JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Markus Zimmermann, BSc

Submitted at
Institute for Formal Models and Verification

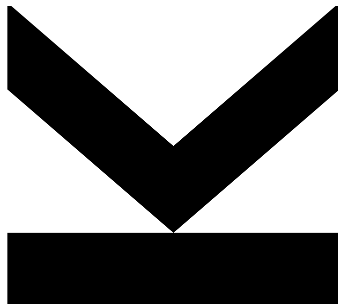
Supervisor
Univ.-Prof. Dr. Armin Biere

Co-Supervisors
Assoc. Univ.-Prof. Dr. Martina Seidl

Dr. Josef Pichler

December 2017

Tavor - A Generic Fuzzing and Delta-Debugging Framework



Master Thesis
to obtain the academic degree of
Diplom-Ingenieur
in the Master's Program
Computer Science

JOHANNES KEPLER UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Abstract

Testing software to verify its correctness and debugging code to locate and patch faults are two important tasks that need to be mastered by every software developer. With increasing complexity of software these tasks become progressively complicated and cumbersome. Hence, approaches that simplify these tasks are needed. Fuzzing and delta-debugging are two zeitgeisty automatic techniques that allow the systematic generation and reduction of test data. However, most implementations of these techniques utilize either fuzzing or delta-debugging with hard-coded models, or are complicated fuzzing frameworks that lack usability.

In this thesis, we introduce TAVOR, a framework and tool for applying both fuzzing and delta-debugging while operating on one user-defined data model, and the TAVOR FORMAT, an EBNF-like notation that allows to define data models for file formats, protocols and test cases. In combination they allow the basic utilization of fuzzing and delta-debugging without any programming knowledge, making these techniques available to non-expert users. Additionally, we present the necessary data structures, interfaces and algorithms to achieve this combination of fuzzing and delta-debugging.

One part of our evaluation is the comparison of TAVOR's fuzzing capabilities with *aigfuzz*, a dedicated fuzzer for the sophisticated AIGER format. In total 16 commands of the AIGER toolset were evaluated to compare the generated test sets. On average the *random* fuzzing strategy of the TAVOR FRAMEWORK reached `9.16%` more line coverage than *aigfuzz*. The best result has been obtained for the *aigunroll* command, where *aigfuzz* covered `24.08%` and TAVOR's *AlmostAllPermutations* fuzzing strategy reached `61.36%`. In summary, this evaluation showed that TAVOR as a generic fuzzer can keep up with a dedicated fuzzing implementation.

Kurzfassung

Das Testen von Software um ihre Korrektheit zu überprüfen und das Debuggen von Source Code zum Finden und Korrigieren von Fehlern sind zwei wichtige Tätigkeiten, die von jedem Softwareentwickler gemeistert werden müssen. Mit ansteigender Komplexität von Software werden diese Tätigkeiten jedoch zunehmend kompliziert und mühsam. Es ist daher nötig Herangehensweisen anzuwenden, welche diese Tätigkeiten vereinfachen. Fuzzing und Delta-Debugging sind zwei dem Zeitgeist entsprechende automatisierte Techniken, für die systematische Generierung und Reduzierung von Testdaten. Die meisten Implementierungen von Fuzzing und Delta-Debugging erlauben jedoch nur die Anwendung einer dieser Techniken anhand eines fest programmierten Datenmodells, oder repräsentieren komplizierte Frameworks zur Anwendung von Fuzzing denen es an Benutzerfreundlichkeit fehlt.

Diese Arbeit stellt Tavor vor, ein Framework und Tool für die gleichzeitige Anwendung von Fuzzing und Delta-Debugging anhand benutzerdefinierter Datenmodelle, und das Tavor Format, einer EBNF-ähnlichen Notation zur Definition von Datenmodellen für Dateiformate, Protokolle und Testfälle. Zusammen erlauben sie die grundlegende Anwendung von Fuzzing und Delta-Debugging ohne das Voraussetzen von Programmierkenntnissen, wodurch diese Techniken auch für Nicht-Experten zugänglich gemacht werden. Zusätzlich präsentiert diese Arbeit alle nötigen Datenstrukturen, Schnittstellen und Algorithmen welche für diese Kombination von Fuzzing und Delta-Debugging nötig sind.

