February 2018 - July 2018

MASTER THESIS

IMPROVING DIAGNOSIS BY GROUPING TEST CASES TO REDUCE COMPLEXITY

Martijn Willemsen
m.j.willemsen@alumnus.utwente.nl

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

Exam committee:
prof. dr. M.I.A. Stoelinga
dr. S. Schivo
dr. ir. H.M. van der Bijl (Axini B.V.)

UNIVERSITY OF TWENTE.

Abstract

Software quality is an important aspect of software design and software development. To ensure software quality, different types of tests are used. But tests have an inherent problem. Dijkstra wrote *Program testing can be used to show the presence of bugs, but never to show their absence!*[11]. Extensive testing should be done to come close to finding all bugs. Computers can help in this extensive testing. One way computers can help is through model based testing. Model based testing has shown a lot of promise in improving product quality and developer productivity[40, 21, 34]. Model based testing, or MBT, uses a model of the system to generate many test cases.

The problem with model based testing is that diagnosing the results is hard. The diagnosis is a reoccurring step in the testing process. This is why the diagnosis should be simplified. If many test cases fail, diagnosing these failures becomes a challenge. Due to the number of the test cases and the number of steps in each test case, the failures become hard to comprehend. It can be unclear why tests fail, and which tests fail because of the same fault in the code. To solve the problem of size and incomprehensiveness two major solutions have been identified. The two solutions are complementary. The first solution is grouping test cases with a similar fault. The second solution is to use root cause analysis or fault localisation to identify the step in the test case that produces the fault.

By grouping test cases, all test cases that have the same underlying problem, the same faulty line of code in the system under test, can be merged into a single group. This reduces the number of test cases to be analysed by hand. Root cause analysis helps in the analysis of a single test case. Root cause analysis tries to identify the step in a test case that causes the failure. Without this fault causing step, the test case will pass. The diagnosis is simplified since finding the faulty step helps in translating the failure into code and helps in reproducing the failure.

The result of this research is an implementation of different methods that can execute root cause analysis and can group test cases. The different implementations, SFL and data mining, are validated. The validation is done using a real world system with introduced faults, and scored on different metrics including analysis duration, F-measure and accuracy. This results in the best approach where SFL is used with steps as components including data.

Acknowledgement

This thesis contains a lot of effort from different people. First of all by Machiel van der Bijl. He has helped in guiding the research, writing this thesis and as a general sparring partner to discuss the approach.

Besides Machiel, at Axini everyone has helped tremendously with any questions that have arised and with inspiration on several aspects of the research.

Finally I would like to thank Marielle Stoelinga and Stefano Schivo in their academic guidance during the research and suggestions on how to improve the work submitted. They also helped forming this research.

Contents

1 Introduction

> *Program testing can be used to show the presence of bugs, but never to show their absence!*
>
> **— Edsger W. Dijkstra**
> 1969

As Edsger W. Dijkstra has already mentioned in 1969: Tests can never show that faults do not exist. This is a problem for designing and building safety-critical software. Some software needs to be *fault-free*. Software for aeroplanes for example. The reason that tests can only show the presence of faults is due to the fact that testing is never complete[1]. A marginally complex system already requires many test cases to cover the full system and to cover the full specification. To be able to come near to completeness, the number of tests that are required is immense. Building this amount of tests by hand is nearly impossible. Computers can help in testing extensively. One way computers can help is through model based testing. Model based testing has shown a lot of promise in improving product quality and developer productivity[40, 21, 34]. Nevertheless it is still not complete. Model based testing, or MBT, is a formal approach to software testing. Model-based testing tools automatically generate high quality test suites based on formal specifications of the tested system. See section 1.2 and 3.1 for a detailed explanation of model based testing.

1.1 Problem statement

The advantage of MBT is also a disadvantage. Due to the large number of test cases, there are also many test cases that are likely to fail. This makes diagnosing the results difficult. Diagnosing the traces is not trivial, but is required every test run that contains failures. The diagnosis, or analysis, is also time consuming and mind numbing, since every step looks similar and the order of the steps matters. This recurrence of the diagnosing stage shows the importance of simplifying this part of the MBT process. There are two major causes that make diagnosis difficult: there are many test cases and these test cases can contain many steps. If there are many test cases that fail, it is a lot of work to analyse. These two problems can be solved using test case grouping and Root Cause Analysis.

[1] Completeness: If the implementation is not correctly implementing the specification, then at least one test fails.

With test case grouping all test cases that have the same underlying problem can be grouped. The underlying problem is determined in different ways, and this is how the different implementations validated in this research distinguish themselves. All tests in a specific group will then show the same problem in a different way, so only one test case has to be analysed by hand instead of all test cases in that group. This simplifies the work required by the user since the user truly has to do less work. Suppose that three out of the ten failed test cases fail on the same step, this step is probably the same line of code for all three test cases. If this step is fixed, all three tests will not fail anymore. This shows that only one third of the work is left to do, both in analysing the faulty step, and in fixing the problematic line of code.

With Root Cause Analysis, the root cause of a problem can be found. A root cause is the step that produces the symptoms of a failure. If this problem is solved, all symptoms of the problem disappear. With the root cause available, the analysis of a single test case is simplified as the probable faulty step is identified. This step can be presented with the inputs to reproduce the failure. Since the identification of the root cause will narrow the focus of debugging the problem, RCA will result in an easier evaluation of the problem and therefore a quicker fix of the fault in the code or model.

Both methods help in simplifying the analysis of the results of a test run with model generated test cases. A simplified analysis can improve the adoption of MBT since it is less expensive and has a smaller learning curve. A higher adoption of model based testing will result in less bugs in production code[34] and higher quality software.

1.2 Model based testing

MBT allows for testing a system under test with respect to a formal specification. The models of the system are used for generating tests to validate the correctness of the system under test.

Labelled transition systems are one of the ways that the models can be described. A labelled transition system contains states and transitions. The states represent system states and the transitions represent inputs and outputs that are possible from that particular state. An LTS also contains an initial state, which is the state in which the execution of the model starts.

MBT has some concepts that are used in this thesis. A test run is a set of test cases with their verdicts. Each test case consists of multiple steps. The ordered set of steps is the test trace.

Model based testing is described in more detail in section 3.1.

1.3 Atana design and implementation

To actually make the proposed improvements to diagnosis a reality, a tool called Atana is created. Atana uses the traces of the passing and failing test cases to group test cases and identify the root cause of a failure for each test. The grouping and analysis process is implemented in different ways. Each of these implementations is validated.

Atana must contain the following components (see also figure 1.1):

- A trace and model parser
- An API for communicating with the grouping and root cause algorithms
- Multiple separate micro services that implement the grouping and root cause algorithms
- A view to return the results to the user

To parse the traces and models, the FasterXML Jackson[2] library is used. This library allows for parsing JSON documents (and more data formats) into JVM based objects. Jackson is used in many high profile projects for this purpose. It is also the easiest way to parse the data. Finally it has great integration with the other libraries and programming languages used in this project.

For the REST API the Spring Boot library is chosen. This library allows for simply annotating methods to create REST endpoints. It also allows for calling other REST endpoints with a single method call. Besides easy REST communication, Spring Boot has been chosen for another reason. Spring projects have easy Java bean injection. This allows for a lot of simplicity during the creation of objects like controllers, services and more. Finally Spring Boot already contains a database management library called Hibernate. This allows for storing the parsed data into a MySQL database without writing any SQL queries. Because of these many conveniences Spring Boot is used as the basic framework that is the core of Atana and binds everything together.

[2] `https://github.com/FasterXML/Jackson`

The first analysis service implementation to check is an algorithm based on Delta Debugging(DD)[45, 43, 44] and Spectrum-based Fault Localisation(SFL)[41]. Both Delta Debugging and Spectrum-based Fault Localisation are techniques that compare the traces of different test cases and their verdicts to find differences between a passing and a failing test cases. The difference that is found, is then assumed to be the faulty step. Delta Debugging does this comparison in a run-time fashion where it makes a small change, reruns the test case, and sees if the failure still happens. Spectrum-based Fault Localisation does this comparison by only comparing runs that are already available. Suppose that all failing test cases use a certain step, that is never used in a passing test case. This step, that only occurs in failing test cases, is most likely the cause of the failures. Delta Debugging and Spectrum-based Fault Localisation were chosen because they have been used in similar problems and have had good results.

Since this research focuses on analysis after running the test, the first analysis service implementation mainly focuses on Spectrum-based Fault Localisation. The Delta Debugging idea could be implemented in future work to validate the results that SFL provided. This validation could be done by removing the faulty step that was identified by SFL and re-run the test to check if this time the test passes.

The second analysis service implementation applies a data mining algorithm to solve the same problem. Weka[18] is used as a framework for the data mining algorithms since it allows to run several algorithms on the same data set. To see what kind of data mining approach works best, different algorithms are used. Data mining was chosen because it seems to be the silver bullet in any problem. The only way to validate this, is to test it on a real system and compare it to other approaches. Another reason is the fact that looking for patterns is how data mining works on a very basic level. Failing test cases also have patterns that cause the failure. Finding these patterns will help in identifying the root cause and will help in grouping test cases with a similar failure. Data mining is especially well suited for this task, because it allows for some uncertainty in which patterns are the same. Suppose that a parameter of one of the test steps is different in two similar test cases, data mining can still allow them to be the same since the parameter falls in a certain interval.

Returning the results to the user is done through Swagger UI. This tool is created for documenting and testing REST endpoints. This does mean that the results are not properly styled, but the information is available. This also means that there is proper documentation for all endpoints, which allows for easy extension of the project.

The described components are all shown in the architecture in figure 1.1.

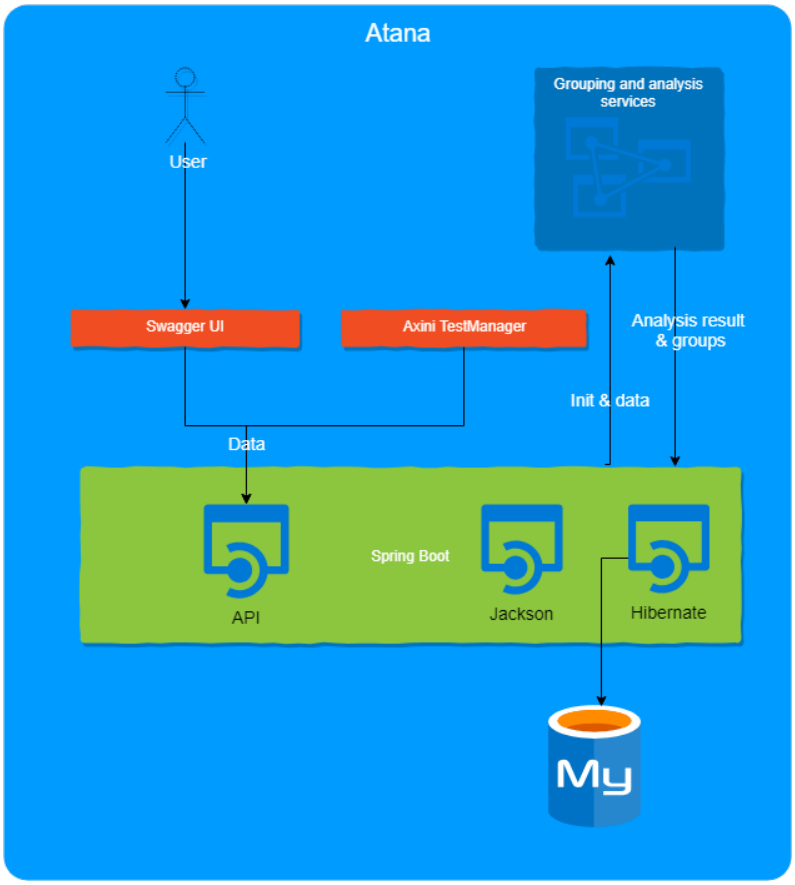

Fig. 1.1.: Architecture describing Atana. Swagger, Spring Boot, Jackson, Hibernate and Axini TestManager are existing tools and libraries used to build Atana. The grouping and analysis service and the API in the Spring Boot component are newly created.

The two different analysis service implementations are validated using a mutation testing approach, where several mutants are created and the test cases are applied. Mutants are versions of the original system under test in which a fault is introduced. After running all the tests on the different mutants, the data is stored in Atana and the different analysis services are used to find the groups and root causes. With the known mutations in the current mutant it can be validated that Atana gives the correct test grouping and root causes. The implementations is evaluated based on accuracy, F-measure, recall and resource usage.

Besides the tool, documentation of the tool is also important to be able to reproduce these results. The description of the design and implementation tool contains information about how the tool is developed, how it is tested and what important considerations are taken into account during the implementation process.

1.4 Results

The results of this research is generated by running multiple experiments with Axini TestManager, Atana, and the two analysis services. The experiments are run on four data sets that use different models of the same SUT and different test case generation strategies. For each of the data sets several analysis methods were used.

In the end for the SFL analysis service, the best performing experiment used steps as the components in SFL and the labels of the steps contain the data stored in the model. The best performing experiment for the data mining analysis service used the EM technique.

The comparison between the two best performing results is not possible on most aspects due to a different approach to collecting the results. On only a one single metric, the two are comparable which allows to conclude that SFL using steps with data is the best solution to the research question.

1.5 Contributions

This thesis and related results delivers the following contributions.

- Application of formal fault localisation methods from white box testing to black box testing using MBT.

- A validated new approach to analysing the results of a MBT test run.

- Improve the ease of use of MBT when using large test sets with many failures.

1.6 Assumptions

In this thesis some assumptions are made. The assumptions were not researched explicitly, but are assumed to be reasonable. The following assumptions are made:

1. Model based testing creates many test cases. Since it is an automated technique, it can create many paths through the model that have little differences. All of these paths can be transformed into a test case. This results in many test cases.

2. There are many test cases that have a partial overlap. Since the test cases are created based on paths (sequences of states and transitions) through the model, two traces can have a path that occurs in both traces. Suppose the system under test (SUT) is a web-app with multiple modules. To get to a module, the user must click a menu bar. This is where the overlap can come from. All test cases that will test the same module will have to click the menu bar. If the menu bar is not properly implemented, all test cases that use this menu bar will fail. It is also possible that there is a failure in preparing data for a test case. In the model this overlap would look like a sequence of states and transitions (a path) that are in both test cases.

1.7 Motivating example

Finding the root cause of a symptom is not always easy since the root cause is not always the last step in a test case. The root cause of a failure is the step that causes the failure. If this step was not in the test case trace, the symptoms would disappear. An example of the fact that the root cause is not always the last step in a test case is shown in figure 1.2. This example shows that a simple error, such as switching a greater-than with a less-than, can be hard to find and only shows up after some lengthy test cases (even though this contrived example is still relatively small). There is only a slight difference between the model and the implementation: gt! and lt! are reversed. This could be due to a line of code that compares an inequality. If this inequality check is done the other way around, the output is reversed (as in this example where the transition from state 2 to state 3 is changed into a transition from state 2 to state 4). The other transitions are still available in subsequent states. So if the following test trace was run, it would pass the test: $input?->gt!->c?->d!->f!->reset?->reset!$, but this test trace would fail: $input?->gt!->c?->e!->f!$. The reason that this test trace would fail is

that the test will end-up in state 9. In this state, the output f! could not be observed because the tester expected to be in state 7. This is also where the test will fail. The bug is not in the last state, where the test case failed, but in the transition between state 2 and state 4. Finding this kind of problem in the implementation (and report it in the model) is the aim of this research.

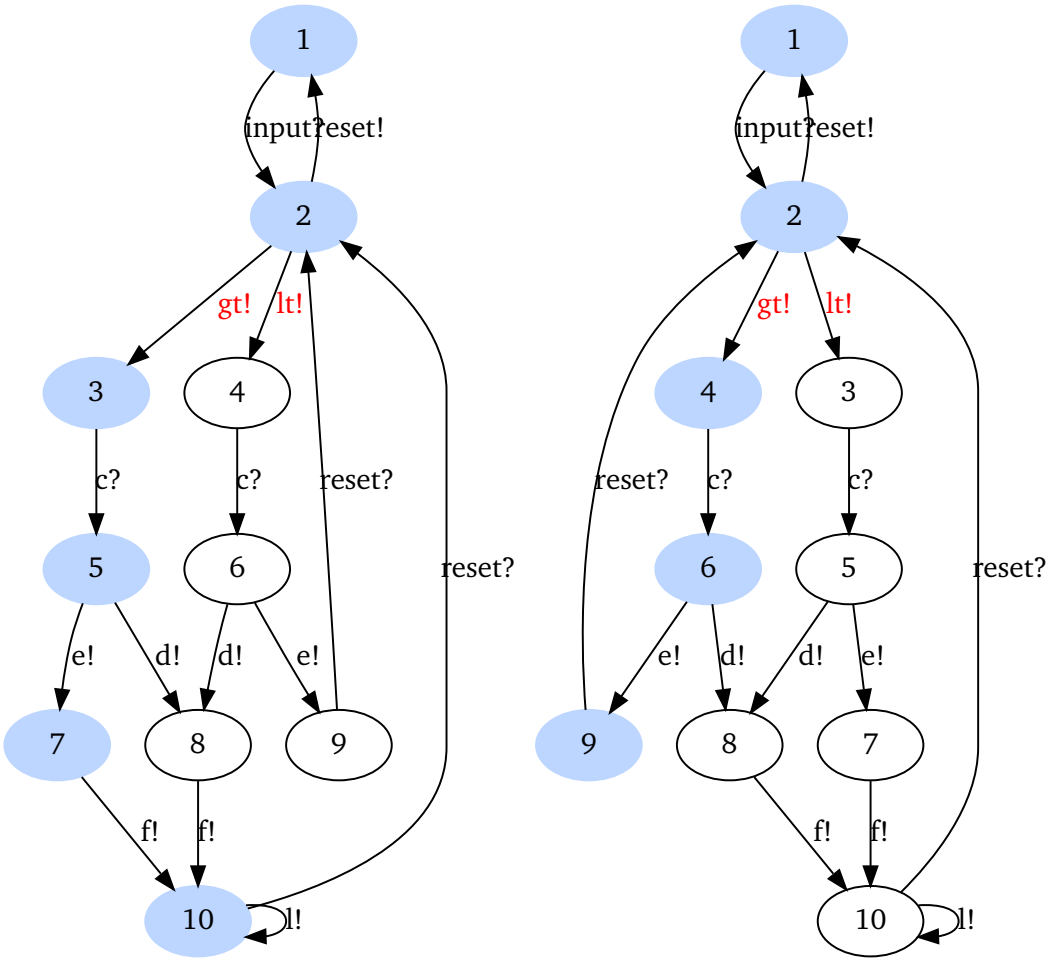

Fig. 1.2.: Example of a model (left) and implementation (right) with a greater than or less than fault which is highlighted in red (note: quiescence is omitted)

1.8 Project context

This graduation project is performed in cooperation with Axini. Axini is a software company specialised in model-based testing. Their main product is called TestManager. It is a model-based testing tool which uses symbolic transitions systems to model systems under test. Axini also has the problem that diagnosing the results of the tests can be hard and time consuming. Some systems only start showing symptoms after running a test case with 2000 steps. For now they avoid these problems by running a test set with few test cases and each test case does not go 2000 steps

deep. Axini TestManager (Axini TM) is used to generate test cases and collect data for analysis in this thesis.

