, "Solutions of the kth best route through a network—a review," *Journal of Mathematical Analysis and Applications*, vol. 3, no. 3, pp. 547–559, 1961.

[22] L. Roditty and U. Zwick, "Replacement paths and k simple shortest paths in unweighted directed graphs," in *Automata, Languages and Programming*, L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, and M. Yung, Eds., Springer Berlin Heidelberg, 2005, pp. 249–260, ISBN: 978-3-540-31691-6. DOI: https://doi.org/10.1007/11523468_21.

[23] J. Y. Yen, "Finding the k shortest loopless paths in a network," *management Science*, vol. 17, no. 11, pp. 712–716, 1971, ISSN: 1526-5501. DOI: 10.1287/mnsc.17.11.712. [Online]. Available: http://dx.doi.org/10.1287/mnsc.17.11.712.

[24] V. M. Jiménez and A. Marzal, "A lazy version of eppstein's k shortest paths algorithm," in *Experimental and Efficient Algorithms*, K. Jansen, M. Margraf, M. Mastrolilli, and J. D. P. Rolim, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 179–191, ISBN: 978-3-540-44867-9. DOI: https://doi.org/10.1007/3-540-44867-5_14.

[25] D. Eppstein, "Finding the k shortest paths," *SIAM Journal on Computing*, vol. 28, no. 2, pp. 652–673, 1998. DOI: 10.1137/S0097539795290477. [Online]. Available: https://doi.org/10.1137/S0097539795290477.

[26] W. Hoffman and R. Pavley, "A method for the solution of the nth best path problem," *J. ACM*, vol. 6, no. 4, pp. 506–514, Oct. 1959, ISSN: 0004-5411. DOI: 10.1145/320998.321004. [Online]. Available: https://doi.org/10.1145/320998.321004.

[27] R. Bellman and R. Kalaba, "On kth best policies," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 4, pp. 582–588, 1960. DOI: 10.1137/0108044. [Online]. Available: https://doi.org/10.1137/0108044.

[28] M. Sakarovitch, "The k shortest routes and the k shortest chains in a graph," Operations Research Center, University of California, Berkeley, Report ORC-32, 1966. DOI: 10.21236/ad0642823.

[29] D. R. Shier, "Iterative methods for determining the k shortest paths in a network," *Networks*, vol. 6, no. 3, pp. 205–229, 1976. DOI: https://doi.org/10.1002/net.3230060303. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/net.3230060303. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/net.3230060303.

[30] E. de Queirós Vieira Martins, "An algorithm for ranking paths that may contain cycles," *European Journal of Operational Research*, vol. 18, no. 1, pp. 123–130, 1984, ISSN: 0377-2217. DOI: https://doi.org/10.1016/0377-2217(84)90269-8. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0377221784902698.

[31] S. S. Skiena, *The Algorithm Design Manual*, 3rd. Springer, 2020, ISBN: 978-3-030-54255-9. DOI: 10.1007/978-3-030-54256-6.

[32] G. J. Holzmann and D. Peled, "An improvement in formal verification," in *Formal Description Techniques VII: Proceedings of the 7th IFIP WG 6.1 international conference on formal description techniques*, D. Hogrefe and S. Leue, Eds. Boston, MA: Springer US, 1995, pp. 197–211, ISBN: 978-0-387-34878-0. DOI: 10.1007/978-0-387-34878-0_13. [Online]. Available: https://doi.org/10.1007/978-0-387-34878-0_13.

[33] E. M. Clarke, O. Grumberg, M. Minea, and D. Peled, "State space reduction using partial order techniques," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 3, pp. 279–287, Nov. 1999, ISSN: 1433-2779. DOI: 10.1007/s100090050035. [Online]. Available: https://doi.org/10.1007/s100090050035.

[34] K. Stobie, "Model based testing in practice at microsoft," *Electronic Notes in Theoretical Computer Science*, vol. 111, pp. 5–12, 2005, Proceedings of the Workshop on Model Based Testing (MBT 2004), ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2004.12.004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1571066104052296.

[35] A. Pretschner, O. Slotosch, E. Aiglstorfer, and S. Kriebel, "Model-based testing for real: The inhouse card case study," *International Journal on Software Tools for Technology Transfer*, vol. 5, pp. 140–157, 2004. DOI: 10.1007/s10009-003-0128-3.