Ein Teil unserer Evaluierung ist der Vergleich von Tavors Fuzzing-Fähigkeiten mit *aigfuzz*, einem dedizierten Fuzzer für das anspruchsvolle AIGER-Format. Insgesamt wurden 16 Befehle vom AIGER-Toolset evaluiert, um die generierten Testsets zu vergleichen. Durchschnittlich erreichte die *random* Fuzzing-Strategie vom Tavor Framework `9.16%` mehr Line-Coverage als *aigfuzz*. Das beste Ergebnis wurde für den Befehl *aigunroll* erzielt, für den *aigfuzz* `24.08%` Abdeckung erzielt und Tavors *AlmostAll-Permutations* Fuzzing-Strategie sogar `61.36%` erreichte. Zusammenfassend lässt sich durch diese Evaluierung sagen, dass Tavor als generischer Fuzzer mit einer dedizierten Fuzzing-Implementierung mithalten kann.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Goal of Thesis . . . 2
1.3 Contributions . . . 3
1.4 Outline . . . 4

2 Background **7**
2.1 What Is Fuzzing? . . . 7
2.1.1 Mutation-Based Fuzzing . . . 8
2.1.2 Generation-Based Fuzzing . . . 9
2.2 What Is Model-Based Testing? . . . 10
2.3 What Is Mutation Testing? . . . 11
2.4 What Is Delta-Debugging? . . . 13

3 The Tavor Framework **17**
3.1 Components . . . 18
3.2 Tokens . . . 19
3.2.1 Example Implementation - The Smiley Token . . . 21
3.2.2 Advanced Token Concepts . . . 25
3.3 Fuzzing Strategies . . . 28
3.3.1 Basic Example Fuzzing Strategy . . . 30
3.3.2 The AllPermutations Fuzzing Strategy . . . 31
3.4 Fuzzing Filters . . . 35
3.5 Reducing Strategies . . . 37
3.5.1 Basic Example Reducing Strategy . . . 39
3.5.2 The Linear Reducing Strategy . . . 40

4 Tavor Format **45**
4.1 Token Definition . . . 45
4.2 Terminal Tokens . . . 47
4.3 Embedding of Tokens . . . 47
4.4 Alternations . . . 48
4.5 Groups . . . 49
4.6 Character Classes . . . 51
4.7 Token Attributes . . . 52

4.8 Typed Tokens . . . 54
4.9 Expressions . . . 56
4.10 Variables . . . 59
4.11 Statements . . . 60

5 Tavor CLI **63**
5.1 Command `graph` . . . 64
5.2 Command `fuzz` . . . 67
5.3 Command `validate` . . . 69
5.4 Command `reduce` . . . 69

6 The go-mutesting Framework **73**
6.1 Motivation . . . 73
6.2 Components . . . 74
6.3 Mutators . . . 75
6.4 Exec Commands . . . 76

7 Evaluation **79**
7.1 Case Study: Coin Vending Machine . . . 79
7.1.1 Definition of the Coin Vending Machine . . . 80
7.1.2 Keyword-Driven Testing . . . 80
7.1.3 Tavor Format and Fuzzing . . . 82
7.1.4 Mutation Testing . . . 86
7.1.5 Delta-Debugging . . . 87
7.2 Fuzzing the AIGER ASCII Format . . . 89
7.2.1 Introducing the AIGER ASCII Format . . . 89
7.2.2 Experimental Setup . . . 92
7.2.3 Results and Conclusions . . . 94
7.3 Fuzzing the JSON Format . . . 98
7.3.1 Introducing the JSON Format . . . 98
7.3.2 Experimental Setup . . . 100
7.3.3 Results and Conclusions . . . 102

8 Conclusion **107**
8.1 Summary . . . 107
8.2 Future Work . . . 108

A Tavor Framework Pseudo Codes **111**

B Tavor CLI Command Line Arguments **113**

C Case Study: Coin Vending Machine **119**

Bibliography **129**

Chapter 1

Introduction

In this chapter we introduce the main topics and motivation, followed by the goals, contributions and the chapter structure of this thesis.