1.9 Thesis outline

The rest of this thesis is structured as follows. In chapter 2 the research goal and the research questions are presented. The background information required for this thesis is explained in chapter 3. The related work and the literature review/summary of the related work are presented in chapters 4 and 5. The research method is shown in chapter 6. The design and implementation of Atana is described in chapters 7 and 8. And the results are presented and discussed in chapter 9. Chapter 10 contains the conclusions, and suggestions for future work are presented in chapter 11

2 Research goal and research questions

2.1 Research Goal

The goal of this research is to find a way to group test cases that fail in a very similar way. Grouping tests helps in diagnosing problems in the SUT because they allow developers and testers to focus on fewer failed test cases. Since testing is expensive and time consuming, better and easier diagnostics help in lowering costs and spend more time running tests rather than staring at a set of failed test cases.

Similarity can be defined by the root cause of the failure. Using the root cause is important since in model generated test cases there are often overlapping pieces of the SUT tested by different tests. If there is a single fault in the SUT, every test that uses that code will fail. This creates a set with a lot of tests that all fail with the same root cause. If a tester or developer has to go over them all, this will be a very exhausting tasks, both mentally and time-wise.

A way to group test cases is to create a program that can identify the step in the test trace, the transition in the model, in which a fault occurred (the root cause). The basis of this program could be a machine learning technique that consumes the traces of failed tests. The goal of this program is to make the output of generated test cases insightful for users. Data mining is all about patterns. Parts of test traces can reoccur in different tests. If such a partial trace is reoccurring in many failing tests, this partial trace is likely the root cause of the failure for the tests that use this partial trace.

Different techniques, besides data mining, will also be considered to group test cases. Techniques like delta debugging(DD) [44], Spectrum-based Fault diagnosis(SFL) [41], and trace minimisation also seem promising to group test cases. These are all techniques that work by modifying traces to gain information. Trace minimisation, or trace reduction, tries to shorten traces by removing loops and other steps that are not required to reproduce the failure. If this process is performed on all traces, there is a set of shortest possible traces left, that all reproduce the same failure. All traces that are the same now, should be in the same group. DD and SFL are directed more towards root cause analysis. These techniques can find a root cause by comparing traces for passing and failing tests. In SFL this is done by identifying all steps or transitions that occur in failing tests but do not occur in passing test cases. The

occurrence in failing tests makes them more likely to cause a failure. It is usually not the case that a single step or transition only occurs in passing or failing tests, this is why a similarity metric is used to determine if a step or transition is faulty. For DD the comparison between failing and passing test is done by changing a single step or transitions to see if this still causes a failed test case. If the test case does not fail anymore, the modified step is probably the root cause of the failure. If DD or SFL can find a root cause, they can group the tests based on that root cause.

A big advantage of this approach to diagnosis, where the diagnosis is performed mostly automatically, is that it makes the use of automated testing using models more accessible and quicker. Many programs could have fewer faults if testing is was outsourced to computers and the results are concisely communicated to the testers and developers. This accessibility, or ease of use after running the tests, is also a major addition to the state of the art.

The program that results from this thesis, Atana, could be extended in the future to help avoid a known bugs and faults during testing, or focus on a set of often failing tests to make testing more efficient. There is a lot that could be improved on this area of model based testing. The scope of this thesis is initially limited to grouping failing test cases after they have been run to improve diagnostics.

2.2 Research Questions

From this goal, the following main question is derived: **What method can effectively group similar failing test cases to make diagnostics easier?** This question consists of two major parts: grouping similar cases and diagnostics. These two parts work together where the first part is helping to achieve the second part. Diagnostics means the analysis of the results after running the tests. Often this analysis will consist of finding the location where the test case has failed. With Root Cause Analysis the diagnostics can be simplified. Adding the grouping would allow a developer to avoid the analysis of all failed test cases in a group.

To answer this question the some sub-questions are considered.

1. How can algorithms like Delta Debugging and Spectrum-based Fault Localisation be used to find the root cause of a failed test case?

2. Does a data mining clustering algorithm work in grouping (failed) test cases?

3. Which data mining techniques can be used to find the root cause of a failed test case?

4. How can the information incorporated in the model be used to improve Spectrum-based Fault Localisation in Model Based Testing?

3 Background

In this thesis, there is some terminology used that will be explained in this chapter. There are also some common abbreviations used, which are also explained.

3.1 Model based testing

Model based testing is the process of checking if a system under test is in compliance with a model[39]. The model describes the required behaviour of an implementation. If a SUT is in compliance with the model, the SUT implements the required behaviour. In this research, labelled transition systems (LTSs) and symbolic transition systems (STSs) are used to describe the model of the SUT. The LTSs are used in the examples and unit tests in Atana, while the STSs are used by the Axini Test Manager. A set of generated test cases can be used to validate the SUT. The biggest advantage of MBT is that a valid model will create valid test cases which need no manual intervention and can test the complete system. This removes all manual labour and therefore human mistakes.

Testing can be done using three strategies: white box, black box or gray box. Black box means that the SUT is interpreted as a black box which means that there is input and output that can be observed, but nothing about the internals of the system is known. During gray box testing some internals are known. And finally during white box testing all internals are known and can be used during testing (such as source code and program flow). In this research black box testing is used. Which means that the internals of the SUT are unknown to the test executor.

Labelled Transition Systems are a way of describing a model of a system. A Labelled Transition System (LTS) consists of states and transitions with labels on the transitions which describe the actions ([39] section 3.1). The actions can be inputs, outputs and quiescence. Inputs end with or start with a question mark. Outputs end with or start with an exclamation mark. Quiescence means there is no output. Formally a LTS is defined as a 4-tuple $< Q, L, T, q_0 >, where$

- 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

An example of a Labelled Transition Systems is shown in figure 1.2.

During the execution of the tests with a running SUT, a reset sequence is required. This reset sequence is used to reset the SUT to its initial state. A reset sequence is a sequence of actions that can always, from any state, return the SUT to the initial state from which the tests are started. This also means that a fault in this reset sequence can make tests fail that do not seem to use faulty steps.

Axini uses an extended form of an LTS called an STS: Symbolic Transition System[13]. STSs extend on LTSs by incorporating an explicit notion of data and data-dependent control flow. There are two major differences between an LTS and an STS. The first difference is that an STS contains internal variables. These are added to an LTS in a separate set V, which makes the 4-tuple a 5-tuple: $< Q, L, T, V, q_0 >$. The other difference is that in an STS variables can be updated and checked during transitions. These two differences allow STS models to be better compatible with the extensiveness of complex systems. STSs therefore also allow for constraints on transitions that can be taken or not and remove states that would otherwise have been required to add constraints.

3.2 Abbreviations

The important abbreviations used in this thesis are given in the table below.

SUT	System under test	The SUT is the system that will be tested by the generated test cases. Besides SUT there are also some other abbreviations that mean about the same, like SUC and PUC. SUC is a system under concern and the same holds for PUC which is a program under concern.
MBT	Model Based Testing	MBT is the process of using models of the SUT to test a SUT.
LTS	Labelled Transition System	The LTS is a model used in MBT. See also section 3.1.
STS	Symbolic Transition System	The STS is a model used in MBT. See also section 3.1.
RCA	Root Cause Analysis	RCA is the process of finding the cause that creates the symptom.

3.3 Definitions

The distinction between fault, error, and bug can be vague. These terms have been defined in the IEEE Standard Terminology[22]. A fault is an incorrect step, process, or data definition. An error or bug is the difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. In this paper the term bug denotes a problem with the code. The term fault is used to denote any problem. And the term error is used to denote any problem caused by a human.

A fault and a symptom also need distinction. Many faults may not be directly observable, but a seemingly invisible fault may manifest itself through failures at other observable locations. While faults are the root cause of failures, symptoms are external manifestation of failures. [7] In this context it is useful to know if we are dealing with an actual fault or a symptom of a fault.

Since the thesis focuses on root causes, that is defined as well. According to Lamrauoi et al, a root cause is the fundamental reason for the occurrence of a failing program execution [26]. So in the context of this thesis, that is the step or transition that makes the test case fail.

4 Related Work

Quite some research has been done on model based testing, and in many related fields. An example is automatic test case generation. The generation of test cases can be done in different ways. Generation strategies range from looking at memory usage and trying to attack a system, to generating state walks in a finite state machine, to code coverage based test generation. With these strategies the test cases can be created by listing inputs (actions) and expected outputs (results). These traces of inputs and outputs form the test traces. Test case generation strategies are all very different, but one thing that they all have in common is that many many test cases can be created with possibly very long traces and reproduction paths. To reduce the load on the tester or developer that uses MBT tools, Root Cause Analysis has been used [31].

Several search queries were used for finding related work. The main keywords used are: data mining, model based testing, spectrum-based fault localisation and root cause analysis. Besides these keywords that were used in searches for papers, citations of relevant papers are also used to further improve the related work. Especially since the related work of others could be useful in this section. These search queries and related work resulted in several themes and approaches that reappeared in the majority of the papers and in which they can be grouped.

4.1 Root Cause Analysis techniques

Researching the topic of RCA introduced many different applications of RCA. Often applications are in hardware. For example a chair that is made, but has failing joints. The root cause of these failing joints is somewhere during the production process of the joints, so maybe the tenon machine creates to large pockets for the tenons to fit. The configuration of the tenon machine is the root cause of all chairs with failing joints. The approaches of papers that use RCA in hardware is not really transferable to software programs and those are therefore not discussed. All related work provided different insights in the whole concept of root cause analysis and what is important in this topic.

One insight is the localisation of faults in the whole framework. Where does the fault occur? In which part of the development and testing tool-chain? This insight shows that it is possible to find if test case generation produces faults, or if some other step in the testing framework has produced a fault [30, 31]. Finding faults

in the whole framework could be very useful when a SUT returns something that the test tool does not understand. Identifying these faults can be very hard to do by hand, because of the many permutations that are possible and is hard to keep track of. The topic of RCA in tool-chains is related to this thesis, yet still different in the sense that the goal of this research is to really interpret the results of the tests, not the whole tool-chain. This thesis and the related research could be combined to improve the usability of the whole tool-chain. Furthermore the tool that is the created in [30] is based on predefined rules, which require effort of the user. This research hopes to remove this effort and make it a seamless integration with the rest of the tool-chain and have a small learning curve.

A different way of approaching the problem of RCA is from a more mathematical point of view. Lamraoui et al [26] propose to use a new encoding for the full flow-sensitive Trace Formula. The full flow-sensitive Trace Formula is equivalent to the control flow graph of the program over a certain path through the program. With this graph, a root cause can be found by going over the graph and check assertions on the different calls to other methods. These checks keep the parameter values in mind to give more details on when a certain method call fails, and where this call originates from. A more detailed explanation can be found in [27]. The full flow-sensitive trace formula can find all root causes of faults but is usually large and not very scalable. To create a new encoding, and there for make the full flow-sensitive Trace Formula more scalable, they make, among others, use of coverage information. This could prove important in proving where a developer could map a state into the code. So if the resulting program of this research would determine the state in which the root cause occurs, this still has to be translated to the location in the code. The feasibility of this approach is still left to determine but this idea can prove valuable in the path to success. The new encoding does not seem as useful since it creates traces itself where this research assumes that the traces are already created. The post-processing step where different traces are being compared does seem useful in the trace reduction (see also section 4.2). In this step the failing traces and the succeeding traces are being compared to exclude passing sub-traces from a failing one.

Zeller [45, 43] has a different method to reach the root cause of a problem. He uses Delta Debugging, which tries to isolate the difference between a passing and a failing test and with this information he tries to find the full cause effect chain. The chain contains a reason for each effect that is observed. An example of such a cause-effect chain could be that the user inputs a negative value. This input is not validated, and is trusted. This input is later used as a lookup in an array. There is no value found in the array at that specific index. Which causes the program to use an exception handler. The exception handler tries to save the problem by extracting a default value, which was not yet set. So the program crashes and does not return anything

to the user. With this chain one can find the root cause of the failure. Which is, in the example, the invalid input to be allowed. The biggest disadvantage of this approach is that it is required to re-run tests to check if a specific change in the test trace still causes a failure. Zeller has also applied this technique to real programs and uses a debugger to find the root cause within the program. Finding the root cause within the code of a program is something that cannot be done in a black box MBT project since it does not have the source code available. The idea of finding differences between the steps in passing and in failing test cases could be useful in this research. What is very useful is that Zeller has also created and provided a prototype called AskIgor.org (described in more detail in [8]). The prototype could help in implementing this technique. Artho [5] has extended the technique described by Zeller with an iterative version of delta debugging which uses previous versions of a program to find the bug. This is especially useful when there is no input available that passes, since a passing program is a requirement to delta debugging by Zeller. This is not as useful as regular delta debugging since this research aims at finding the root cause of a failure from a test case based on a single program version, not multiple. Nor does it have access to the raw program to compare the differences. A big advantage of delta debugging is that it also helps in readability as it reduces the size of the reproduction path as well. The reproduction path is the set of steps that have to be performed to reproduce a fault. The importance of being able to find a short reproduction path shows especially in [43] where they reduced the GNU Debugger to a single faulty line. In essence, Delta Debugging is a step-by-step change of the test to identify at which step the test starts failing or passing. The smallest change to go from passing to failing or the other way around is the root cause of the failure. Note that this requires at least two tests, but explodes exponentially when more tests are compared. This essence could be useful and applied to model based testing.

Some other techniques have also been proposed, like comparing coverage of a passing and a failing program with the difference being the probable root cause (Harrold et al [19, 23]). Another technique is explicit specification, which is based on a model checking. It combines multiple methods to extract a useful debugging trace from the counter example that is provided by the model checker (Groce [16]). And finally, Nearest neighbor (Renieris et al [33]) is proposed, which tries to find the most similar test that passes to a failing test and does a comparison between these two tests. This is very similar to Delta Debugging and SFL and could even be considered as an addition to simplify both these techniques.

4.2 Increase readability and diagnostics

Trace minimisation, or trace reduction, is a technique to shorten the traces of a failing test. This helps with the readability of what actually went wrong since there is less to read and therefore less to remember. An example of such a trace minimisation procedure is presented in figure 4.1. The model on the left contains one faulty transition. The first test trace contains a loop, which is removed from the second test trace. The actual faulty step remains in the test trace, but the trace is a lot shorter. This makes the failure reproducible for humans without having to perform too many steps. Kanstrén et al [24] are trying to make traces shorter using data mining. They use a test case generator and keep constraining the generation of test cases. Each test is then run again and shorter cases are looked for that reproduce the same fault. Note that this will result in a single shortest test case that reproduces the fault.

Kanstrén et al also used the reduced set of traces for pattern mining. First they summarised the number of different steps required to reach the failure. Secondly they ordered the trace sequences.

The work of Kanstrén et al provided a lot of inspiration. One idea that came up was to use data mining to generate traces and the utility function of the traces is the time it takes to reproduce the same fault. If the trace never reproduces the fault, it has an infinitely large utility value, while the shortest trace has the lowest utility. This could be used in a generative algorithm. Another idea is to make a tree structure to show the equivalent test sets / traces. The root of the tree is the last state in the set of states and transitions that occur in all test cases. Then it branches into the last performed steps in all traces before this state, and so on (see figure 4.2 as a simple example where 1 and 2 are the two test cases and 3 is the combined tree). The broader the tree, the more problems are there in that specific piece of code, since one can arrive at the same fault in many different ways with many different test cases. This a visual interpretation of the traces and helps with root cause analysis as well.

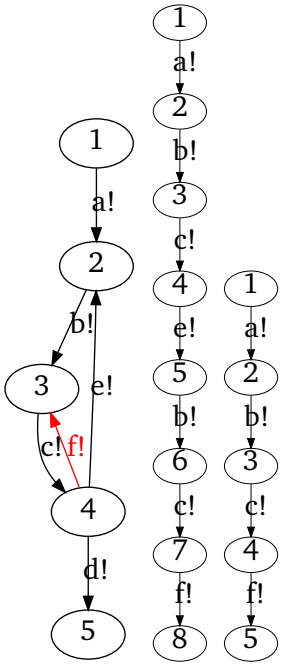

Fig. 4.1.: An example of trace reduction. The left figure is the model. The middle one is the long trace. The most right figure is the shortened trace

A different way to increase readability is to combine different traces in a single representation. Adam et al [4]

suggest again a graph based representation (see again figure 4.2). This representation makes use of the human's recognition of images. In a more measurable way: a state that has the most incoming and outgoing transitions is probably an important state in determining the root cause. The rest of the research by Adams et al is less relevant to this work since it focuses a lot on generating the traces from user sessions. This trace generation is irrelevant since the traces are already known. One thing that is very interesting is the fact that they consider that some steps in a trace do not really matter for the result. Simply removing steps in a trace is a way to reduce the traces as well. After removing a step the whole traces have to be re-run, which could take some time.

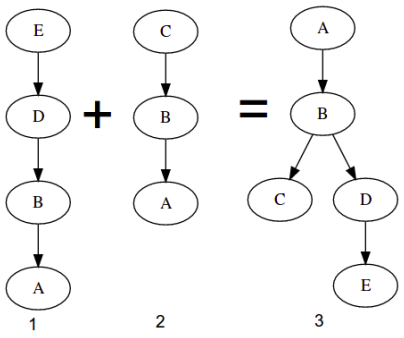

Fig. 4.2.: An example of combining two test cases (1 & 2) into a tree (3) where the similar steps are combined.