[36] H. Takagi, S. Saito, K. Fukuda, and C. Asakawa, "Analysis of navigability of web applications for improving blind usability," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 14, no. 3, 13–es, 2007.

[37] C. D. Nguyen, A. Marchetto, and P. Tonella, "Combining model-based and combinatorial testing for effective test case generation," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012, Minneapolis, MN, USA: Association for Computing Machinery, 2012, pp. 100–110, ISBN: 9781450314541. DOI: 10.1145/2338965.2336765. [Online]. Available: https://doi.org/10.1145/2338965.2336765.

[38] S. Edelkamp, V. Schuppan, D. Bošnački, A. Wijs, A. Fehnker, and H. Aljazzar, "Survey on directed model checking," in *MOCHART 2008: Model Checking and Artificial Intelligence*, D. Peled and M. Wooldridge, Eds., ser. Lecture Notes in Artificial Intelligence, vol. 5348, Springer-Verlag Berlin Heidelberg, 2009, pp. 65–89. DOI: 10.1007/978-3-642-02568-6_5. [Online]. Available: https://doi.org/10.1007/978-3-642-02568-6_5.

[39] A. Wijs, "Bfs-based model checking of linear-time properties with an application on gpus," in *Computer Aided Verification (CAV 2016), Part II*, S. Chaudhuri and A. Farzan, Eds., ser. Lecture Notes in Computer Science, vol. 9780, Eindhoven, The Netherlands: Springer International Publishing, 2016, pp. 472–493. DOI: 10.1007/978-3-319-41540-6_26. [Online]. Available: https://doi.org/10.1007/978-3-319-41540-6_26.

[40] Z. Wu, Y. Liu, Y. Liang, and J. Sun, "Gpu accelerated counterexample generation in ltl model checking," in *Formal Methods and Software Engineering*, S. Merz and J. Pang, Eds., Cham: Springer International Publishing, 2014, pp. 413–429, ISBN: 978-3-319-11737-9.

[41] H. Aljazzar and S. Leue, "Generation of counterexamples for model checking of markov decision processes," in *2009 Sixth International Conference on the Quantitative Evaluation of Systems*, IEEE, 2009, pp. 197–206. DOI: 10.1109/QEST.2009.10.

[42] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem* (Lecture Notes in Computer Science), 1st ed. Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg, 1996, vol. 1032, pp. IV, 143, Springer Book Archive, Springer-Verlag Berlin Heidelberg 1996, ISBN: 978-3-540-60761-8. DOI: 10.1007/3-540-60761-7. [Online]. Available: https://doi.org/10.1007/3-540-60761-7.

[43] C. Chou and D. Peled, "Verifying a model-checking algorithm," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, vol. 1055, Passau, Germany: Springer, 1996, pp. 241–257.

[44] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '05, Long Beach, California, USA: Association for Computing Machinery, 2005, pp. 110–121, ISBN: 158113830X. DOI: 10.1145/1040305.1040315. [Online]. Available: https://doi.org/10.1145/1040305.1040315.

[45] D. Peled, "Combining partial order reductions with on-the-fly model-checking," in *Computer Aided Verification*, D. L. Dill, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 377–390, ISBN: 978-3-540-48469-1. DOI: 10.1007/3-540-58179-0_69.

[46] A. Lluch-Lafuente, S. Edelkamp, and S. Leue, "Partial order reduction in directed model checking," in *Model Checking Software*, D. Bošnački and S. Leue, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 112–127, ISBN: 978-3-540-46017-6. DOI: doi.org/10.1007/3-540-46017-9_10.

[47] D. Lavery and W.-W. Hwu, "Unrolling-based optimizations for modulo scheduling," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, 1995, pp. 327–337. DOI: 10.1109/MICRO.1995.476842.

[48] S. Y. Lee and T. W. Ling, "Unrolling cycles to decide trigger termination," in *VLDB*, Citeseer, vol. 99, 1999.

[49] S. Kim, S.-M. Moon, J. Park, and K. Ebcioglu, "Unroll-based copy elimination for enhanced pipeline scheduling," *IEEE Transactions on Computers*, vol. 51, no. 9, pp. 977–994, 2002. DOI: 10.1109/TC.2002.1032620.