1.1 Motivation

Testing software is a complicated and cumbersome but necessary task to verify the correctness of a program. Nowadays software developers usually lean to exercise testing using automated measures, such as writing and executing unit, integration and system tests, instead of or additionally to manually testing the program under test. Even though such automation has clear advantages for catching regressions of modifications to existing programs, developers often tend to only utilize test data that they expect to be interesting to a program. This leads to the problem that only expected behavior is tested by such automated measures. However, given enough time, users of a program will exercise unexpected behavior too, which can lead to program crashes, false results or vulnerabilities. Such negative outcomes make it necessary to invest more time into software testing, to increase the probability of covering all corner-cases of the program under test. However necessary such thorough testing is, it is no guarantee to find all problems of a program, and it can be therefore seen as one of the more bothersome tasks of developing software. Fortunately, automated techniques exist to at least help with the task of thorough testing. One such technique is fuzz testing, or simply fuzzing, which was introduced by Miller et al. [16]. Fuzzing, in its most basic form, generates unstructured random data as input for the execution of a program. Even tough such data can lead to early success in testing software for unexpected behavior [16], it depends on pure luck given the underlying randomness, and is therefore prone to poor coverage for the program under test. Each condition of a program reduces the effectiveness of unstructured random generated data to reach certain program areas, since the data is more likely to be invalidated with each condition. To overcome this limitation, models

can be additionally utilized to generate more structured data to cover deeper execution paths, which is therefore more likely to uncover problems of the program under test.

However useful fuzzing is to generate interesting data, the results can often be largely sized. Even if data reproducibly exercises a problem, its size becomes a limiting factor for the developer, since more context has to be included for fixing the program. Debugging the given problem can be aided by simply reducing the size of the context. Hence, by trimming data of irrelevant parts so that it still reproduces the same problem. This procedure is called delta-debugging and was introduced by Zeller in [22]. However, an unstructured reduction of the given data, can lead to invalidating a condition of the underlying execution path. Utilizing a model of how expected data is structured, can help to systematically reduce huge data and keep it valid, to at least increase the probability of hitting the same problem as the unreduced data.

This thesis explores the assumption that both fuzzing and delta-debugging can strongly benefit from utilizing the same model which represents expected data for a program under test. The subsequent Section 1.2 substantiates the goals of this thesis for the exploration of this assumption and lists restrictions to these goals.

1.2 Goal of Thesis

Fuzzing and delta-debugging are strong techniques to aid testing and debugging of software programs. Many implementations for these techniques exist but suffer from the following two distinct disadvantages:

1. Most implementations are either frameworks to implement specialized models in the framework's programming language, or programs with the sole purpose of either fuzzing or delta-debugging with hard-coded models. Programming is therefore a seemingly required skill to utilize both techniques. This raises the following questions: What are the common data structures and algorithms that can be shared to utilize fuzzing and delta-debugging? What are the underlying concepts that are necessary to define arbitrary models? How can such definitions be represented to make fuzzing and delta-debugging of these models available to non-programmers?

2. Additionally, implementations for fuzzing and delta-debugging commonly utilize only one of the two techniques inside a single application. This raises the following question: How can a model be defined without any additional tweaks, so that both techniques can operate on it within the same application?

These aforementioned questions are the foundation of this thesis and can be formulated as the following main goals:

1. Define the main concepts, data structures, algorithms and interfaces that are necessary to utilize a common model for applying fuzzing and delta-debugging.