Nieminen et al [31] have done research into Root Cause Analysis as well and the result of their analysis process is an XML file. This is a very useful result as it is semi-structured data which can easily be read by a machine. Using XSL and a viewer this can also be read by humans. This allows the data to be further processed in the tool chain. Whether the usage of XML is really relevant for readability in the current research is still to be determined, but a useful thing to consider. Nieminen et al have used algorithms for extracting the root cause of failure for a mobile switching server. Regrettably the paper does not report what algorithms were used.

Finally Weiglhofer et al[41] have done research in grouping failing test cases to improve the readability of the result. They used Spectrum-Based Fault Localisation (SFL) as a grouping method. Chen et al [6] also used SFL for Root Cause Analysis. The research that Weiglhofer et al and Chen et al have done has a similar goal to the research done in this thesis, and is therefore very useful as a comparison in how well both methods work. There is a difference between their research and ours in the sense that they used LOTUS as a specification language and tried to create groups based on branch coverage in LOTUS. In this research states are used as classification entities; each test case can fail due to a fault in that state. This is a new approach to using SFL, but since it has been applied to many different situations, it is promising. It has also been validated by Abreu et al [1]. How the states translate to code is not in the scope of this research. The pseudo code provided by Weiglhofer et al can help in implementing test case grouping service with Spectrum-Based Fault Localisation.

4.3 Data mining techniques

Data mining is a very broad field; there are many different techniques. This section of related research focuses on what other researchers have used in similar problems and what is important to consider when data mining the traces of a test case.

The first step for any data mining problem is to prepare the data. Without data preparation the data mining process will not perform as well as with clean data. The build model can incorporate coincidences that are irrelevant to the real problem. Suriadi et al [36] really focus on that aspect of the whole process. They tried to enrich the data by combining the traces with external data and with the data itself. An example of this could be to add stack-traces to the execution traces. Or maybe count occurrences of a specific execution step within all test cases and use that as a feature in the data mining process. The data preparation step is also mentioned by Gypta et al [17]. They have their data stored in a MySQL database and used some joins to prepare the data. They also formatted dates consistently and finally converted the data into the form so the classifier can read it. This shows some important steps the need to be taken before the actual data mining can start.

Grishma et al [15] propose to use unsupervised learning to find root causes, because there are hardly any labelled datasets for this task. In some cases the defects might be labelled improperly. That is why they suggest and test some unsupervised learning techniques:

K-means K-means creates K clusters where all points in the cluster are close to each other according to some distance metric

Agglomerative Clustering The procedure starts by considering individual objects and then merging those objects which are closely placed.

COBWEB COBWEB builds taxonomy of clusters without having a predefined number of clusters.

Density Based Scan Regions in which the objects are densely present are considered as clusters, and the separating regions are characterised by less dense objects

Expectation Maximization Expectation Maximization is a statistical method that tries to estimate different parameters from unobservable variables.

Farthest First Farthest first selects the node that is the farthest from the current node to create clusters (see [10], section 2.2 for a more detailed description).

Grishma et al conclude that K-means, in combination with a different algorithm to make K-means more effective, works best in a situation where there is no labelled data set available and the root cause is still required to be found by grouping data. Based on this suggestion K-Means is uses a data mining technique, especially since it is a grouping technique. Girshma et al did not go into how they used the technique to find the root causes.

From the suggestion to use unsupervised learning by Grishma et al, and because of the promising results that data mining techniques like neural networks and deep learning have shown, Neural Networks is another algorithm that can be used in an unsupervised learning environment. Stevanovic et al [35] have used an unsupervised neural network to detect malicious website visitors. This is a classification or clustering problem into four classes or clusters: human visitors, well-behaved web crawlers, malicious crawlers and unknown visitors. They have used two techniques: the Self-Organizing Map (SOM) and Modified Adaptive Resonance Theory 2 (Modified ART2). Stevanovic et al do not evaluate the accuracy of the two different classifiers, but do get reasonable results which shows that these techniques work. What is noticeable is that they do have an initial label for each data entry. In this work it can be assumed that every symptom is the initial label of each entry and the neural network will change these labels if applicable.

Lat et al [25] also had the problem of not having a labelled dataset. They solved this using semi-supervised learning. They first identified a general set of root causes, and trained on a relatively small set of self-labelled data. They used this to form the rest of the data. To form the rest of the data, they used K-means clustering and label propagation[48]. Their problem was defined as a classification problem, which is something that could be done as well. The root cause of a problem can be classified into any of the existing states or transitions since this is a finite set. So using a small set of training data and an unsupervised learning system, a classifier could be trained that can later classify new traces into an existing state. This only works if the model of the system remains the same.

Gupta et al [17] used a J48 decision tree as a classifier to find root causes. This classifier is build into Weka [18]. Weka contains a GUI and has an extensive API which allows researchers to test and compare multiple classifiers on the same dataset with ease. Gupta et al chose this over R, Tanagra, Yet Another Learning Environment (YALE), and Konstanz Information Miner (KNIME) because of this easy to use interface. Weka does not contain a complex neural network yet. It only contains a Multilayer Perceptron classifier, which is a supervised neural network. Weka can

however be extended with new algorithms. This shows that Weka could also be a useful tool in this research. Gupta et al did not research software failure based on tests, and had to find out if a trace is abnormal or not. This is big difference with this research where it is known that a test failed but need to find out what caused that failure. One thing that could also be very useful is that they treat the data set as a sequential dataset for which special data mining techniques could be used[12]. Reidemeister et al [32] used a decision tree as well, they used the C4.5 classifier for RCA, which is the abstract form of a J48 decision tree. They also tried to find faults, but applied it again in a very different application.

Cheng et al [7] use logistic regression for fault localisation in complex networks. They look for root causes in networks of systems instead of a single system, but their approach could probably work for a single system as well. A big advantage is that it is targeted at a graph and a labelled transition system is a graph as well. Logistic regression is a supervised learning algorithm, so the data should be labelled, which is not the case in our data sets. Cheng et al solve this by using two stages. An estimation stage and a prediction stage. The estimation is done with regression analysis. They estimate the labels to get a labelled data set. In the prediction phase there is a classification problem left, that is also faced in this research, except that logistic regression can only classify into two classes.

Thung et al [38] try to find root causes of a bug by studying the treatments of bug report. Treatments can be submitting a fix to version control or marking a bug report as "won't fix". If a tester or user finds a bug, they will report it. The developer will fix the bug and commit the fix for the bug to a version control system. These two sources are used for data mining. This time it is a labelled dataset since both the symptom (the bug report describes this) and the root cause (the code fix describes this) are available. This is a very different classification problem, but a very interesting point of view and one that could actually be used in iteratively developed software. If one integrates the automatic test run with every build of the software and one compares the before and after result, labelled data could be extracted. This is a slow process and relies on humans to correctly fix the bug and not just suppress the symptom. It also requires a lot of time and effort. This way of working does provide a very fine-grained result when the classifier is trained since it can tell on line-by-line basis if that line is possibly a root cause of the symptom or not. The data mining technique that is used for this is an SVM classifier. An extension to this classifier was also used in [37].

5 Literature questions and summary

This chapter will discuss concisely what information is needed to answer the research questions posed in section 2.2. This chapter will also discuss the answers to these information questions or literature questions. This chapter can be considered as a conclusion/summary to the related work.

To answer the research questions, these literature questions need to be answered first:

1. What are methods to get *similar* cases? Where similarity is defined as testing a common area of a SUT.

2. What methods for root cause analysis are available?

3. What data mining technique works for grouping and clustering data and do they work unsupervised?

4. How can a proper test set be generated for validation?

The first question is about getting similar cases. Similar cases are test case that test the same area of a SUT. There are quite some methods that can achieve this, as is evident from the related work. All the methods are worked out in more detail in section 6. In general three main methods can be distinguished: using a data mining clustering technique; finding the root cause of a failure and group based on the same root cause; and based on a number of the same states and transitions in a test case (for example based on trace minimisation).

For finding root causes, there are also some methods shown in the related work but these are not very descriptive and there might still be quite some work to be done on this front. Of course a brute force technique could be used where a step is removed from the test case trace and if the test does not fail anymore, this step is the root cause of the failure. A second, and probably more sensible method is using SFL, or Spectrum-based Fault Localisation. The third method that was found is to use trace minimisation and if a minimal trace can be found. Using trace minimisation does not return a single state, but a shortest reproducible path that can be seen as the root cause. Finally using a decision tree and use the decision that has the highest entropy as the root cause.

The related work has identified quite a number of different data mining techniques that can be used for different purposes. For now grouping, or clustering, data and classifying data are the focus points of this research. For grouping data the following techniques have been proposed: K-means, Expectation Maximisation (EM) and a neural network called self organising map. For classifying data into the correct classes the following techniques have been proposed: J48 decision tree (C4.5 implementation), logistic regression, and SVM classifier. All these different classifiers can be tested if they work with Weka. Weka[18] is a data mining framework with many data mining techniques built in and Weka can be integrated in Atana, any of the classifiers can be applied that is defined in Weka and test it. Not all of the mentioned algorithms work unsupervised so it might be necessary to manually label the data set or use label propagation as mentioned by Lat et al [25, 48].

Generating a proper validation set is vital for this research to deliver valuable results. Mutation testing is often used to validate unit tests by creating mutants and checking if the tests will fail. If they don't fail, the tests are not checking enough. In fact bugs are introduced in the code and check if the tests catch these bugs. Since SUT with introduced bugs is needed, a mutant generator can be used to introduce these bugs in the SUT.

The SUT that is used in this research to validate Atana is a python program. After looking up the available mutation testing tools for python, the following three relevant tools were found:

- Mutmut: `https://github.com/boxed/mutmut`
- Cosmic Ray: `https://github.com/sixty-north/cosmic-ray`
- Mutpy: `https://github.com/mutpy/mutpy`

Out of these three, only Mutmut allows storing the resulting file. It mutates control sequences (like: '<' becomes '<=', and 'in' becomes 'not in') and inputs (like: integers are increased by one).

Mutmut is based on Mutpy and Cosmic Ray. Mutpy is extensively described in [9]. The main focus of Mutpy is on how incompetent mutants (mutants that create invalid code and result in TypeErrors at runtime) can be avoided and deleted from the test set. The authors of [9] also describe all different mutations that Mutpy can do, which allows for a better evaluation of the Mutmut mutation tool. If Mutpy would support just creating mutants, it would have been a better choice since it has a better theoretical underlying base, but this is not the case and therefore Mutmut is used with the knowledge of Mutpy in mind.

6 Research Method

This research uses a practical approach. It will use the design science methodology[20] to find an answer to the given research questions in section 2.2. This allows the research to be validated in a real-world environment. This also validates whether the findings are applicable in a business setting. A final benefit of this approach is that the results of the research can be adopted easily.

The practical approach consists of creating a tool that can solve the problem that is defined in the research goal: make it easier to evaluate the results of model based testing (section 2.1).

Several methods for finding root causes and grouping test cases can be evaluated. The following approaches have been identified:

- Supervised data mining techniques to classify steps into their root cause and group test by their root cause. Example techniques are a Bayes Classifier or an SVM Classifier. To apply supervised learning the root cause state field needs to be filled. To generate these labels, a brute force tactic could be used, in which every combination is tried and see if it fails and then check if this works. Another solution is to use a K-Means clustering technique as described in the related research (section 4.3). Finding how this can be done is one of the uncertainties that must be researched before data mining for the root cause can start.

- Unsupervised data mining techniques to group test cases and find root causes. This can be done in two ways. Just grouping test cases using techniques like K-means. After the tests have been grouped, it can be used to evaluate the root causes. The other way is to use the data mining technique to find the root cause and then group the test cases on the root cause again. Using an unsupervised approach avoids having to label the data, but might also be less accurate.

- Trace minimisation is a technique that can shorten traces to make them easier to interpret. This also allows for comparing traces that look the same, which can be merged into the same group. Traces could be the same if two have the same ending sequence. This does not provide a root cause yet, but neither does the unsupervised grouping data mining technique.

- Delta Debugging to get a minimal trace again and all minimised traces that are the same, should have the same root cause.

- Spectrum-based Fault localisation(SFL) is a very intuitive way of finding the root cause. SFL compares failed tests to passed tests to see if certain parts of failing tests do not occur in passing tests. If there is a part in the failing test that does not occur in the passing test, this is most likely the cause of the failure. SFL is further explained in the related work and section 8.4.1.

- Using a decision tree to see what steps have the most influence in if a test case passes or fails. The steps with the highest entropy, or influence, are the root causes of the failure of that particular test.

To be able to start using data mining techniques, the data has to be pre-processed in a table with the following columns:

- test trace step 1
- ...
- test trace step n
- number of steps in the test case
- number of reoccurring steps in the test case
- root cause

The column with the root cause is the label of the trace (in which "class" the trace must be classified). The rest of the columns are the features that can be used for classification. This layout is used in Weka, which is integrated to simply test different data mining algorithms. How the table is filled with values depends on the settings of the data mining analysis service. The different options are described in section 7.5.

Many different solutions have been identified, but the scope is limited to validating two solutions. At first Spectrum-based Fault diagnosis is tried on MBT. This method has proven to work in other applications and is therefore very promising. Validating the effectiveness in MBT applications is a great addition to the state of the art and might answer the research question in itself. The second approach that is tried is the unsupervised learning approach. Data mining and machine learning appear great

tools to solve many problems and finding the root cause of the failure could be one of these problems.

6.1 Validating the solution

When a possible solution has been implemented, it needs to be validated. The validation consists of running the implemented solution on a program that has a known set of bugs with known root causes. This test results in a score that can be compared between different solutions.

The different solutions are scored according to the following scoring criteria:

- Number of bugs found
- Number of failing tests correctly grouped
- Duration of the search / resources used
- Regular precision and accuracy metrics using false positives, true positives, false negatives and true positives.

The method that is awarded the highest overall score will be the considered the method that answers the research question: What method can effectively group similar failing test cases to make diagnostics easier?

To be able to gather these scores a SUT with an introduced bug must be created. To create the SUT with specific types of test cases that should be found is quite a difficult thing. Defining test cases is notoriously hard, and defining good test cases is even harder. To create the program and introduce bugs, a passing SUT with an existing model and test cases can be used. The MBT system can generate a lot of test cases. After having verified that everything still works, bugs are introduced. To do this mutation testing is used. This process creates mutants of the working programs. A mutant has a specific line of code changed in some way, which possibly makes it an invalid program for which the tests should not pass. Many mutants could be created to create a test set of multiple systems.

Note that these faults/bugs are introduced by the mutant generator. After all mutants have been generated, a proper test set is available that will allow for proper testing of the implemented solutions described in the previous section.

7

Design

This chapter describes the design of Atana. Atana, the software that will collect test data and help in trace analysis, works with several components. Some of these components are also discussed separately. Finally security is discussed as this is part of the whole design and is applicable to all components.

Atana is the core of the solution to the problem described in section 1.1. Atana receives and stores the information required, like the test model, the test traces, the coverage information and log files. Atana also manages/orchestrates the components that execute the analysis. These components, or analysis services, provide the implementation that execute the analysis and returns the result. The analysis services can be implemented as separate programs, or micro-services. This separation allows the analysis services to be implemented in the language and framework that fits best with that specific implementation.

To be able to receive and store the aforementioned data, Atana must meet a few requirements. First it should expose a REST endpoint where the data can be sent to. Furthermore it should be able to parse and validate the data that was send to that endpoint. After parsing the data, it should be stored in a database. The last requirement for Atana is that it should be able to send specific data to an analysis service on demand and return the result.

For the implementation of the requirements for Atana the following libraries and tools have been chosen. First Jackson[1] is used for parsing the JSON data into data objects. Jackson is a serializer and a deserializer for JSON and XML. The deserialised data objects can be stored in a database. The database that has been chosen is a MySQL database since this is the same database that Axini was already using. Finally Spring[2] is chosen. Both for its ease of use for services, REST controllers and database repositories. And also because Spring can execute REST-calls to the different external micro-services through a easily testable component. Combining these libraries turns into the architecture in figure 7.1.

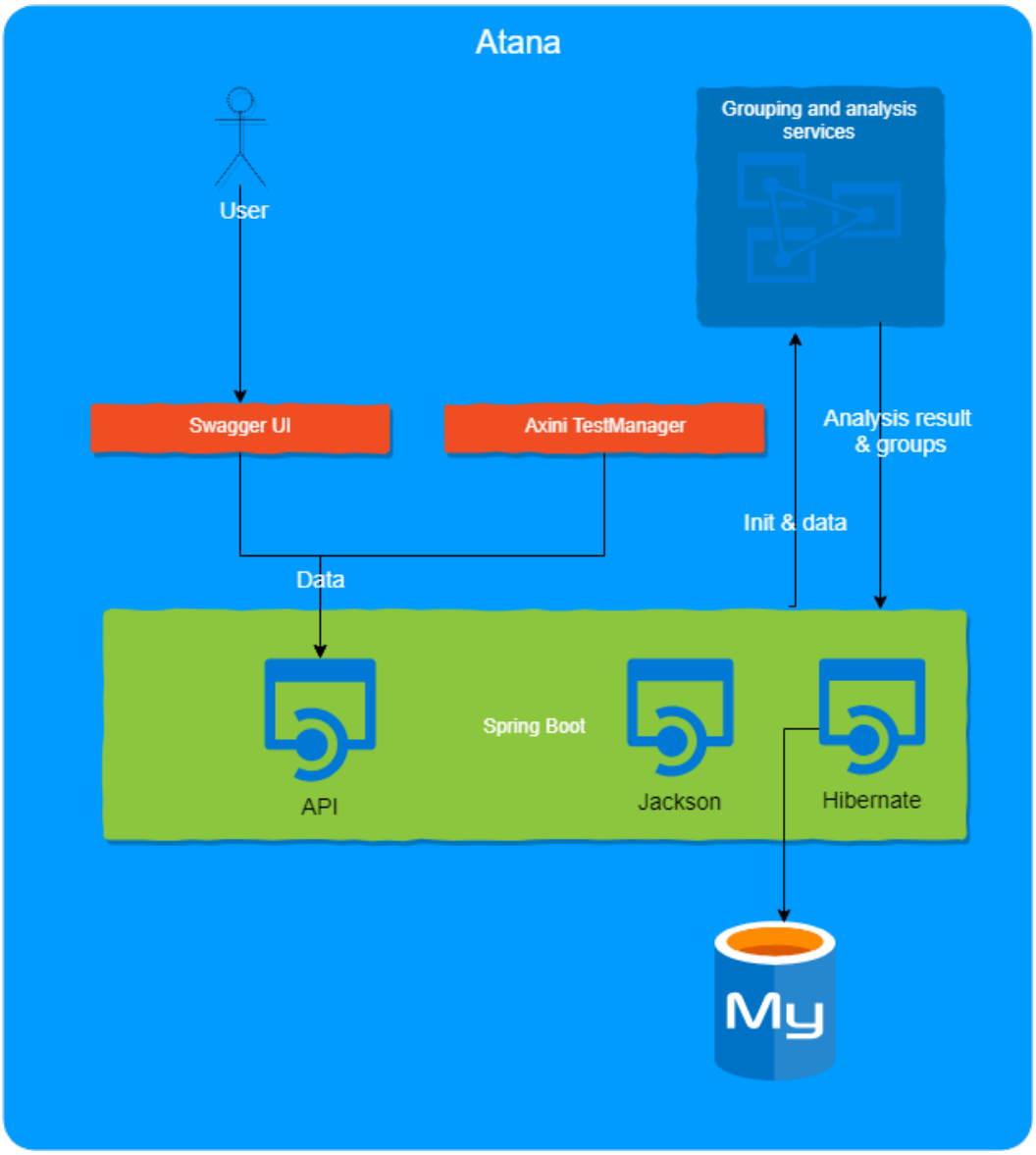

Fig. 7.1.: Architecture describing Atana. Swagger, Spring Boot, Jackson, Hibernate and Axini TestManager are existing tools and libraries used to build Atana. The grouping and analysis service and the API in the Spring Boot component are newly created.