[50] V. Sarkar, "Optimized unrolling of nested loops," in *Proceedings of the 14th International Conference on Supercomputing*, ser. ICS '00, Santa Fe, New Mexico, USA: Association for Computing Machinery, 2000, pp. 153–166, ISBN: 1581132700. DOI: 10.1145/335231.335246. [Online]. Available: https://doi.org/10.1145/335231.335246.

[51] G. S. Murthy, M. Ravishankar, M. M. Baskaran, and P. Sadayappan, "Optimal loop unrolling for gpgpu programs," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2010, pp. 1–11. DOI: 10.1109/IPDPS.2010.5470423.

[52] I. Singh, S. K. Singh, R. Singh, and S. Kumar, "Efficient loop unrolling factor prediction algorithm using machine learning models," in *2022 3rd International Conference for Emerging Technology (INCET)*, 2022, pp. 1–8. DOI: 10.1109/INCET54531.2022.9825092.

[53] O. S. Dragomir, T. Stefanov, and K. Bertels, "Optimal loop unrolling and shifting for reconfigurable architectures," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 2, no. 4, Sep. 2009, ISSN: 1936-7406. DOI: 10.1145/1575779.1575785. [Online]. Available: https://doi.org/10.1145/1575779.1575785.

[54] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, "Hoppity: Learning graph transformations to detect and fix bugs in programs," in *International Conference on Learning Representations (ICLR)*, 2020.

[55] E. W. Dijkstra, *Notes on Structured Programming*. Techn. Hogeschool, 1970, pp. 7–7, Cited on page iii.

Acronyms

AML Axini Modeling Language. 26, 28, 29, 54
AMP Axini Modeling Platform. 5, 26, 27, 30, 32, 34, 40, 41, 57
APSP all-pairs shortest-path. 10
ATM Automatic Teller Machine. 29, 37, 40, 41
BFS Breadth-First-Search. 12–14, 40, 43, 44, 47
DFS Depth-First-Search. 44
LTS Labelled Transition System. 6, 7, 11, 14, 16, 18, 21, 26–28, 46
MBT Model-Based Testing. 4–11, 14, 16–19, 22, 26, 39–41, 43, 47
o.o.t. out-of-time. 32
PIN Personal Identification Number. 29
PoC Proof of Concept. 5, 40
POR Partial Order Reduction. 19, 44
SPS_AllPaths Shortest Path Shrinker - All Paths. 18
SPS_POR_LabelCount Shortest Path Shrinker - Partial Order Reduction - Label Count. 19
SPS_POR_Subpaths Shortest Path Shrinker - Partial Order Reduction - Subpaths. 19
SSSP single-source shortest-path. 10
STS Symbolic Transition System. 26, 46, 47
SUT System Under Test. 4–8, 14–17, 22, 23, 26, 28, 37–39, 41, 42, 46, 47
UML Unified Modeling Language. 7

Appendix A

GitHub repository

A.1 AML models

For the AML models of the experiments, please see:
`https://github.com/seedot1234/model-centric-shrinking-in-MBT/tree/main/models`. Due to confidentiality reasons, the model for Experiment One: SmartDoor, cannot be made publicly available. This model can be requested from the author.

A.2 Raw data

For the raw shrinking data of the experiments, please see:
`https://github.com/seedot1234/model-centric-shrinking-in-MBT/tree/main/data`.