2. Additionally, conceive a declarative language for defining such models that is usable without the knowledge of a programming language.

In order to keep within reasonable limits, this thesis focuses, additional to the aforementioned main goals, on the following subgoals:

- Implement generative fuzzing for the conceived models, which allows the generation of valid data that can be used for positive testing.

- Make it possible to define at least one sophisticated data model to directly compare this thesis to other fuzzing implementations. The sophisticated format chosen for this purpose is the AIGER ASCII format (AAG) defined at [1]. The format is capable of defining and-inverter graphs that allow the definition of combinational circuits.

- Implement the necessary interfaces to allow fuzzing and delta-debugging of external programs.

- Allow and showcase delta-debugging for simple data models but not necessarily for sophisticated models.

1.3 Contributions

The following list is an excerpt of the contributions that were made in consequence of this thesis:

- One of the main contributions is the implementation of TAVOR, a now established framework and tool, which utilizes a common model for fuzzing and delta-debugging. The implementation has been open sourced using the permissive MIT license [2].

[1] http://fmv.jku.at/aiger/FORMAT
[2] https://github.com/zimmski/tavor

- The conception and implementation of the TAVOR FORMAT for the definition of data, such as file formats and protocols, has also been open sourced along with TAVOR [3].

- Additional to TAVOR, other open source contributions have been made. The project GO-MUTESTING [4], a mutation testing framework and tool for GO source code, has been established and is to this date the most widely used mutation testing tool in its area. Large contributions have been made to the extensively used GO package go-flags [5], a command line option and configuration parser, which lead to a maintainership of this package. Many contributions to the widely adopted GO source code static-analysis projects errcheck [6] and golint [7] have been made. The project go-leak [8], a GO package to identify resource leaks, has been established as no implementation existed during the implementation of TAVOR.

- Patches to the GO project [9] have been made of which one of them fixed a silent corruption of the internal structure of the container/list package. Additionally, the identification of inconsistencies in the encoding/json package lead to patches to the GO package.

- Contributing TAVOR to the open source community lead to a mentoring position for the lowRISC organization [10] in the Google Summer of Code 2015. As a result one student used TAVOR for generating assembly test cases for the RISC-V architecture.

1.4 Outline

The remainder of this thesis is structured as follows:

- In Chapter 2 we explain the terminology as well as the main topics and techniques of this thesis.

- In Chapter 3 we present the TAVOR FRAMEWORK by describing its main design goals, components, data structures and algorithms.

[3] https://github.com/zimmski/tavor/blob/master/doc/format.md
[4] https://github.com/zimmski/go-mutesting
[5] https://github.com/jessevdk/go-flags
[6] https://github.com/kisielk/errcheck
[7] https://github.com/golang/lint
[8] https://github.com/zimmski/go-leak
[9] https://golang.org/
[10] http://www.lowrisc.org/

- In Chapter 4 we introduce the TAVOR FORMAT: an EBNF-like notation which allows the definition of data, such as file formats and protocols, without the need of programming.

- In Chapter 5 we present the TAVOR CLI: the user interface for non-programmers to utilize capabilities such as fuzzing and delta-debugging of the TAVOR FRAMEWORK.

- In Chapter 6 we introduce GO-MUTESTING: a framework for performing mutation testing on source code of the programming language GO.

- In Chapter 7 we provide an evaluation of the implemented solutions under different scenarios.

- In Chapter 8 we discuss the conclusions we could draw from this thesis and give an outlook for possible future extensions.

Chapter 2

Background

This chapter provides an overview of the main topics discussed in this thesis: namely, *fuzzing* in Section 2.1, *model-based testing* in Section 2.2, *mutation testing* in Section 2.3 and *delta-debugging* in Section 2.4. Each section consists of a common definition of the corresponding topic, its terminology, techniques, typical workflow and its components. Furthermore, each section presents how the respective main topic is employed in this thesis and how these topics are related to each other.