7.1 Architecture

The architecture for Atana is shown in figure 7.1. The user can interface with the system through Swagger UI. Swagger UI is a tool that helps in documenting APIs and also allows to send requests from the web interface. The documentation of the API is also made available through Swagger UI. Swagger UI can be accessed through a browser. Other applications, or a separate front-end, could connect to Atana without using Swagger UI.

Axini TestManager generates and runs the tests on the SUT. Data from Axini TestManager is needed in Atana. The data (the model, coverage information and test traces) can be downloaded as a set of JSON files, which should be submitted to Atana. This can be done at once, or in several steps. This data flow from the Axini TestManager to Atana is shown as a single request, called data, in the architecture.

The API layer is the layer that is the gateway to the rest of the system. It provides access to the database, which is the core of Atana. To be able to write queries to the database the Hibernate framework is used. This allows for type safe queries and the queries (SQL) are automatically generated. The other major component of Atana is Jackson. Jackson can convert JSON document into data objects and the other way around. This is required to be able to store the JSON requests in the database with Hibernate. This whole set of components is running on a framework called Spring Boot. This framework provides the REST implementation for the API, easy registration and usage of services and seamless integration with Hibernate.

The analysis services are separate micro-services. These services are described in more detail in sections 7.4 and 7.5. The analysis services provide the implementation that execute the analysis and returns the result. To do this the service needs data. This data is supplied by Atana through two requests. The first post request sends all available test result data to the service that is required, like the test model and the test traces. This request is shown as the init step in the architecture. The second post request sends the test case that needs to be analysed. This request is shown as the data step.

7.2 Component interaction

This section describes the interaction between the different components. To describe this interaction, an example is used. This example describes a test run that was

[1] https://github.com/FasterXML/jackson
[2] https://spring.io/

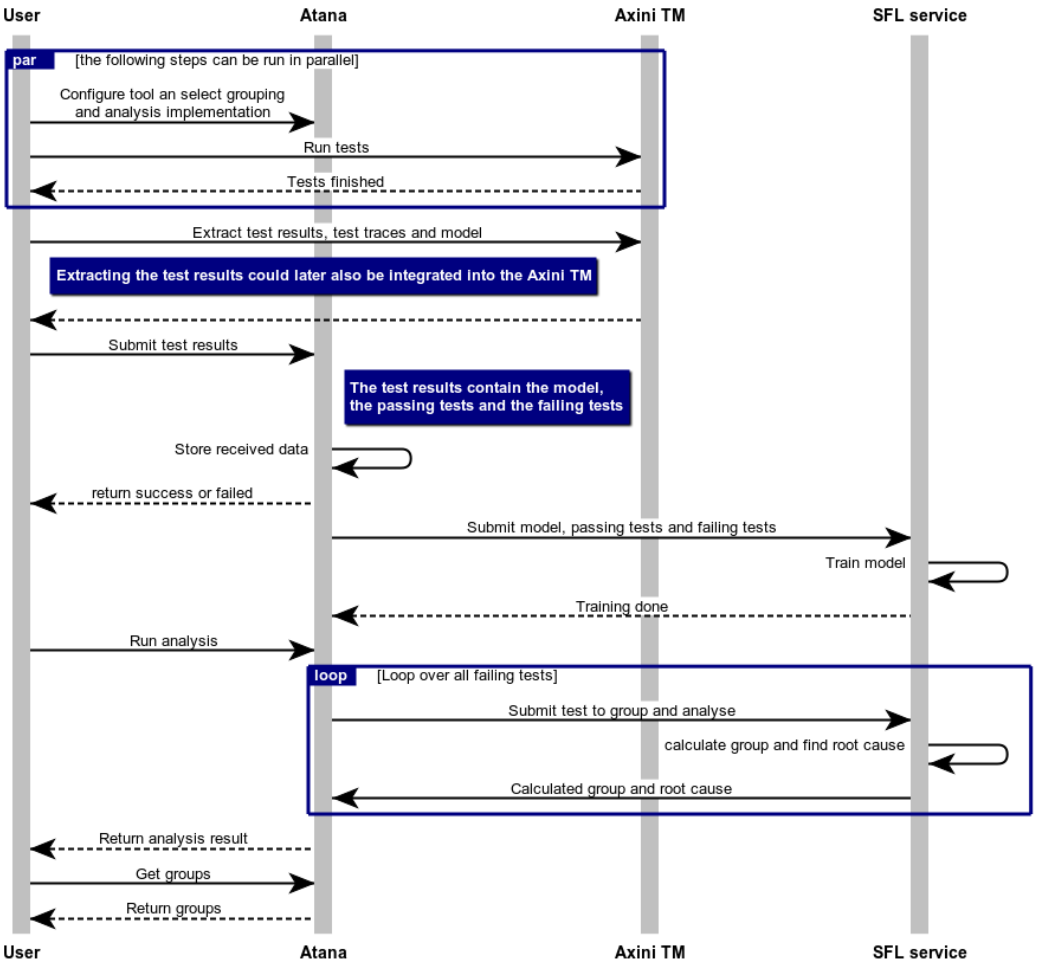

Fig. 7.2.: Sequence diagram that represents the actions executed by the different actors during the usage of Atana and the related services. The solid arrows represent the requests that are started by the current actor. The dotted arrows represent responses to a request.

executed and is later analysed by Atana. The analysis process is described through the interaction between the different components. The different requests are presented in more detail in the sequence diagram in figure 7.2. The actors in the diagram represent the different systems involved, and the user that operates the whole system.

The first thing that has to be done is to prepare the systems. These first requests can be run in parallel and in any order. The configuration step for Atana consists of setting the implementation for the analysis service. For this research all analysis services use the *GroupingAndAnalysisServiceRestImpl* implementation which redirects the methods in the class to a rest endpoint. In this case Atana is configured to use SFL and send the requests to localhost port 8000 since this is where the SFL service will run.

In the meantime the SUT and the test runner (which is a part of Axini TestManager and is not incorporated in the sequence diagram since it is mainly out of the scope of the current research) are started. These are restarted for every mutant. Axini TestManager instructs the test runner to execute the tests. At this point Atana is configured and the tests are run. The results of the tests are still only stored in Axini TestManager.

To get the data in Atana, the user has to extract the test results and model and send these to Atana. To do this, Axini TestManager has a download function. This allows for downloading the model, a model for each test run with coverage information and a set of traces. Having stored this information for later reference, the data can be sent to Atana.

The order of the submission of test results is not important. The results consist of several files. The model is a JSON file. The contents of this file are sent to Atana by the user. Atana parses the file, validates it, and stores it if it is valid. How the model is validated is described in more detail in section 7.6. If the model is valid, the response is an HTTP 200 status (OK). If the model is invalid, an appropriate HTTP 400 status (BAD REQUEST) will be returned with an error message. This allows the user to fix any issues that might have occurred and try again. The traces are also a JSON file which is handled similarly to the model. The logs of the SUT can also be sent to Atana and can be stored. These could be used in future analysis services. To be able to sent multiple log files, this is also sent as a JSON file which contains the name of the log file and the content as key-value pairs.

At this point all data is stored in Atana, and every step until this point are generic and are required for all of the different analysis services. The next steps in the process contain small deviations for different analysis services. The moment that the user has sent all data to Atana, the data is sent to the configured analysis service; in this case the SFL analysis service. The data that is sent is the same data that was previously stored in Atana. The SFL analysis service stores this information in memory since the information is already persisted in a database by Atana. This also allows for quick access to the relevant data. When Atana has sent all the data to the SFL analysis service and informs the SFL service about this, the SFL analysis service starts to train a model. The model of SFL is a matrix with transition or step coverage, which is explained in more detail in section 7.4. That section also contains an example of this matrix and a description of how this matrix is used. During the training process Atana is kept up-to-date by progress-calls. A progress-call is an HTTP POST request containing a double with which represents how many percent of the data has been processed.This could be used in a front-end to show a progress bar but the front-end design is out of scope for this project.

When the training is done, and the progress reaches a hundred percent, the user can analyse a single test trace, or all traces that belong to a test run. This can be done by submitting another request to Atana with the index of the failed trace. If the index is -1, Atana will extract all failing traces and send them to the SFL analysis service. The SFL analysis service will take the current trace that is required to be analysed and returns the result. This result is again stored by Atana for future reference and is also returned to the user.

At this point, the user knows the answer to why some traces have failed. The received analysis result indicates what the faulty transition, state or steps were in the submitted test trace.

This design is valuable since it allows for a lot of reuse of existing parts, which in turn allows for rapid development.

7.3 Tool and language motivation

This section discusses the motivation for selecting different tools and programming languages for the different components. The selection of a tool and language depends on several aspects. One of the main aspects is security. This is always an important topic in any project, and can often be an after thought (see also section 7.6). A second important aspect is interaction between tools and languages. The combination of tools should match and interact. This should not require compatibility layers which reduce the speed of development. This is important because the goal of the current research is to find results, not creating an optimally working robust system.

To find the right tools and languages, different options are compared. These options should all be at least be partially familiar to be able to quickly start working on the development of Atana and related components. A second requirement is of course the applicability of a certain language, tool or framework to solve the problem at hand. When the comparison is done, the best tool or language is selected and used.

First the language and framework for Atana were chosen. A familiar and good base for any REST-based server is Spring Boot. This framework allows to quickly create a simple application that can receive requests. What makes it easy to handle requests is the approach to mapping a url to a method in the application. Spring does this with annotations on all of the REST controllers that contain the relative path to which that method will respond. We chose to use Hibernate as the ORM (Object/Relational

Mapping) since Spring and Hibernate work very well together and Hibernate is the default database manager for Spring. Hibernate can create database queries from data objects. Hibernate also allows for type safe queries, which means that no casting exception will occur at runtime. This helps in preventing mistakes made by the programmer. Spring and Hibernate use Java as a programming language, but Java has some flaws regarding nullability[3]. This is solved in Kotlin, which is interoperable with Java. Kotlin solves this problem by making it explicit if a variable can be null or not. This will prevent mistakes by the programmer and therefore the quality of Atana.

For the separate analysis services, a new language and tool can be selected. For the data mining service, Weka[18] was selected. This is a tool that allows for comparing data mining algorithms and a lot of experimentation with data mining as a whole. Weka is written in Java, so Kotlin and Spring are chosen again to turn Weka into a REST service. For the SFL analysis service, a different language is chosen. SFL contains many simple loops, where each iteration only performs few commands. To optimise for speed a low level language[4] was selected. To keep the service secure, C and C++ were deemed to be too vulnerable, especially from a memory standpoint. Rust[5] was suggested by a security course on the Univerity of Twente[6] as a secure alternative. This is why Rust is selected for SFL. To turn the Rust application into a REST service, Rocket[7] is used. This is very similar to Spring Boot with fewer functions. Rocket allows for the creation of REST endpoints with annotations. The serialisation and deserialisation is provided by Serde[8]. This allows for the deserialisation of the JSON requests that Rocket receives. Rocket and Serde follow the unix philosophy: *Make each program do one thing well*[14], section 1.4.

7.4 SFL analysis service

This section describes the SFL analysis service in more detail. How SFL works is also described in the related work, section 4.2 and in more detail in section 8.4.1. The SFL analysis service is created to analyse traces using the SFL algorithm. It has to return the state, transition or steps that cause the failure of the test case that is being analysed. To do this, the SFL algorithm is implemented using Rust. SFL does not

[3] Java variables can be *NULL*. If a method is called on a variable that is *NULL*, Java throws a *NullPointerException*. The fact that objects can be *NULL* is called the nullability of objects.

[4] A low-level language is a programming language that uses little abstraction to the cpu's instructions and has therefore little overhead which makes the language fast and gives the programmer more control over the execution of the code.

[5] https://www.rust-lang.org/

[6] `https://osiris.utwente.nl/student/OnderwijsCatalogusSelect.do?selectie=cursus&cursus=201600051&collegejaar=2017&taal=en`

[7] `https://rocket.rs/`

[8] `https://serde.rs/`

use the order of the transitions or steps. To introduce the order of execution in the algorithm, pairs of transitions or steps are also included. The maximum length of these pairs can be specified in the configuration.

The implementation consists of several steps. First the training stage will create a matrix with the occurrence of each transition or test step in each test trace. This matrix is human readable and looks similar to the matrixes presented in papers such as [2]. An example is also included in table 7.1. The analysis stage is the second stage. This stage calculates the similarity between the error vector and the test case being analysed for each transition. There are different metrics available to calculate this similarity. Ochiai[2] will be used by default, but this could be substituted in future research by other metrics[1]. To determine if a transition is causing the failure, the similarity with the error vector can be treated as a probability that a transition is faulty. To be faulty, the similarity to the error vector of a transition must be above a configurable threshold. The similarity values are included as the last row in the table in table 7.1.

This approach to analysing a trace is valuable because it is structured approach to analysis. It is also human readable and verifyable. Another value to this approach is the application of a white box analysis technique in a black box environment using model based testing.

Suppose the model in figure 7.3 is send to the SFL analysis service. This model does not contain any loops which makes sure there is a limit to the number of test trace that can be created. In this case there are four test traces:

trace 1 $init! \rightarrow a? \rightarrow a? \rightarrow c!$ (verdict: passed)

trace 2 $init! \rightarrow a? \rightarrow a? \rightarrow d!$ (verdict: passed)

trace 3 $init! \rightarrow b? \rightarrow b? \rightarrow c!$ (verdict: failed)

trace 4 $init! \rightarrow b? \rightarrow b? \rightarrow d!$ (verdict: failed)

The four test traces are also send to the SFL analysis service. This should result in the matrix shown in table 7.1. The default threshold is set to 0.85, which means that SFL should return state 3 as the faulty state. This is because the similarity of both the transition into the state (b?) and out of that state (b!) are above the threshold. If trace 2 and 4 were to fail, the test step d! should be returned as the faulty transition.

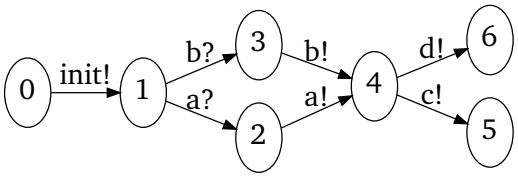

Fig. 7.3.: SFL example model

	!init	?a	!a	!c	!d	?b	!b
passed	x	x	x	x			
passed	x	x	x		x		
failed	x			x		x	x
failed	x				x	x	x
	0.5	0.0	0.0	0.5	0.5	1.0	1.0

Tab. 7.1.: SFL matrix which can determine the faulty transition(s)

7.5 Data mining analysis service

The second analysis service implementation analyses traces using several data mining techniques. This service therefore has to support these different techniques. It should also support returning the analysis result in the same format as the other analysis services do. This is the only format supported by Atana. To be able to support different data mining techniques, WEKA is used.

This service also contains several stages in the implementation. First there is the training stage. This stage will use all data[9] to train the unsupervised data mining technique. After training the same data is also used to create a lookup table with the different results for each test trace. The second stage is the analysis stage. This stage can be used to run a second data mining algorithm to find the root cause of a trace given all the test traces in the same cluster. How the traces are clustered is configured by the data mining technique.

This second approach is valuable in that it can use hidden patterns to improve the analysis of traces.

7.6 Security

To make sure that the system is designed and implemented in a secure way, security and validation are one of the design aspects from the start. McGraw has emphasised[28] that software security is often done too late, and that it should be part of the design stage of a project.

[9] Since the data mining technique and model are not evaluated by itself, all training data can be used. Especially since the data mining technique is an unsupervised clusterer, which should prevent overfitting by design.

The security design should make sure that the system is memory safe. Especially due to the speed considerations for SFL memory safety is important. The security design should also prevent null pointer exceptions. These happen due to carelessness of a programmer that a value can be null. Finally the security design should also make sure that invalid data is rejected early.

The programming languages that were used in Atana are Kotlin[10] and Rust[11]. Kotlin is a language that compiles to java byte code and uses the JVM. Kotlin is a statically typed language with type inference, which means that not all types have to be specified as one would have to in Java. The types also make nullability explicit. During the deserialisation step of the JSON documents into data objects, all strings have to be converted into objects that have specific rules about their formatting. This allows for some type safety in the JSON. A url for example must have a protocol and a main body. If this is not the case, the request will return with HTTP status 400 (Bad Request). Rust[29] was specifically designed for memory safety without garbage collection. Rust tracks objects and automatically deletes them when they are not used anymore. Besides memory safety, Rust also guarantees that there are no data races. While these safety guarantees hold, it is still a low level language that is also very fast. Rust does not allow for values to be null, which makes the language inherently null-safe.

Besides security by language, the data objects are also validated manually. To do this, Java Bean Validation (JSR 303[12] and JSR 380 [13]) is used. This library allows to add constraints on fields with annotations. These annotations are then used for validation. All requests are validated in the REST controllers. Along with Java Bean Validation some Kotlin specific features are used. Kotlin has two precondition functions[14]. These functions are called *require* and *check*. They throw an exception when the condition is not met. This allows for adding specific preconditions in the controller when appropriate. The preconditions are executed in production software in contrast to java asserts, which are only executed when the software is running tests.