Appendix B

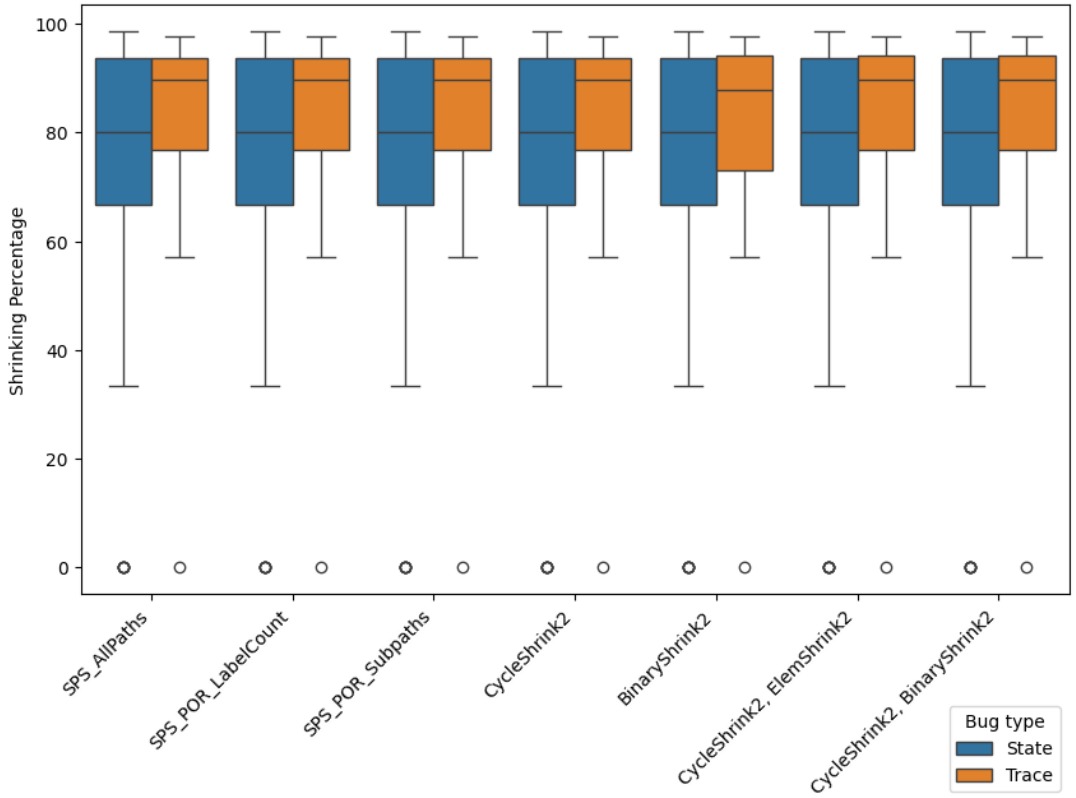
Additional Information

B.1 Data including all test cases

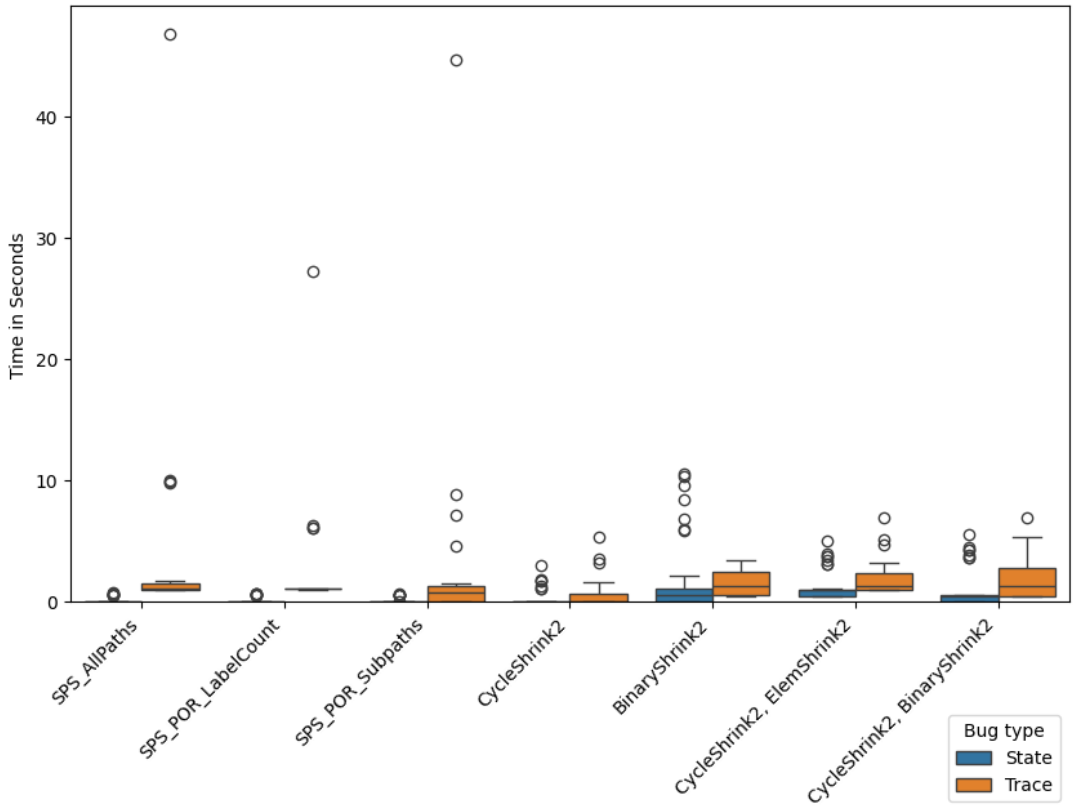
B.1.1 SmartDoor

Table B.1: SmartDoor Results - Means per Bug Type

	Test Cases per Shrink		Time per Shrink		Shrinking Percentage	
	State	Trace	State	Trace	State	Trace
SPS_AllPaths	1.00	9.75	0.09	4.74	71.75%	81.22%
SPS_POR_LabelCount	1.00	7.05	0.09	3.16	71.75%	81.22%
SPS_POR_Subpaths	1.00	8.05	0.09	3.68	71.75%	81.22%
CycleShrink2	1.42	4.50	0.23	0.84	71.75%	81.22%
BinaryShrink2	6.22	9.30	1.55	1.58	71.43%	80.14%