2.1 What Is Fuzzing?

Fuzz testing, or simply *fuzzing*, is a software testing technique which—in its original form introduced by Miller et al. [16]—uses random generated data as input for the execution of software programs. The system under test, which can range from a single program to complex software infrastructure, is then monitored for unexpected behavior such as program crashes, and program defects, buffer overflows and memory leaks. Nowadays fuzzing is a popular choice for uncovering software vulnerabilities by security testing [15, 19, 20], since it does not require to understand the source code or individual components of the system under test. Furthermore, fuzzing does not even require to have access to the source code but solely relies on the execution of the system. Hence, every generated input that exercises a problem during fuzzing, is a true positive for a deterministic system under test and therefore adds value to the testing process. This makes fuzzing, a superior choice in contrast to other techniques, at least if this characteristic is the most interesting for choosing a technique. For example static analysis, which looks at source code without executing it, relies on heuristics which can sometimes produce false positives. Hence, it can be necessary to check and verify every case manually, making such a technique far more inefficient.

The fundamental components of a fuzzer, the program for applying fuzzing, as described in [15], are the `Fuzz Generator`, the `Delivery Mechanism` and the `Monitoring Sys-`

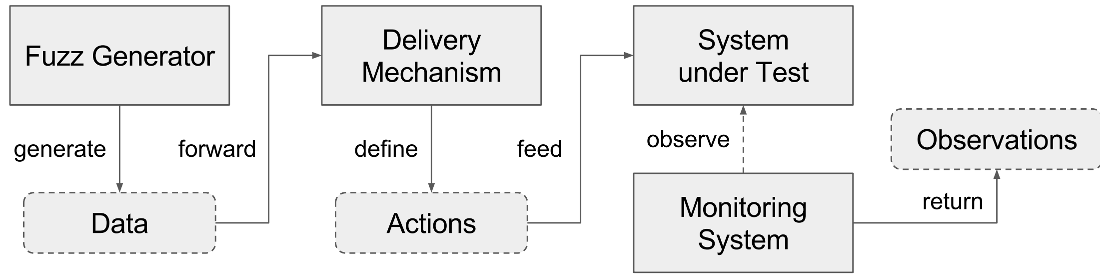


Figure 2.1: Fundamental Components of Fuzzing

`tem`. These components, as well as their typical interactions, are depicted in Figure 2.1. The fuzz generator creates data which is then fed by the delivery mechanism to the system under test. Each system can have its own mechanism of accepting input data, e.g., a graphical user interface needs different actions than a program using a command line interface, and it is therefore the responsibility of the delivery mechanism to apply the correct actions for the given data and system. Observing the execution of these actions is the responsibility of the monitoring system that defines which problems can be found by the fuzzing process. The monitoring system can be as simple as checking the exit status of a program, i.e., to detect program crashes, or more sophisticated such as the instrumentation for buffer overflows during the execution of a program.

Even though Miller et al. showed in [16] that the generation of random bytes as inputs for programs can harvest good results, its effectiveness gets limited by every conditional branch of the program under test. Each check and verification of a program requires the input data to be more structured for reaching deeper program areas. To overcome this issue two types of approaches are used by more advanced fuzz generators [20]: *mutation*-based and *generation*-based techniques.

2.1.1 Mutation-Based Fuzzing

Mutation-based fuzzers, or mutative fuzzers, take existing data and simply change it according to different rules. Such rules can be as simple as randomly toggling bits or as complex as combining two different sets of data into a new set. The advantage of mutation-based fuzzers is that they can work without the knowledge of how data must be structured and can therefore be implemented independently of the system under test. However, since they only adapt existing data, it is highly unlikely that program areas that are not bound to the existing data are exercised with the mutated data. Additionally, the advantage of using existing structured data is thwarted since the applied changes make the resulting data again more likely to be invalidated by existing checks and verifications of the program.