Using these languages makes sure that the programmers cannot forget about nullable variables. The validation of data is done through annotations on the model objects. Each field can have an annotation which for example limits the range of an integer, or requires a collection to consist of at least a single element.

[10] https://kotlinlang.org/
[11] https://www.rust-lang.org/
[12] `http://beanvalidation.org/1.0/spec/`
[13] `http://beanvalidation.org/2.0/spec/`
[14] https://github.com/JetBrains/kotlin/blob/master/libraries/stdlib/src/kotlin/util/Preconditions.kt

This security design helps in understanding the data better and helps in the adoption of security by design in new projects and shows its importance.

8 Implementation

This chapter details the implementation of the design from chapter 7. It contains all components that have been presented in the design, and could be used to help reproduce the experiment. Finally it discusses some problems that were discovered during the implementation of the design and how these might be resolved.

The implemenation of the different services can be found on GitHub: `https://github.com/marty30/Atana`.

In general, the test driven development approach is used. This helps in writing software in such a way that it does what you want form the start, and no long detours have to be made. For the execution TDD, the minimal change approach is used. This approach means that every commit tries to change as little as possible to make a test pass and once the test passes, the test is updated if applicable. This is a repeating process until the test is fully written and passes. TDD helps in gaining a high coverage test set for working software. It also helps in refactoring and lets the programmer keep track of what to do next when an interruption has occurred.

8.1 User and data collection

In the sequence diagram in figure 7.2, the first actor is the user. The user could be the tester that uses Axini TestManager to test their application. The user is therefore interested in the result that is generated by Atana. During this research, no user was present but the actions were executed by a script that collects the data for analysis. The data consists of the mutated SUT, the model, the test traces, the log files and the analysis result. Data collection is required for this research to compare the different implementations and to validate the effectiveness of the solution that is designed and implemented.

To collect the data, the python script starts the SUT, it triggers a test run in Axini TestManager, and waits for it to be finished. Once Axini TestManager is finished running the tests, the script downloads the traces and model from Axini TestManager and collects the logs from the SUT. All this information is stored as raw data in a folder[1]. When the data is collected, it is sent to Atana as if the script is the actual user of Atana.

[1]This raw data folder can be requested at the author

After all mutants are started, the tests are run and the data is collected and shared, everything is prepared for the analysis stage. This stage is started by a second script. It requires a list with all test runs and uses this to individually send all test cases of a test run to Atana. Atana returns the analysis result. This result is collected in such a way that the results are easily comparable. To make the results more easily comparable, the results are stored in a spreadsheet. The spreadsheets can be requested from the author. A summary of these spreadsheets is also shown in chapter 9: Results and discussion.

The result of this data collection is a set of files that can be used in other research projects, and allows for evaluation of the different solutions.

8.2 Atana

This section describes the implementation of Atana. Atana is created to validate and store data, and to orchestrate all components. Atana can also generate statistics that help in performing the current research. This is the main entry point for the user.

Atana is built with Spring Boot. This is a framework that allows for building simple REST endpoints and is interoperable with Hibernate, which is used as a database manager. These two points solve the main functional requirements that Atana has to fulfil. To parse the JSON document that is sent to the REST endpoints of Atana, Jackson is used. This parses a JSON document into a Java or Kotlin object. Parsing documents helps in persisting everything to the database and in accessing only parts of a certain object. For example getting a single string representation of a step is implemented in Kotlin using the label and the label parameter fields in the traces. This would be less maintainable and more error prone if no parser was used and JSON objects were interpreted as strings.

To send the data to the other required services, we chose to use another feature of Spring called the *RestTemplate*. This class can send REST requests to other services. The calls to other services can also be mocked during testing to test all code in isolation. All calls that run between Atana and the analysis service implementation are implemented using this class.

Springfox's Swagger 2 implementation[2] is used to allow easy interaction between Atana and the user. Swagger is a documentation tool that allows users to try sending requests. The user can simply copy the JSON and fill all required parameters to interact with Atana. And because Swagger is a documentation tool, it also takes

[2] `https://springfox.github.io/springfox/`

care of the documentation for the API. An example of what SwaggerUI looks like is shown in figure 8.1. The different controllers are shown and can be expanded. Each controller contains different operations. Each operation already contains a short description of what that particular operation does. When an operation is expanded, as shown for the operation */statistics*, a request can be made by clicking the *Try it out!* button. This will execute the operation with the provided data if applicable.

This implementation of Atana allows for storing the data that was received from Axini TestManager. It also allows to send this stored information to an analysis service once this is desired. Finally this implementation of Atana allows to gather statistics over the stored data and return them to the user as a JSON document, which in turn helps to optimise the work in the current research. The fact that it is known if a test run only contains failed or passed tests is an example of this optimisation since these test runs do not require analysis.

The next sections will describe the different analysis services that have been implemented.

8.3 Baseline analysis service

This section discusses implementation the first analysis service. This service functions as a baseline for the other analysis services. The baseline is useful to see if other services work better or worse compared to this baseline.

The baseline service is based on how Axini TestManager implements the determination of the verdict. TestManager marks a test run as a failure the first time it receives an unexpected output. So a logical assumption would be to assume that the last test step in the trace is the faulty test step. This baseline is implemented as the last step analysis service. This is not a separate micro service since the service would have too much overhead for such a simple algorithm. This service just extracts the last test step and returns this as the faulty test step. There is no need for a training stage in this service.

This implementation allows to determine if the other services perform better or worse than just assuming the last test step is the faulty test step.

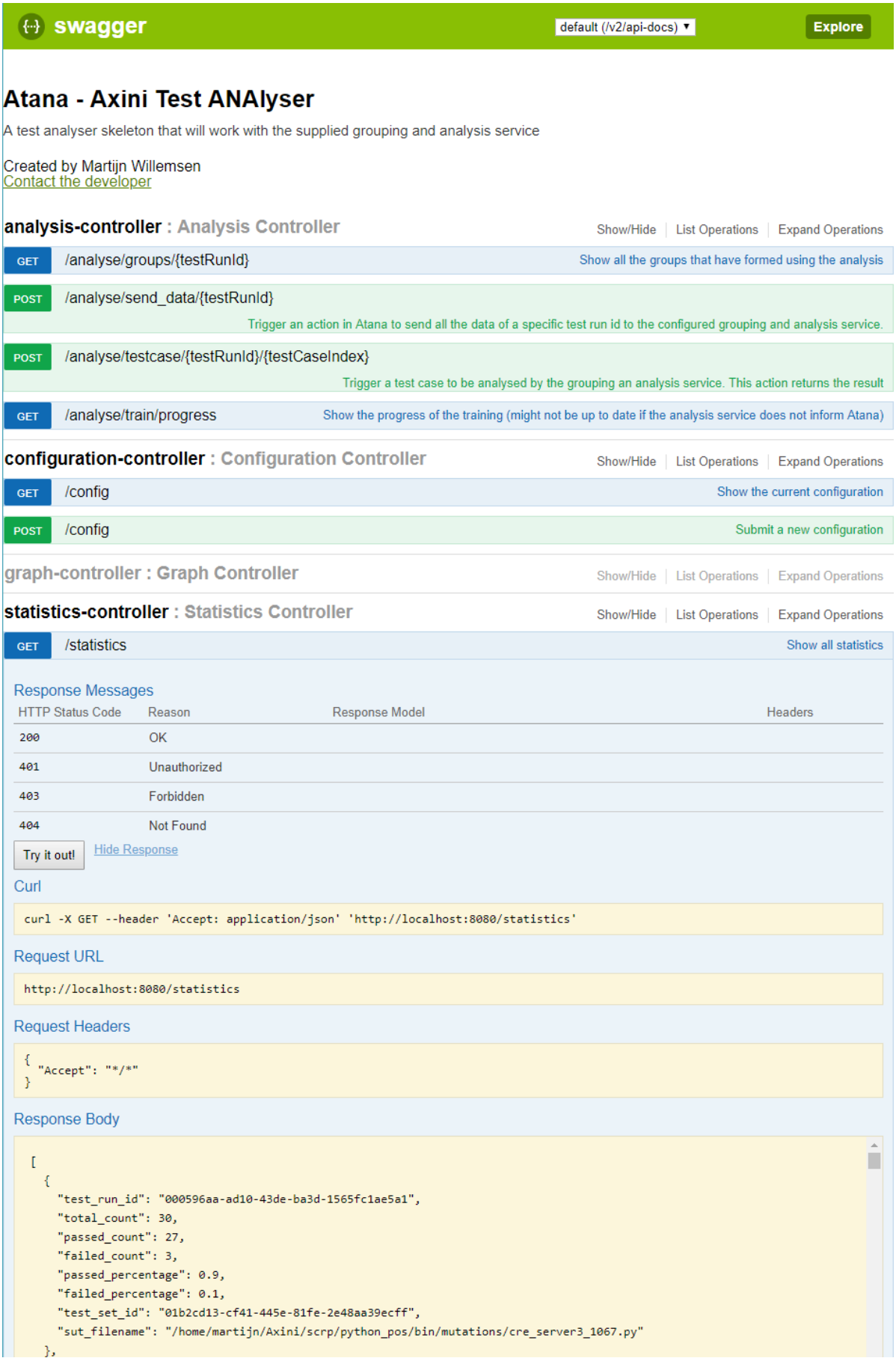

Fig. 8.1.: Example of SwaggerUI with documentation

8.4 SFL analysis service

This section describes the implementation of the second analysis service. This analysis service uses spectrum-based fault localisation (SFL) to find the faulty step or transition.

8.4.1 SFL description

Spectrum-based Fault Localisation or SFL is an approach to finding the root cause of a failure. In short, it looks at all steps and determines which steps are mostly occurring in failing test cases. If a step does not occur in a passing test case and only in the failing test cases, it is the likely cause of the failure. To do this by hand, a table can be made with the steps/transitions in all test cases on one axis, and the different test cases on the other. The cells of the table show if a particular step is part of the corresponding test case. An example of such a table is shown in table 7.1. In this example there are four test cases and seven different steps that can be executed. The test cases are generated from the model in figure 7.3. In this case, it is easy to see that two steps must be causing the failures: ?b and !b. These two transitions must therefore be the root cause of the failure and should be reported. All test cases that use these transitions should be grouped in the same group, because they all have the same root cause.

Originally SFL was developed to work with lines of code. Coverage data could be used with (unit) tests to fill the table and automatically identify faults in the code. Because the current research is working in a black box environment, code is not available. To apply this same technique in a black box environment, the assumption was made that the lines of code can be replaced by the steps or transitions in the test cases, since they represent the code.

Only Weiglhofer et al have tried applying SFL to MBT before and they did so successfully [41]. An implementation of this approach is required to validate if the SFL approach helps in identifying the faulty steps and transitions.

The service is implemented using the Rust programming language, since it is a low level language that is optimised for speed and security. The REST-endpoints are created using the Rocket framework[3]. This results in a micro-service that can be run on any server (for example using the container platform docker[4]), is fast and memory- and type-safe.

[3] `https://rocket.rs/`
[4] `https://www.docker.com/what-docker`

To determine which steps contribute to a failing test, a probability for each step is used. This is the probability that a certain step is faulty. If the probability that the step is faulty is higher than a predefined threshold, it is returned if it is part of the test case that is currently being analysed. This solution is based on Zhang et al [46], Wong et al [42] and Abreu et al [3]. Calculating the probability is done using Ochiai's similarity coefficient. The procedure to calculate the Ochiai similarity coefficient is extensively described in [2] and also included in equation 8.1.

$$
\begin{gathered}
S(j) = \frac{a_{11}(j)}{\sqrt{(a_{11}(j) + a_{01}(j)) * (a_{11}(j) + a_{10}(j))}}, \\
\text{where } a_{pq}(j) = |\{i \mid x_{ij} = p \wedge e_i = q\}| \text{ and } p, q \in \{0, 1\} \\
x_{ij} = p \text{ indicates whether block j was covered } (p = 1) \text{ in test} \\
\text{run } i \text{ or not } (p = 0). \text{ Similarly, } e_i = q \text{ indicates} \\
\text{whether a run } i \text{ was faulty } (q = 1) \text{ or not } (q = 0).
\end{gathered}
\tag{8.1}
$$

During the training stage of this service, a table is built that is similar to table 7.1 without the similarity coefficients. This table is the coverage matrix that is required to calculate the Ochiai similarity coefficient. The Ochiai similarity coefficient for the test trace under analysis is calculated during the analysis stage. To create this table some actions have to be taken. Each test has to be split up into the transitions it consists of (1). For each test and transition, the coverage must be determined: if a transition is covered by a test, *true* is added to the coverage matrix for that transition, otherwise *false* is added(2). The verdict of each test must also be added to the coverage matrix as the error vector(4). Finally for some traceability, the test id is also added(3). When this is done, the coverage matrix consists of a list of test ids, a list of verdicts and a map of transitions (as the key) and lists of booleans that represent if the test at that specific index covers the transition or not. An example of this is shown in figure 8.2.

This coverage matrix can be used for the analysis, but to print it, a library is used. This library requires a table data structure. To create a table, the transitions are added as the first row (to generate a sort of header for the table). After the header row, each test is added as a row. To create this row the map of transitions and list of booleans is iterated.

With the base of SFL implemented, the first tests can be analysed. The SFL analysis service can also be tested for interoperability with Atana. The first tests we performed highlighted some interpretation problems. After these were solved, some functional problems remained, which are discussed in the following section.

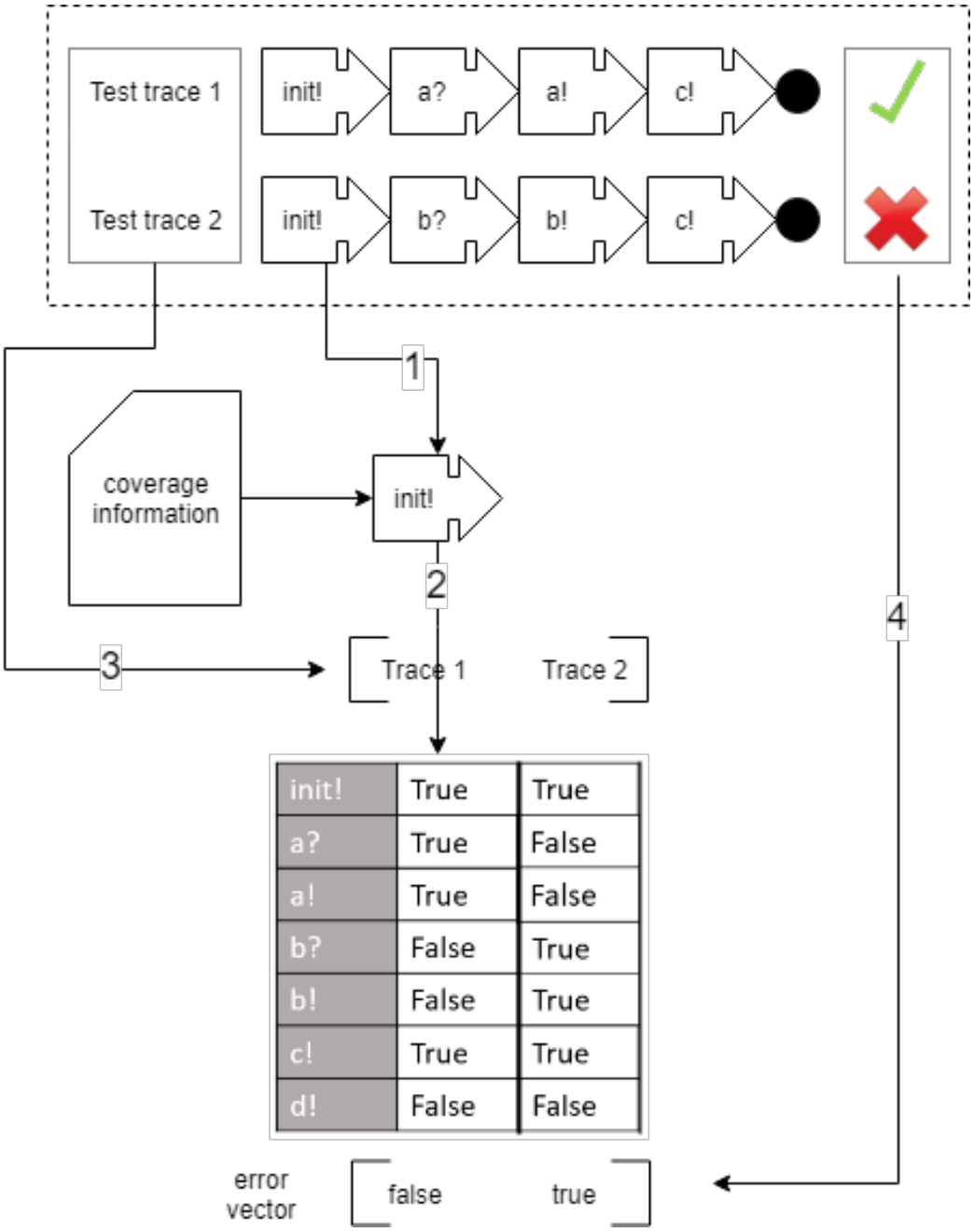

Fig. 8.2.: SFL coverage table construction. The numbers represent the actions as described in section 8.4.1.

8.4.2 Problems with SFL

During our tests of the Spectrum-based Fault Location method, we found a number of shortcomings that limit its application in the context of model-based testing. The first problem is that basic SFL assumes that order of execution does not matter. This might not always be the case and the fact that order is not considered is a shortcoming of basic SFL. The order is not taken into account because SFL just looks at the similarity for a single component (a column in the matrix), not at the location of the column or the combination of the columns.

More researchers have identified this problem with SFL and have proposed different extensions to SFL [46, 47, 3]. Using call sequences[47] is not possible in the context of model based testing. A call sequence resembles a stack trace. The call sequence consists of a set of function calls. Every function that calls another function is added to the sequence. This creates a sequence of functions that in the end calls the current function, The approach by Zhu et al[47] requires knowledge of the SUT and who calls which function, and this is also not available in a black box testing environment.

To somehow include the order of execution sequential steps are also considered as a single cause for failure. This means that two sequential steps in a trace are combined as a component in the table of SFL. If a test case contains both steps sequentially, it uses that special component and this knowledge is used in the SFL procedure. If there are two steps that cause the failure that are not sequential, this fault is still missed in this implementation of SFL. A solution would be to add all combinations of steps as a single step. This solution is not feasible, since it would significantly increases the size of the data to analyse.