CycleShrink2, ElemShrink2	3.34	6.55	1.63	2.14	71.78%	81.33%
CycleShrink2, BinaryShrink2	2.98	6.60	0.79	2.04	71.78%	81.33%



(a) Shrinking Percentage per Bug Type



(b) Shrinking Time per Bug Type

Figure B.1: SmartDoor Results - Percentiles

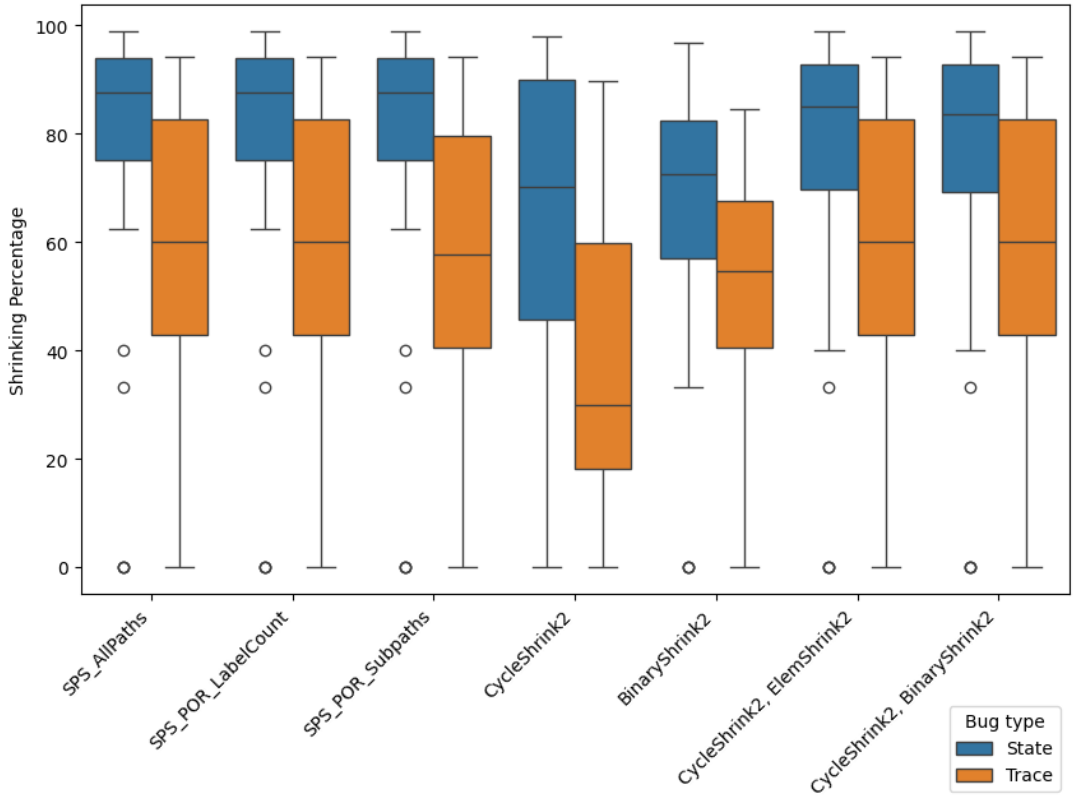
B.1.2 Koopman

None of the test cases generated by AMP during Experiment Two: Koopman's Vending Machine, resulted into a shortest path. This means that the data as reported in Chapter 6, is already the data containing all test cases. We therefore do not also report it here in the Appendix.