Another disadvantage of SFL is that multiple test cases are required to pass to get relevant results. In a real world scenario it might happen that all tests fail. In this case, SFL cannot perform proper analysis and will return all transitions. This does not provide any new information. If all test cases fail, SFL cannot calculate a difference between the passing and the failing steps, because there are no passing tests. This means that the similarity will always be 1 because all occurrences of a certain transition always happen in a failing test trace. A fallback could be used, like returning only the last transition, but this is not used during this research to keep the comparison pure. How many passing tests are required to be able to analyse a single failing test could be investigated in future work.

Finally, SFL requires a determined threshold. This threshold is hard to determine. To translate the possibility of a failure into a number between 0 and 1 is hard because

there is no reference on when a value is high or low. For this research we used an estimate. The estimate can be verified by running the data extraction process with multiple different values and on more data. Different projects with different models could require different thresholds because of the number of steps and transitions in a model. Another solution could be to make the threshold dependent on the similarity values of the other transitions.

8.5 Data mining analysis service

This section describes the implementation of the data mining analysis service. This implementation helps in validating the possibility that a data mining technique can help in analysing the traces of the failing test cases. This analysis service is written in Kotlin and uses the Spring framwork again for the REST endpoints. To run the different data mining algorithms, Weka[18] is used. Weka is a data mining framework that is developed for learning. Weka was chosen because there are already many classifiers, clusterers and desicion trees built in and more can simply be added. It is also used frequently in academics.

To run data mining algorithms using Weka, the data must be read by Weka. Weka cannot read the JSON files that contain the traces, which is a problem. If Weka does not have any data, it cannot be used. To get data into Weka, the arff file format it required. The arff format is a tabular format. First it contains the headers for the table with their data format. After this header definition, the arff file contains the data. This is the table contents. These preferably contain only numbers and enum-values, but could also contain strings. If the data contains strings, many classifiers and clusterers are not suitable anymore.

To turn the original data into the arff file format several methods could be used.

- The first option would check for occurrences of a transition or step in a trace. This is very similar to the way SFL converts the data into a table. An example of what this would look like is shown in table 8.1.

- A second option could be to count occurrences instead of just checking for them. An example of what this would look like is shown in table 8.2.

- A third option that could be used is focused more on the order of the steps. This third option replaces the check for the occurrence of a transition with a check for the index of the first occurrence of that transition. An example of what this would look like is shown in table 8.3.

Tab. 8.1.: Trace to table conversion using the first approach: occurrences

!init	?a	!a	?b	!b	!c	!d
x	x	x				x
x	x	x			x	
x			x	x		x
x			x	x	x	

Tab. 8.2.: Trace to table conversion using the second approach: count

!init	?a	!a	?b	!b	!c	!d
1	1	1	0	0	0	1
1	1	1	0	0	1	0
1	0	0	1	1	0	1
1	0	0	1	1	1	0

Which method works best is left as future work, since the scope of the current research is mainly focused at SFL improvement. For this research the first approach, where each transition is checked for occurrence, is used. The occurrence approach is chosen because it resembles the SFL table most. Future research on this area could validate the importance of the approach to JSON document conversion. Future research could also look for an improved conversion approach.

When the data is readable to Weka, the training can start. This is implemented by Weka and the classifiers that are used. This does mean that the progress calls are not integrated. The classification of the different traces is also part of the training and is stored in memory in a lookup table. This lookup table is useful because this allows for a second classifier. The first classifier is used to group similar failing traces. The second classifier can be used to find the root cause in a group.

The implementation of the data mining service results in another micro-service that can help in analysing the faulty traces. This allows for comparing different data mining techniques.

Tab. 8.3.: Trace to table conversion using the third approach: index of first occurrence

!init	?a	!a	?b	!b	!c	!d
1	2	3	-1	-1	-1	4
1	2	3	-1	-1	4	-1
1	-1	-1	2	3	-1	4
1	-1	-1	2	3	4	-1

9 Results and discussion

This section presents and discusses the results of the different experiments that have been executed. The experiments consist of analysing different data sets with different settings on the analysis service. The data sets and relevant settings for the different experiments are described first. Then the metrics are described. The last parts of this chapter show and discuss the results, including formulating an answer to the research questions.

9.1 Data sets

Four different data sets are used. These data sets are generated based on 2 settings: the test set and the test generation strategy. Two test sets are available. The first one is called *comprehensive* and the second one is called *endurance*. The first test set uses a model that contains both good and bad weather. This means that both the most common paths through the SUT as well as some edge cases are tested. The second test set, endurance, uses a model that focuses mainly on bad weather. The first test set can be seen as common usage of the SUT. The second test set was included because the first test set did not find most of the introduced bugs. Another reason for using the second test set as well is because it represents the usage of the SUT by users trying to break the system.

Two test generation strategies have been used. The first is called *WeightedLabelRandom* and the second called *FringeCoverage*. The WeightedLabelRandom strategy is an approach that should resemble the users interaction with system. To do this, at each state, the next input is selected randomly from a set of possible inputs. The inputs are however weighted to be able to make more occurring inputs more likely to be chosen than inputs that are used scarcely. The fringe coverage strategy tries to cover as many transitions as possible. Fringe coverage also looks at possible uncovered transitions that might get covered if a certain input is given in the current state.

The combination of these settings results in four data sets. The data sets that use the WeightedLabelRandom test generation strategy have been created by executing the test over all available mutants of the SUT (as described in section 6.1). The other two data sets that use the FringeCoverage test generation strategy were generated by executing the tests over a randomly generated subset of 100 mutants.

9.2 Settings

In this section the settings for both the SFL analysis service and the data mining service are presented.

9.2.1 SFL settings

The analysis services also have settings that can be used to make small changes to the implementation. The first setting is how the data should be converted for the coverage matrix. There are two options available: transitions and steps. When the transition based conversion is selected, the SFL analysis service will use the transitions in a test to determine what the faulty transitions are in the test trace that is under analysis. The transition is serialised as the name of the start state, the transition label, and the name of the end state. When the step based conversion is selected, the SFL analysis service will use the actions (input to the SUT and output from the SUT), which are steps in the test trace, to determine the faulty input or output action of the test trace that is under analysis.

Besides the conversion method based on transitions or steps, the conversion can also include transition data. If the data is included in the conversion two transitions/steps with the same label, but different data should be considered as different components in SFL. If the data was not included, these two steps or transitions are considered the same component.

The similarity threshold is another setting for the SFL analysis service. This is the threshold that determines if a transition or step is faulty or not. This is very helpful in tuning the SFL analysis service to reduce the number of false positives. This setting is set to 0.85 in the experiments run for the current research. In other research this setting could be varied to find the optimal threshold value.

Finally, the number of pairs for ordered data determines the number of consecutive transitions or steps to be considered a single component to SFL. This setting allows for introducing order in SFL, as described in section 8.4.2. For the experiments ran in the current research, the number of pairs is set to 1, which means that two consecutive transitions or steps are combined into a single component for SFL.

9.2.2 Data mining settings

The settings for the data mining analysis service are the clustering algorithm and the data conversion method. The cluster algorithm that should be used is the first

setting of the data mining service. The value should be the full class name for Weka to use. Example values are:

- weka.clusterers.XMeans
- weka.clusterers.EM
- And potentially many more

The second setting is the data conversion method (see also section 8.5). There are three different options available at this point:

- Presence
- Count
- Index

There are more settings available, like a separate root cause analysis classifier and the corresponding training file, but these are out of scope for the current research and were only explored for recommendation in future work.

Combining all settings would result in over 40 experiments. In a short comparison between the different conversion methods the added value of doing more experiments with different data conversion methods is evaluated. The difference between the conversion methods proved small. Because of the small difference and the scope limitations, only a single data conversion approach is used in the experiments. The other conversion methods, and possibly different methods are left as future work. This reduces the number of experiments to 24.

9.3 Evaluation

The different experiments are evaluated over several metrics. This section describes these metrics and how the resulting numbers have been calculated.

Number of bugs found Since every mutant contains a single bug, every test run should find a single bug. The number of bugs found is the number of times that the analysed result contains the determined faulty step or transition.

Number of bugs not found This is the opposite of the number of bugs found.

Percentage bugs found The number of bugs found plus the number of bugs not found is the total number of test runs that were executed with faulty results. The number of bugs found divided by this total number of test runs is the percentage of bugs found.

Average duration [1] This is the 10% trimmed mean of the calculated duration of each test run analysis. This means that the highest and lowest 10% of the data are not included in the mean.

Total true positives The number of transitions that were determined to be faulty according to the analysis service, that was also manually determined to be faulty.

Total true negatives The number of transitions that were not determined as faulty by the analysis service, nor by manual analysis.

Total false positives The number of transitions that were determined as faulty by the analysis service, but were not manually flagged as being faulty.

Total false negatives The number of transition that were not determined to be faulty according to the analysis service, but were manually flagged as being faulty.

Precision Precision is the ratio of correctly predicted positive observations to the total predicted positive observations. High precision relates to the low false positive rate. $Precision = TP/(TP + FP)$

Recall Recall is the ratio of correctly predicted positive observations to the all observations. $Recall = TP/(TP + FN)$

F-measure/F1 score The F-measure is the weighted average of Precision and Recall. This score takes both false positives and false negatives into account. The F-measure is often more useful than the accuracy. $Fmeasure = 2 * (Recall * Precision)/(Recall + Precision)$

Accuracy Accuracy is the most intuitive performance measure and it is simply a ratio of correctly predicted observation to the total observations. $Accuracy = (TP + TN)/(TP + FP + FN + TN)$

[1] Note that the duration is only an indication: different experiments were run on different machines, the duration is not a reliable measure for direct comparison between two experiments.

Number of correctly grouped tests This is the number of tests that were grouped in the correct group. This is calculated by determining how many groups were made that only contained actual faulty transitions. The number of elements in all these groups are summed to get the value of this metric.

Number of correct groups This is how many groups that have been made, are actually correct. This metric uses the same logic as the *Number of correctly grouped tests*, but instead of summing the elements, the number of groups are counted.

Average correct per group This metric is simply an average of the the number of correctly grouped tests. This metric shows how much work is saved by using the analysis service over manually determining the faulty transition.

9.4 Baseline results

This section discusses the baseline results for the different data sets. For every test, the baseline analysis service returns the last step in the test trace as the faulty step. The reasoning behind this approach is explained in section 8.3.

Table 9.1 shows the results of the baseline analysis on the four different data sets. Note that due to compute capacity, only the WeightedLabelRandom test generation strategy has been run on all mutants, the FringeCoverage strategy has been run on a randomly generated subset of 100 mutants out of 3803. This subset was stored and used for every experiment using FringeCoverage as a test generation strategy. A large part of the mutations were not found by the test runs since the mutations did not affect the covered paths.

Remarkable in table 9.1 is that FringeCoverage does not find any bugs. In general the baseline finds 10% of the bugs with an F-measure of about 8%. This is not a really high score. The average correct per group shows that the groups that were made, were rather good. An average of about 10 tests per group is a reduction of 10 times the work of those test runs. The number of false negatives is also rather high, and as the actual faulty step is hidden in the other discarded steps, this increases the work required to analyse a test run.

Tab. 9.1.: Analysis result using the last step analysis approach on the four different data sets

	Comprehensive		Endurance	
Metric	**Weighted**	**Fringe**	**Weighted**	**Fringe**
Number of bugs found	14	0	7	0
Number of bugs not found	104	6	59	4
Percentage bugs found	11.86%	0.00%	10.61%	0.00%
Average duration (sec)	5.474	23.438	85.553	22.638
Total true positives	14	0	10	0
Total true negatives	7186	0	55952	0
Total false positives	133	14	213	5
Total false negatives	138	14	94	5
Precision	9.52%	0.00%	4.48%	0.00%
Recall	9.21%	0.00%	9.62%	0.00%
F-measure	9.36%	n.a.	6.12%	n.a.
Accuracy	96.37%	0.00%	99.45%	0.00%
Number correctly grouped tests	94	0	31	0
Number of correct groups	10	0	3	0
Average correct per group	9.4	n.a.	10.33333	n.a.

9.5 Data mining analysis results

This section discusses the data mining analysis results. These results are different from the rest of the results since the data mining analysis service only returns groups of test cases that are likely to contain the same failure. Due to the approach taken to test this service, each test run should return a single group since each mutant contains a single introduced fault. If a test run returns multiple groups, it is impossible to determine which group is the correct group, and therefore the group containing the most test traces is assumed to be the correct group. This approach also means a direct comparison between the SFL analysis service and the data mining analysis is not meaningful.

Table 9.2 contains the data mining results for the comprehensive model of the SUT. The results for both test generation strategies (WeightedLabelRandom and FringeCoverage) are contained in this table. The results for the endurance test set is presented in table 9.3. A selection of these metrics are also combined in figure 9.1.

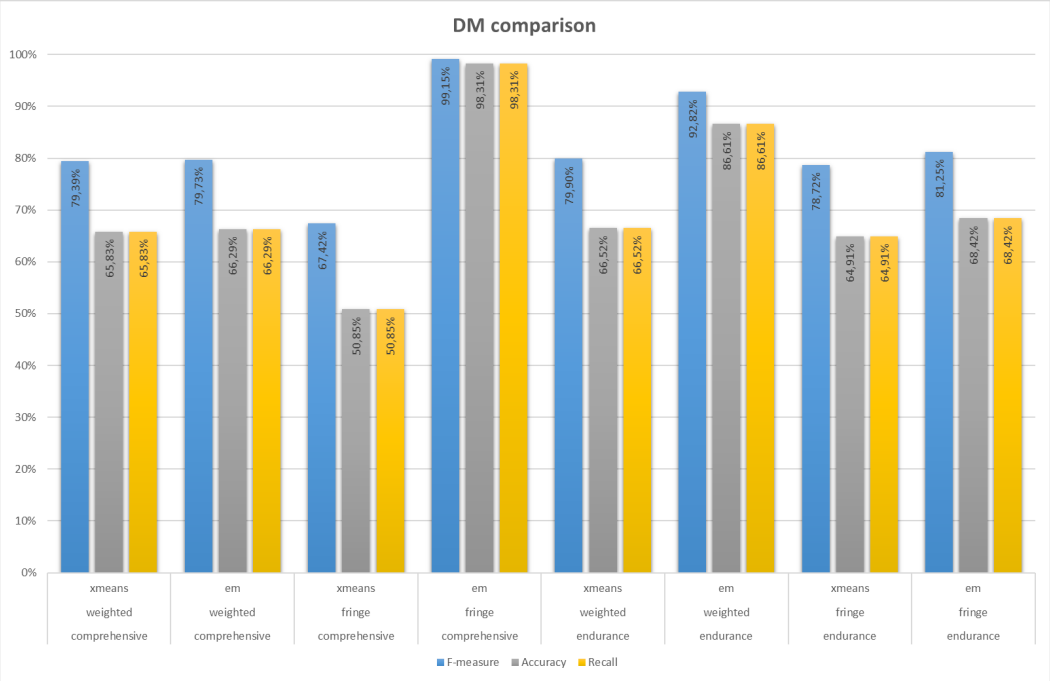

Fig. 9.1.: DM comparison for F-measure

Tab. 9.2.: Data mining results for the **comprehensive** test set using the **WeightedLabelRandom** test generation strategy and the **FringeCoverage** test generation strategy.

	WeightedLabelRandom		Fringe	
Metric	**XMeans**	**EM**	**XMeans**	**EM**
Average duration	4,87	1652,58	26,74	26,82
Total true positives	996	1003	30	58
Total true negatives	0	0	0	0
Total false positives	0	0	0	0
Total false negatives	517	510	29	1
Precision	100,00%	100,00%	100,00%	100,00%
Recall	65,83%	66,29%	50,85%	98,31%
F-measure	79,39%	79,73%	67,42%	99,15%
Accuracy	65,83%	66,29%	50,85%	98,31%
Number correctly grouped tests	996	1003	30	58
Number of correct groups	118	118	6	6
Average correct per group	8,44	8,50	5,00	9,67

Tab. 9.3.: Data mining results for the **endurance** test set using the **WeightedLabelRandom** test generation strategy and the **FringeCoverage** test generation strategy.

	WeightedLabelRandom		Fringe	
Metric	**XMeans**	**EM**	**XMeans**	**EM**
Average duration	113,52	203,88	27,70	30,52
Total true positives	457	595	37	39
Total true negatives	0	0	0	0
Total false positives	0	0	0	0
Total false negatives	230	92	20	18
Precision	100,00%	100,00%	100,00%	100,00%
Recall	66,52%	86,61%	64,91%	68,42%
F-measure	79,90%	92,82%	78,72%	81,25%
Accuracy	66,52%	86,61%	64,91%	68,42%
Number correctly grouped tests	457	595	37	39
Number of correct groups	171	171	4	4
Average correct per group	2,67	3,48	9,25	9,75

Table 9.2 and table 9.3 show several noticeable things. First the average duration of the EM technique for the WeightedLabelRandom test generation approach is extremely high. This might be caused by the machine the analysis was run on. Running this experiment, and the other experiments, more often could show if the machine was extremely slow this once. This means that the duration is not really a reliable metric and is only used as an indication. What is clear from all data mining experiments is that EM requires more time than XMeans in all direct comparisons. Besides the fact that EM requires more time, it is also clear that more data to analyse means more time required for each of the data points, or test runs. This comparison can be done because 8 experiments were run, and for all of them, EM required more time.

A second observation is that the precision is 100% for each technique. This is due to the fact that the true negatives and false positives cannot be calculated. Which is due to the fact that the data mining techniques only group the different tests, and do not identify a faulty transition. This results in 100% precision. This does affect the F-measure as well, as can be seen in the description of the F-measure in section 9.3. This means that the F-measure cannot be compared directly to the baseline results, or the SFL results. The F-measure can be used to compare the different data mining results.

Besides taking more time, EM also results in a better F-measure score. For some experiments, the difference in F-measure is bigger than for others. But for all

experiments, EM results in a better F-measure. Whether this is worth the trade-off in duration cannot be determined from this data. When the service would be implemented in a workflow of a company for some time, the importance of different factors might become clear.

In the end, looking at the average correct per group, using the data mining approach could result in a reduction of work of about 2.5 to 10 times. Instead of having to inspect all test runs and all transitions, only the groups can be inspected, which means that about 8 test traces are inspected at once since on average there are 8 test traces in a group.

9.6 Spectrum-based Fault Localisation analysis results

This section discusses the results for the experiments using SFL analysis services. The results are presented in four tables one table for each of the described data sets. Table 9.4 contains the results for the experiments using the comprehensive test set with the WeightedLabelRandom test generation strategy. The same test set with a different test generation strategy is shown in table 9.5. This table shows the FringeCoverage test generation strategy. For the endurance test set, the results for the same two test generation strategies are shown in table 9.6 and table 9.7 respectively. A selection of these metrics are also combined in figures 9.2 and 9.3.

Tab. 9.4.: SFL results using steps and transitions with and without data for the **comprehensive** test set using the **WeightedLabelRandom** test generation strategy.

	Steps		Transitions		
Metric	**Without data**	**With data**	**Without data**	**With data**	**Max similarity**
Number of bugs found	75	78	0	0	16
Number of bugs not found	43	40	118	118	102
Percentage bugs found	63.56%	66.10%	0.00%	0.00%	13.56%
Average duration (sec)	11.48	13.11	246.335	389.596	203.748
Total true positives	98	102	0	0	16
Total true negatives	9351	9485	0	0	6491
Total false positives	1117	983	150	118	828
Total false negatives	54	50	148	118	136
Precision	8.07%	9.40%	0.00%	0.00%	1.90%
Recall	64.45%	67.11%	0.00%	0.00%	10.53%
F-measure	14.34%	16.49%	n.a.	n.a.	3.21%
Accuracy	88.97%	90.27%	0.00%	0.00%	87.10%
Number correctly grouped tests	1252	1256	0	0	26
Number of correct groups	439	137	0	0	10
Average correct per group	2.852	9.168	n.a.	n.a.	2.6

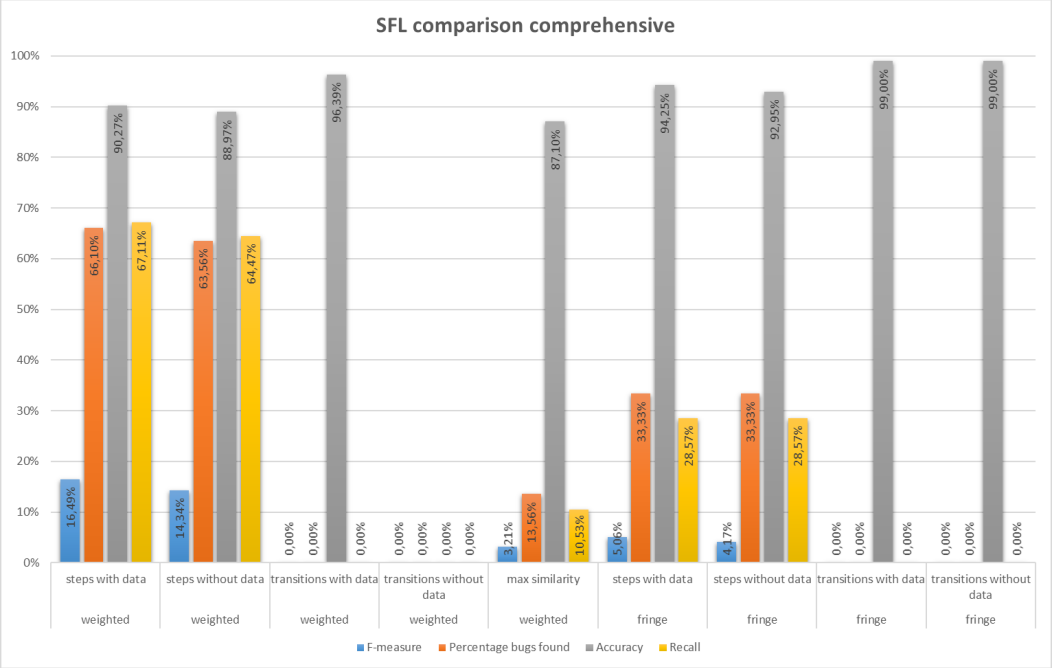

Fig. 9.2.: SFL comparison for F-measure and percentage of bugs found for the comprehensive test set

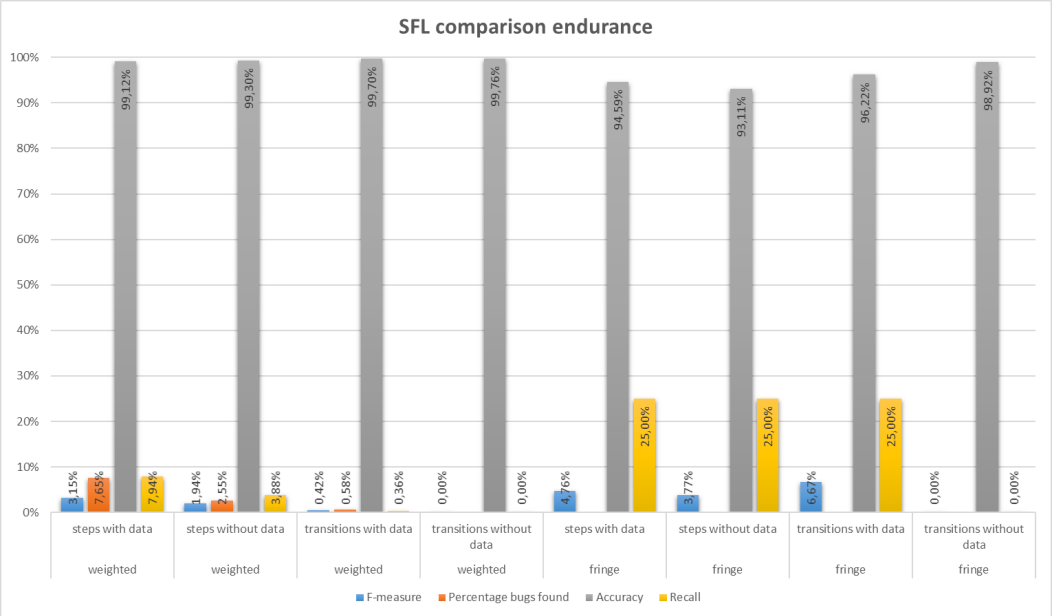

Fig. 9.3.: SFL comparison for F-measure and percentage of bugs found for the endurance test set

Tab. 9.5.: SFL results using steps and transitions with and without data for the **comprehensive** test set using the **FringeCoverage** test generation strategy.

	Steps		Transitions	
Metric	**With data**	**Without data**	**With data**	**Without data**
Number of bugs found	2	2	0	0
Number of bugs not found	4	4	6	6
Percentage bugs found	33.33%	33.33%	0.00%	0.00%
Average duration (sec)	91.877	30.887	881.119	1009.439
Total true positives	2	2	0	0
Total true negatives	1228	1211	1292	1292
Total false positives	70	87	6	6
Total false negatives	5	5	7	7
Precision	2.78%	2.25%	0.00%	0.00%
Recall	28.57%	28.57%	0.00%	0.00%
F-measure	5.06%	4.17%	n.a	n.a.
Accuracy	94.25%	92.95%	99.00%	99.00%
Number correctly grouped tests	0	0	0	0
Number of correct groups	0	0	0	0
Average correct per group	n.a.	n.a.	n.a.	n.a.

Tab. 9.6.: SFL results using steps and transitions with and without data for the **endurance** test set using the **WeightedLabelRandom** test generation strategy.

	Steps		Transitions	
Metric	**With data**	**Without data**	**With data**	**Without data**
Number of bugs found	13	4	1	0
Number of bugs not found	157	153	170	171
Percentage bugs found	7.65%	2.55%	0.58%	0.00%
Average duration (sec)	21.602	44.525	19.187	47.844
Total true positives	22	10	1	0
Total true negatives	151991	142542	155261	155361
Total false positives	1097	761	190	191
Total false negatives	255	248	280	180
Precision	1.97%	1.30%	0.52%	0.00%
Recall	7.94%	3.88%	0.36%	0.00%
F-measure	3.15%	1.94%	0.42%	n.a.
Accuracy	99.12%	99.30%	99.70%	99.76%
Number correctly grouped tests	17	19	13	381
Number of correct groups	9	14	6	160
Average correct per group	1.889	1.357	2.167	2.381

Tab. 9.7.: SFL results using steps and transitions with and without data for the **endurance** test set using the **FringeCoverage** test generation strategy.

	Steps		Transitions	
Metric	**With data**	**Without data**	**With data**	**Without data**
Number of bugs found	1	1	1	0
Number of bugs not found	3	3	3	4
Percentage bugs found	25.00%	25.00%	25.00%	0.00%
Average duration (sec)	30.901	27.041	64.786	112.657
Total true positives	1	1	1	0
Total true negatives	699	688	711	732
Total false positives	37	48	25	4
Total false negatives	3	3	3	4
Precision	2.63%	2.04%	3.85%	0.00%
Recall	25.00%	25.00%	25.00%	0.00%
F-measure	4.76%	3.77%	6.67%	n.a.
Accuracy	94.59%	93.11%	96.22%	98.92%
Number correctly grouped tests	1	1	15	0
Number of correct groups	1	1	1	0
Average correct per group	1	1	15	n.a.

Tables 9.4, 9.5, 9.6, and 9.7 show the results for the experiments that were run using the SFL analysis service in different configurations. Comparing these results brings up several observations. First of all is the large number of experiments that did not result in any bug found. 6 out of the 16 experiments did not find any bugs. All of these experiments were using transitions as the components in SFL. The only experiments that used transitions and did find a bug were from the endurance test set and were using data. But these two experiments have rather poor F-measures.

Secondly, out of the experiments that did return bugs, 90% of the steps and transitions that were classified, were classified correctly. This means that only 10% of the work has to be executed manually. This is a really good result on the accuracy.

The fact that the usage of transitions as the components in SFL performs this poorly, was not expected. The expectation was that model knowledge, as captured by these transitions including the start and end state, would improve the F-measure by improving the precision. Especially since the transitions contain more information, as well as order, than only the steps. On the other hand the steps might differ more, which results in a larger set of components. Inspection of the raw analysis results show that the experiments using the transitions nearly always return "*No problematic steps found*". This message from the SFL analysis service means that

none of the transitions had a similarity higher than the defined similarity threshold. Investigating this further shows that the similarity coefficients for a test run where the bug was not found, the step-based analysis resulted in a similarity ranging from 0 to 0.577. For transition-based analysis of the same test run, the similarity ranged from 0 to 0.5, including results that were not a number (for example because of a division by zero). See table 9.8 for a full overview. This shows that the similarity threshold was too high to identify any problematic steps.

To validate if the wrong similarity threshold was used, a separate experiment was executed called *max similarity*(see table 9.4). This experiment returned the first transition found with the highest similarity, regardless of the similarity threshold value. The results of the experiment show that the bugs could be identified but the threshold was not exceeded. The F-measure of the max similarity experiment is still relatively low though, but this could very well be because it only returned the first with the highest similarity coefficient, instead of all with the highest similarity coefficient. This experiment shows that the transition based experiments did not have the correct similarity threshold value set.

Another reason that might explain the difference between step-based and transition-based analysis is the coverage in the SFL matrix. For the step-based analysis most steps are not marked as covered in the matrix. On average only 5% of the tests covered a certain step, while 75% of the tests covered a certain transition. 20% of the transitions were even covered by all tests. Which is in contrast to the steps, where only 1% of the steps were covered by all tests (See again table 9.8). This big difference in the SFL coverage matrix might explain why the step-based analysis works better than the transition based analysis.

The last reason for the lower accuracy of transition based SFL is the fact that conversion between steps and transitions have to be performed. These conversions are based on assumptions and only based on the label of the steps and the transitions. This makes that only part of the model knowledge is used. A better link between the steps and transitions could improve the transition based analysis. An improved conversion could also remedy the large difference in coverage for the transition and step based coverage matrices.

Continuing the list of observations, the FringeCoverage test generation strategy does not differentiate between the experiments in the results. The outcomes of the experiments are very similar to the other experiments that used the FringeCoverage test generation strategy. This is likely the cause of the small sample set. The comprehensive test set that was generated using FringeCoverage only resulted in 4 test runs that could be analysed since the other 96 did not find the introduced bug, or did not run at all because of raised exceptions in the SUT. The FringeCoverage

Tab. 9.8.: Comparison of different SFL coverage matrices. The test run without a bug found is from the endurance test set with id 15ad67d6-2a36-474f-ac96-3ca7cb0525a7. The test run with a bug found is from the comprehensive test set with id c4d35395-36cd-443d-92f5-1395a5535b1a.

	Steps with data	Steps without data	Transitions with data	Transitions without data
Test run without bug found				
% covered	5	6	74	75
% not covered	95	94	26	25
% fully covered	1	2	22	22
% fully not covered	0	0	15	14
# steps/transitions	5503	2239	3498	3420
Similarity range	0-0.577	0-0.577	0-0.5, NaN	0-0.5, NaN
Test run with bug found				
% covered	6	7	36	37
% not covered	94	93	64	63
% fully covered	1	1	14	15
% fully not covered	0	0	24	23
# steps/transitions	1232	683	1338	1297
Similarity range	0-0.873	0-0.873	0-0.730, NaN	0-0.730, NaN

based experiments are therefore only used for confirmation of observations that were made based on the WeightedLabelRandom based experiments.

The fourth observation that can be made from the four tables is that the usage of data in the steps or transitions improves the F-measure. For each of the direct comparisons, the F-measure is higher if data is used as part of the step or transition. The reason for this could very well be that many transitions themselves do not fail, but the data that was sent does not match. For example, if the prices were calculated incorrectly in the SUT, this was not found if the data was ignored, but was found if the data is included.

Another observation that spikes interest is the number of false positives. On average the number of false positives is 52 times higher than the number of true positives. This means that more than 91% of the identified faulty steps and transitions are false positives. The reason for the high number of false positives is probably due to discriminating power of the algorithm. Especially on a large data set, the different steps occur more often which lowers the spread of the calculated similarity. Because there are many steps and transitions that have a high similarity, there are quite a few false positives. See figure 9.4. This figure shows that a low spread of similarities makes the number of false positives high. By combining the order in the trace (if a probable faulty step is later in the trace, it is more likely to be faulty), the number of false positives could be lowered. Another approach to reducing the number of false positives could be to combine SFL with Delta Debugging. DD could modify the trace by removing the assumed faulty steps or transitions, and check if the test trace still fails. These approaches and more other approaches to lower the number of false positives could be researched in future work.

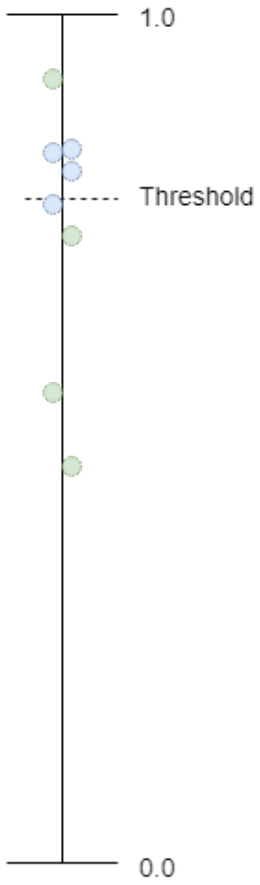

Fig. 9.4.: The different spread of the similarity of four values. The spread of the green values is higher than the spread of the blue values. Only the highest value should have been above the threshold, while the rest should have been below the threshold. This shows a possible cause for the number of false positives.

Comparing the false positives with the true negatives shows a different story. This comparison leads to less than 11% of false positives (an average of 4%) compared to the true negatives. This

shows that the approach does work. Overall the accuracy is also really good, ranging from 88% to 99%. This is another indication that the approach works. At least 88% of the transitions or steps are correctly labelled and do not require further manual analysis.

The goal of the research was to make analysis easier by reducing the size of the data where a user should look. This works for more than half of the cases. For 90% of the returned results, the reduction is correct. This shows that SFL really can help in analysing the root cause of faulty test traces, even though there is room for improvement regarding the number of false negatives. The number of false negatives should be reduced since these are labelled as being not faulty while they are, which makes analysis more difficult rather than easier.

Another way in which the goal of the research is achieved by SFL is in the number of correctly grouped tests. The average for SFL is almost 4. This shows that grouping the tests does work, even though the groups are not really large. The best result is an average group size of 15. This average is created based on a single group. A better average based on more groups is around 9 per group. This group size allows for only analysing a single test that represents the whole group. This also can help in finding reproduction paths for a failure. On this front the goal of the research is achieved.

9.7 Comparing results

This section compares the results of the different experiments with the baseline and with the results of the other experiments. This comparison contains some remarks that might have bias because of a different calculation of the metrics for the data mining analysis service.

This section also answers the research sub-questions posed in section 2.2. The first sub-question was about the SFL analysis service. The second and third sub-question were about the data mining analysis service. The last sub-question was about incorporating model information in SFL. The first and last question are discussed in section 9.7.2 while the second and third question are discussed in the following section, section 9.7.1. Finally the main research question(What method can effectively group similar failing test cases to make diagnostics easier?) is discussed in section 9.7.3.

9.7.1 Data mining ↔ baseline

This section tries to compare the results of the data mining analysis service to the baseline as defined in section 8.3. The focus of this comparison is mostly on the metric *average correct per group*, as defined in the results tables, since this metric is the only one that can truly be compared. This section also answers the following sub-questions from section 2.2:

- Does a data mining clustering algorithm work in grouping (failed) test cases?
- Which data mining techniques can be used to find the root cause of a failed test case?

The baseline has no results for the FringeCoverage test generation strategy. This means that any result is better than the baseline. The data mining analysis service has results for the FringeCoverage test generation strategy. On average this is about 8 tests per group. This is better than none, and rather close to the WeightedLabel-Random test generation strategy for the baseline. The WeightedLabelRandom test generation strategy for the endurance test set is not working as well as the baseline. The reason for this is unclear, since it is somehow hidden in the model of the data mining algorithm. This is one big disadvantage of data mining in general. The difference between the comprehensive test set and the baseline is negligible.

Concluding this comparison, the data mining approach works best on a small data set, which is counter intuitive because data mining is often used on larger data sets. This also answers the first sub-question listed above. The clustering algorithm does work, but there is still room for improvement and tuning. The second sub-question on the other hand was not answered since no approach was identified that can find root causes. This is also the reason that the comparison between the data mining analysis service and the other analysis services are not as meaningful as expected. To be able do some comparison, the groups made by the different analysis services, and their sizes, are used. All in all, grouping the test cases was the goal of this research, and this has been achieved.