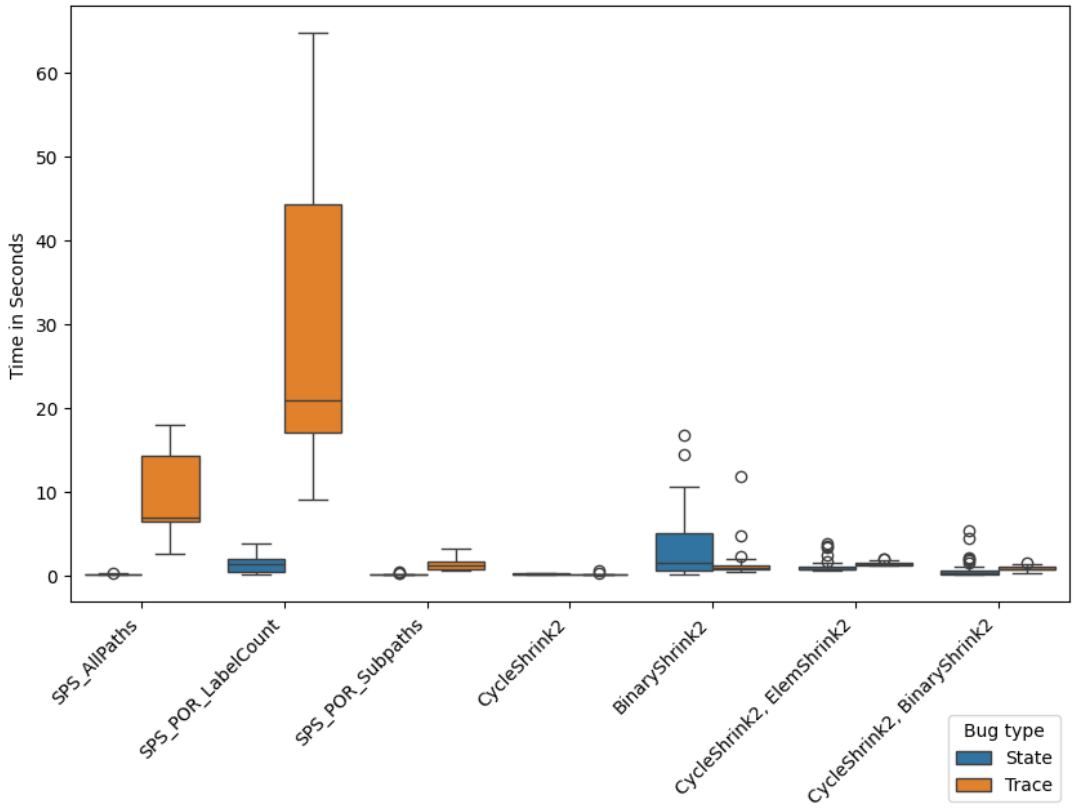
B.1.3 ATM

Table B.2: ATM Results - Means per Bug Type

	Test Cases per Shrink		Time per Shrink		Shrinking Percentage	
	State	Trace	State	Trace	State	Trace
SPS_AllPaths	1.00	17.45	0.10	9.78	75.51%	58.85%
SPS_POR_LabelCount	1.00	17.45	1.29	30.83	75.51%	58.85%
SPS_POR_Subpaths	1.00	3.20	0.09	1.33	75.51%	56.36%
CycleShrink2	1.43	1.40	0.10	0.09	62.17%	37.82%
BinaryShrink2	20.57	13.25	3.59	1.71	64.49%	52.61%
CycleShrink2, ElemShrink2	7.03	8.15	1.11	1.39	73.74%	58.85%
CycleShrink2, BinaryShrink2	7.03	8.85	0.77	0.78	73.14%	58.85%



(a) Shrinking Percentage per Bug Type



(b) Shrinking Time per Bug Type

Figure B.2: ATM Results - Percentiles

B.2 Statistical Significance

Below we report the same results as in Chapter 6, indicating for datapoints whether they are significantly different to other datapoints. Significance was determined by performing a paired sample t-test.