9.7.2 SFL ↔ baseline

This section compares the SFL analysis service results with the baseline analysis service results. This section also discusses the first and last sub-questions from the research questions.

The first metric used in the comparison is the F-measure. The F-measure for the step based experiments for the comprehensive test set is 15%. This is a significant difference to the 9% found for the baseline. The FringeCoverage test generation strategy improves the F-measure for every experiment, since the baseline could not find any bugs. For the transition based experiments and the step based experiments for the endurance, there is an average F-measure of less than 1%. This is less than the 9% and 6% in the baseline. This means that SFL works better than the baseline for several data sets, but worse for the others.

The number of false negatives, or the recall, is another interesting metric to look at, since it is important that the correct faulty transition is at least part of the returned possible set of faulty transitions. This shows the same as the F-measure. The comprehensive test set using the WeightedLabelRandom test generation strategy works significantly better than the baseline, but the endurance test set performs worse. For FringeCoverage the same holds. This confirms the conclusion of the previous paragraph: SFL works better than the baseline for several data sets, but for some data sets SFL does not perform as well.

Finally there is the accuracy. On average the accuracy for SFL based analysis is 90%. This is similar to what some of the baseline experiments return, except that the baseline results has a lot higher variance since some experiments have an accuracy of 95% while others have 0%. This means that 90% of the classification is correct, which also means that only 10% requires extra investigation and work.

The two relevant research sub-questions for SFL are:

1. How can algorithms like Delta Debugging and Spectrum-based Fault Localisation be used to find the root cause of a failed test case?

2. How can the information incorporated in the model be used to improve Spectrum-based Fault Localisation in Model Based Testing?

The first question is answered in more detail in the implementation description in section 8.4. The results show that this approach does work, but there is still room for improvement. The second question is answered with an experiment. Using the transitions as the components for SFL is an approach to incorporating model information in SFL. This approach has proven to work (see the max similarity experiment in table 9.4), but this information does not seem to improve the results based on the current data. In future work, different ways of creating the coverage matrix could be explored to improve the results for SFL based on transitions.

9.7.3 Data mining ↔ SFL

This section tries to find the best combination of settings for analysing any new data set, in which it answers the main research question. The different settings are presented at the beginning of this chapter. Finding the best combination of settings is done by comparing the results of the data mining analysis service with the results SFL analysis service.

The first and most important metric considered is the F-measure. The highest SFL F-measure is for an experiment run on the comprehensive test set using the WeightedLabelRandom test generation strategy. This experiment used steps with data as the component conversion method. The threshold used for this experiment was 0.85. The F-measure for this SFL experiment is 16,49%. The highest data mining F-measure is 99% for the comprehensive test set using the FringeCoverage test generation strategy for the EM data mining technique.

Since the two F-measures are not directly comparable due to different calculation of the F-measure, this difference in F-measure does not make the data mining approach better. A better comparison is the average correct per group. The best data mining result has an average correct per group of 9,67. The best SFL result that is reliable is 9,17. This is very close to the data mining result.

Based on these two metrics and the fact that SFL is more powerful, the best analysis service seems to be SFL, using steps with data as the component. SFL is more powerful in the sense that it not only creates groups, but also can identify a single step in the test trace that is faulty.

10 Conclusion

The goal of the current research was defined in section 2.1 as *finding a way to group test cases that fail in a similar way to help in diagnosing problems in the SUT*. To achieve this goal, Atana and the related analysis services were developed. Atana works seamlessly with Axini TestManager and the different analysis services. Furthermore Atana can be managed by running a script as well as by human users. Finally Atana is properly documented to be able to be improved upon by other researchers. To make this possible several existing components have been used, like Rocket, Serde, Hibernate, Spring and Swagger. Some new tools have also been created. These new tools are created using the Rust and Kotlin programming languages and use the respective libraries that work with these languages.

In this thesis the development and validation of Atana and the related analysis services is described. Atana is developed as a central hub in the web of services. The development was lead by a security-by-design-view. This extra focus makes Atana useful in a business context without too much adaptation.

This thesis also shows a new application of the existing fault localisation technique SFL. The white-box based fault localisation technique is applied in a black-box setting, using the information stored in the model of the system to make this application possible. This new approach is validated by several experiments.

The combination of tools works in the following fashion. Axini TestManager is used to run the tests and extract the data required by Atana. Atana receives and stores the model, test runs and logs as JSON documents. Once the analysis of a single test run that the user required was started, the relevant documents are sent to the appropriate analysis service that was configured by Atana. The result of the analysis is returned to Atana and is presented to the user in the desired format. Executing these steps on four different data sets generated by Axini TestManagers results in the data presented and discussed in chapter 9. The discussion of the results shows some positive and some negative points.

First of all, in many cases the accuracy of the analysis was above 85%. This shows that all SFL experiments label the majority of steps and transitions correctly. Another positive observation is the reduction of work. Even if all groups only contain two steps or transitions, this means that only half the work has to be performed. This is due to the fact that only one of the two steps or transitions has to be analysed. The average correct steps or transitions per group of all SFL experiments combined is 4. This means that three quarters of the work can be saved, which lowers both

time consumed and strain on the users. This reduction of work does assume that the groups are created correctly. But even if the groups are not totally correct, there is still a majority of the steps and transitions that is labelled correctly. Once the majority of the problems is found and resolved, the test can be run again, which results in a new analysis based on more information. This is an incremental strategy that requires less effort from the users but still matches many software development projects.

The F-measure and the number of false negatives on the other hand are not as well. The F-measure is influenced by the precision which is low for many experiments. Since the false positives do not matter as much as the false negatives, this does not have to be a show stopper. The number of false negatives should somehow be reduced. Possibly by finding a better SFL similarity threshold or by combining several analysis approaches.

All in all, this shows that the approach to solve the research goal and research questions designed in the current report does work, but there is still room for improvement. Especially the number of false negatives of SFL should be reduced. This could possibly be achieved by combining different fault localisation and analysis techniques which allows for a lower SFL threshold.

11 Future work

The research presented in this report has answered many questions, but raised even more. In future work, these questions could be answered as well. Our work also has resulted in some new insights and contains various limitations, which could be incorporated in future work.

First of all, new analysis approaches could be explored. For example using trace minimisation, or delta debugging. These could be implemented as a new analysis service that could work with Atana to make use of the infrastructure created in our work. To be able to do this, Axini TestManager, or an alternative, should support manually generating test cases to validate if a minimised, or altered, trace still causes a failure.

Another approach that could improve the results would be to analyse tests that resulted in an error as well as using the log files. These two areas have not been explored as part of the research work described here but could yield more information that may prove useful to the analysis.

Furthermore the different values used in SFL, like the similarity threshold and the required number of passed tests to make a test run usable, could be investigated. The numbers used in our work were based on estimations made by reviewing previous test runs. By validating these estimations, the validity of this work could be improved even further.

The validation used in this research is based on mutation testing. This creates mutants with a single automatically introduced fault. To validate the solution even better, special test cases could be crafted from real-world scenarios that have proven to be hard to find using conventional methods. These test cases could contain multiple introduced faults which should all be found.

Finally an improvement could be made by combining different analysis services, for example by combining SFL and delta debugging (DD, see chapter 4). If DD can be used to validate the similarity calculated by SFL, fewer false positives should occur. This could also solve the problem with the false negatives since a lower threshold can be set.

On the other hand, there has been made one big design decision that could benefit from a revisit. To store the models, test runs and logs, a MySQL database was chosen. Since all data is sent between the services as JSON documents, a no-SQL database

might have been a better performing solution for storing the data in Atana. This avoids having to join tables in many cases.

Bibliography

[1]R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. „On the Accuracy of Spectrum-based Fault Localization“. In: *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*. 2007, pp. 89–98 (cit. on pp. 23, 40).

[2]Rui Abreu, Wolfgang Mayer, Markus Stumptner, and Arjan JC van Gemund. „Refining spectrum-based fault localization rankings“. In: *Proceedings of the 2009 ACM symposium on Applied Computing*. ACM. 2009, pp. 409–414 (cit. on pp. 40, 50).

[3]Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. „Spectrum-Based Multiple Fault Localization“. In: *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. ASE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 88–99 (cit. on pp. 50, 52).

[4]C. Adam, A. Aliotti, and P.-H. Cournede. „Learning from User Workflows for the Characterization and Prediction of Software Crashes“. In: 2017, pp. 1023–1030 (cit. on p. 22).

[5]Cyrille Artho. „Iterative delta debugging“. In: *International Journal on Software Tools for Technology Transfer* 13.3 (2011), pp. 223–246 (cit. on p. 21).

[6]C. Chen, H.-G. Gross, and A. Zaidman. „Spectrum-based fault diagnosis for service-oriented software systems“. In: 2012 (cit. on p. 23).

[7]M.X. Cheng and W.B. Wu. „Data Analytics for Fault Localization in Complex Networks“. In: *IEEE Internet of Things Journal* 3.5 (2016), pp. 701–708 (cit. on pp. 17, 26).

[8]H. Cleve and A. Zeller. „Locating causes of program failures“. In: *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* 2005, pp. 342–351 (cit. on p. 21).

[9]Anna Derezińska and Konrad Hałas. „Operators for mutation testing of python programs“. In: *Research Report* (2014) (cit. on p. 28).

[10]A. Dharmarajan and T. Velmurugan. „Lung Cancer Data Analysis by k-means and Farthest First Clustering Algorithms“. In: *Indian Journal of Science and Technology* 8.15 (2015) (cit. on p. 25).

[11]Edsger W. Dijkstra. „Notes on structured programming“. In: *Notes on structured programming*. Techn. Hogeschool, 1970, 7–7 (cit. on p. iii).

[12]Philippe Fournier-Viger, Jerry Chun-Wei Lin, Rage Uday Kiran, Yun Sing Koh, and R Thomas. „A survey of sequential pattern mining“. In: *Data Science and Pattern Recognition* 1.1 (2017), pp. 54–77 (cit. on p. 26).

[13]L. Frantzen, J. Tretmans, and T.A.C. Willemse. „Test generation based on symbolic specifications“. In: vol. 3395. 2005, pp. 1–15 (cit. on p. 16).

[14]Mike Gancarz. *Linux and the Unix philosophy*. Digital Press, 2003 (cit. on p. 39).

[15]B.R. Grishma and C. Anjali. „Software root cause prediction using clustering techniques: A review“. In: 2015, pp. 511–515 (cit. on p. 24).

[16]A. Groce, S. Chaki, D. Kroening, and O. Strichman. „Error explanation with distance metrics“. In: *International Journal on Software Tools for Technology Transfer* 8.3 (2006), pp. 229–247 (cit. on p. 21).

[17]N. Gupta, K. Anand, and A. Sureka. „Pariket: Mining business process logs for root cause analysis of anomalous incidents“. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 8999 (2015), pp. 244–263 (cit. on pp. 24, 25).

[18]Mark Hall, Eibe Frank, Geoffrey Holmes, et al. „The WEKA data mining software: an update“. In: *SIGKDD Explorations* 11.1 (2009), pp. 10–18 (cit. on pp. 4, 25, 28, 39, 53).

[19]M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. „An empirical investigation of the relationship between spectra differences and regression faults“. In: *Software Testing, Verification and Reliability* 10.3 (2000), pp. 171–194 (cit. on p. 21).

[20]A.R. Hevner, S.T. March, J. Park, and S. Ram. „Design science in information systems research“. In: *MIS Quarterly: Management Information Systems* 28.1 (2004), pp. 75–105 (cit. on p. 29).

[21]John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. „Empirical Assessment of MDE in Industry“. In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 471–480 (cit. on pp. iii, 1).

[22]„IEEE Standard Glossary of Software Engineering Terminology“. In: *IEEE Std 610.12-1990* (1990), pp. 1–84 (cit. on p. 17).

[23]J. A. Jones, M. J. Harrold, and J. Stasko. „Visualization of test information to assist fault localization“. In: *In Proc. International Conference on Software Engineering (ICSE)* (2002). Orlando, Florida, pp. 467–477 (cit. on p. 21).

[24]T. Kanstren and M. Chechik. „Trace reduction and pattern analysis to assist debugging in model-based testing“. In: 2014, pp. 238–243 (cit. on p. 22).

[25]H. Lal and G. Pahwa. „Root cause analysis of software bugs using machine learning techniques“. In: 2017, pp. 105–111 (cit. on pp. 25, 28).

[26]S.-M. Lamraoui, S. Nakajima, and H. Hosobe. „Hardened Flow-Sensitive Trace Formula for Fault Localization“. In: vol. 2016-January. 2016, pp. 50–59 (cit. on pp. 17, 20).

[27]Si-Mohamed Lamraoui and Shin Nakajima. „A Formula-Based Approach for Automatic Fault Localization of Imperative Programs“. In: *Formal Methods and Software Engineering*. Ed. by Stephan Merz and Jun Pang. Cham: Springer International Publishing, 2014, pp. 251–266 (cit. on p. 20).

[28]Gary MacGraw. *Software security: building security in*. Addison-Wesley, 2006 (cit. on p. 41).

[29]Nicholas D. Matsakis and Felix S. Klock II. „The Rust Language“. In: *Ada Lett.* 34.3 (Oct. 2014), pp. 103–104 (cit. on p. 42).

[30]M. Nieminen and T. Raty. „Adaptable Design for Root Cause Analysis of a Model-Based Software Testing Process“. In: 2015, pp. 379–384 (cit. on pp. 19, 20).

[31]M. Nieminen, T. Räty, and R. Teittinen. „Integration of root cause analysis into a model-based testing process of a mobile switching server“. In: 2013, pp. 305–309 (cit. on pp. 19, 23).

[32]T. Reidemeister, M. Jiang, and P.A.S. Ward. „Mining unstructured log files for recurrent fault diagnosis“. In: 2011, pp. 377–384 (cit. on p. 26).

[33]M. Renieris and S. P. Reiss. „Fault localization with nearest neighbor queries“. In: *In Proc. 18th Int. Conference on Automated Software Engineering* (2003). Montreal, Canada (cit. on p. 21).

[34]Bran Selic. „What will it take? A view on adoption of model-based methods in practice“. In: *Software & Systems Modeling* 11.4 (2012), pp. 513–526 (cit. on pp. iii, 1, 2).

[35]Dusan Stevanovic, Natalija Vlajic, and Aijun An. „Detection of malicious and non-malicious website visitors using unsupervised neural network learning“. In: *Applied Soft Computing* 13.1 (2013), pp. 698 –708 (cit. on p. 25).

[36]S. Suriadi, C. Ouyang, W.M.P. Van Der Aalst, and A.H.M. Ter Hofstede. „Root cause analysis with enriched process logs“. In: *Lecture Notes in Business Information Processing* 132 LNBIP (2013), pp. 174–186 (cit. on p. 24).

[37]F. Thung, D. Lo, and L. Jiang. „Automatic defect categorization“. In: 2012, pp. 205–214 (cit. on p. 26).

[38]F. Thung, D. Lo, and L. Jiang. „Automatic recovery of root causes from bug-fixing changes“. In: 2013, pp. 92–101 (cit. on p. 26).

[39]Jan Tretmans. „Model Based Testing with Labelled Transition Systems“. In: *Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected Papers*. Ed. by Robert M. Hierons, Jonathan P. Bowen, and Mark Harman. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–38 (cit. on p. 15).

[40]T. Weigert and F. Weil. „Practical experiences in using model-driven engineering to develop trustworthy computing systems“. In: *IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC'06)*. Vol. 1. 2006, 8 pp.– (cit. on pp. iii, 1).

[41]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 (cit. on pp. 4, 11, 23, 49).

[42]W Eric Wong, Vidroha Debroy, Yihao Li, and Ruizhi Gao. „Software fault localization using DStar (D*)“. In: *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*. IEEE. 2012, pp. 21–30 (cit. on p. 50).

[43]A. Zeller. „Automated debugging: are we close?“ In: *Computer* 34.11 (2001), pp. 26–31 (cit. on pp. 4, 20, 21).

[44]A. Zeller. „Yesterday, my program worked. Today, it does not. Why?“ In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 1687 LNCS (1999), pp. 253–267 (cit. on pp. 4, 11).

[45] Andreas Zeller. „Isolating Cause-effect Chains from Computer Programs“. In: *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*. SIGSOFT '02/FSE-10. Charleston, South Carolina, USA: ACM, 2002, pp. 1–10 (cit. on pp. 4, 20).

[46] Mengshi Zhang, Xia Li, Lingming Zhang, and Sarfraz Khurshid. „Boosting Spectrum-based Fault Localization Using PageRank“. In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2017. Santa Barbara, CA, USA: ACM, 2017, pp. 261–272 (cit. on pp. 50, 52).

[47] Hui Zhu, Tu Peng, Ling Xiong, and Daiyuan Peng. „Fault Localization Using Function Call Sequences“. In: *Procedia Computer Science* 107 (2017). Advances in Information and Communication Technology: Proceedings of 7th International Congress of Information and Communication Technology (ICICT2017), pp. 871 –877 (cit. on p. 52).

[48] Xiaojin Zhu and Zoubin Ghahramani. „Learning from Labeled and Unlabeled Data with Label“. In: (2002) (cit. on pp. 25, 28).

Appendices

Test case suggestions

A

Crafting manual test cases could add to the validation of the different analysis services. This list contains several suggestions for future research:

- A fault that can immediately be discovered since the output transitions are not all present due to this fault. These are the relatively simple faults. The root cause of this problem is the last visited state.

- A fault of which the symptom is shown in a following state. This is a common problem. The fault is in one state, that sets some variable. When this variable is later used, the program tries to use an invalid value and will show a problem in the test. The root cause of this problem is that the variable was set wrong, while the test will fail when the variable is used. If this variable is used in many places this would be a perfect example of where the symptom is at a different spot than the problem and this is where root cause analysis really shines.

- A fault of which the symptom is only shown the second time the test goes over that state. This problem is more difficult to find. An example of these types of faults could be that the index in a loop is off by one. This is hard to spot, but is often clear in a test. The root cause is the loop condition while the symptom shows that a transition is not enabled later in the process. Note that this should not be the last state in the trace since that case is already covered with a different test.

- A test should be generated that will pass. This test case is to check that the program will not group passing tests with the failing ones. Basically this is a sanity check.

- A fault that depends on a variable if it happens or not. This should be somewhere so multiple generated tests will go over this code and it will sometimes pass and sometimes fail. In this case it is expected that the state where this variable is set, is classified as the root cause and therefore all tests that fail due to this fault are grouped and the ones that pass will be left.