B.2.1 SmartDoor

Table B.3: Statistical information: SmartDoor Results - Means per Bug Type

	Test Cases per Shrink		Time per Shrink		Shrinking Percentage	
	State	Trace	State	Trace	State	Trace
SPS_AllPaths	1.00	9.75**	0.09	4.74**	79.22%	85.49%
SPS_POR_LabelCount	1.00	7.05**	0.09	3.16**	79.22%	85.49%
SPS_POR_Subpaths	1.00	8.05**	0.09	3.68**	79.22%	85.49%
CycleShrink2	1.42*	4.50*	0.23*	0.84*	79.22%	85.49%
BinaryShrink2	6.22*	9.30*	1.55*	1.58*	78.87%	84.36%
CycleShrink2, ElemShrink2	3.34*	6.55*	1.63*	2.14*	79.25%	85.61%
CycleShrink2, BinaryShrink2	2.98*	6.60*	0.79*	2.04*	79.25%	85.61%

* Statistically different from the model-centric shrinkers at $p < 0.05$.
** Statistically different from the other model-centric shrinkers at $p < 0.05$.

B.2.2 Koopman's Vending Machine

Table B.4: Statistical information: Koopman Results - Means per Bug Type

	Test Cases per Shrink		Time per Shrink		Shrinking Percentage	
	State	Trace	State	Trace	State	Trace
SPS_AllPaths	o.o.t.	o.o.t.	o.o.t.	o.o.t.	o.o.t.	o.o.t.
SPS_POR_LabelCount	1.00	41.46**	0.30**	30.86**	77.40%	84.27%
SPS_POR_Subpaths	1.00	11.17**	0.20**	5.92**	77.40%	83.99%
CycleShrink2	2.33*	3.17*	0.42	0.67*	68.71%	77.77%
BinaryShrink2	12.50*	23.83*	3.44*	3.55*	78.30%	80.75%
CycleShrink2, ElemShrink2	7.33*	11.12*	2.12*	2.32*	80.17%	85.59%
CycleShrink2, BinaryShrink2	7.50*	12.71*	2.35*	2.04*	80.17%	85.59%

* Statistically different from the model-centric shrinkers at $p < 0.05$.
** Statistically different from the other model-centric shrinkers at $p < 0.05$.

B.2.3 ATM

Table B.5: Statistical information: ATM Results - Means per Bug Type

	Test Cases per Shrink		Time per Shrink		Shrinking Percentage	
	State	Trace	State	Trace	State	Trace
SPS_AllPaths	1.00	17.45 ‡	0.10 †	9.78†‡	83.90%	61.94%
SPS_POR_LabelCount	1.00	17.45 ‡	1.29∗‡	30.83∗‡	83.90%	61.94%
SPS_POR_Subpaths	1.00	3.20 ∗†	0.09 †	1.33∗†	83.90%	59.33%
CycleShrink2	1.43∗ † ‡	1.40∗ † ‡	0.10 †	0.09 ∗ † ‡	69.07%∗ † ‡	39.81%∗ † ‡
BinaryShrink2	20.57%∗ † ‡	13.25∗ † ‡	3.59 ∗ † ‡	1.71∗†	71.65%∗ † ‡	55.38%∗ † ‡
CycleShrink2, ElemShrink2	7.03∗ † ‡	8.15∗ † ‡	1.11 ‡	1.39∗†	81.93%	61.94%
CycleShrink2, BinaryShrink2	7.03∗ † ‡	8.85∗ † ‡	0.77 ∗ † ‡	0.78∗ † ‡	81.27%∗ † ‡	61.94%

* Statistically different from SPS_AllPaths at $p < 0.05$.
† Statistically different from SPS_POR_LabelCount at $p < 0.05$.
‡ Statistically different from SPS_POR_Subpaths at $p < 0.05$.