2.1.2 Generation-Based Fuzzing

Generation-based fuzzers, or generative fuzzers, create new data without the need for existing data. They overcome the issues of mutation-based fuzzers by possessing an almost or even complete knowledge of the structure for the data that is valid for the system under test. By using this knowledge, which is called a model, data can be generated that exercises greater program coverage, since it complies to the expected structure and semantics of the system under test [17].

Given that generative fuzzers have the knowledge of how valid data for a system under test has to be generated, they can be used as an efficient technique for positive testing. Hence, testing the system under test for expected behavior which includes valid as well as invalid data to test for expected error handling that is included in the specification. However, the knowledge of how valid data must be generated can also be used to derive invalid data by systematically adapting the underlying model to effectively apply negative testing. Hence, testing invalid cases which are not covered by the specification in order to provoke unexpected behavior such as system crashes.

Since one of the main goals of this thesis is to combine fuzzing and delta-debugging using the same underlying model, generative fuzzing seems to be the only logical choice for choosing an approach to implement fuzzing. However, generative fuzzing also has strong advantages over mutation-based fuzzing, such as not requiring to accumulate seemingly interesting test data, which can be a time-consuming task. Furthermore, generative fuzzing allows to accurately generate test data for specific scenarios while mutation-based fuzzing either requires to find such cases by chance or manually. An implementation of the generative fuzzing approach of this thesis can be found in Section 3.3. However, techniques rooted in mutation-based fuzzing can be applied to data models, thereby allowing effective negative testing in combination with mutation-based fuzzing. This exact combination has been implemented for this thesis and is described in Section 3.4.

Even though fuzzing allows to effectively generate test data, it has the disadvantage of skipping one important part of testing software: verifying that specified requirements are implemented. In the subsequent section the software testing technique *model-based testing* is introduced, which is able to fill this gap by requiring the validation of the system's output after feeding the system generated test data.

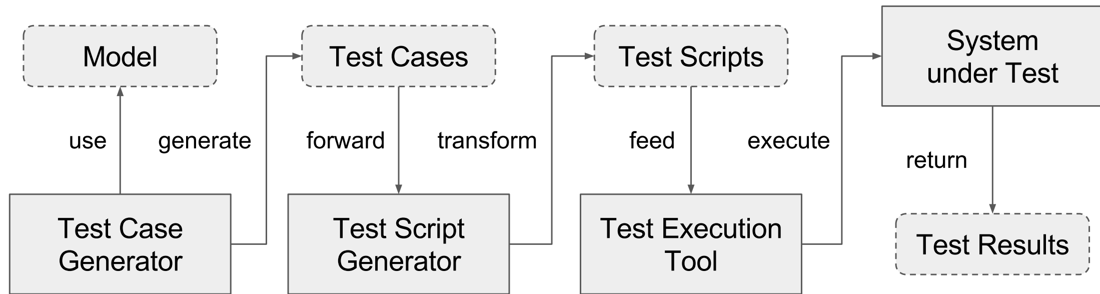


Figure 2.2: Fundamental Components of Model-Based Testing

2.2 What Is Model-Based Testing?

Model-based testing is a software testing technique which derives tests from a model that is based on the requirements on the system under tests. It is therefore a form of black-box testing. Furthermore, Utting et al. define in [21] model-based testing to be "the automation of the design of black-box tests", i.e., model-based testing does not only produce the input data for the execution of the system under test, but it must also generate executable test cases that include checks for the output of the system under test. An extensive list of case studies and references has been gathered by Utting et al. in [21], which showcase various benefits of model-based testing. Among them are reduced testing cost and time, as well as improved test quality, i.e., higher coverage of requirements testing.

The fundamental components of a model-based tester, according to [21], are the `Test Case Generator`, the `Test Script Generator` and the `Test Execution Tool`. These components, as well as their typical interactions, are depicted in Figure 2.2. The test case generator uses the model to generate test cases for the system under test. These test cases may directly be executed. However, their form should be abstract in order to be able to reuse them in different environments. The test script generator then takes these test cases and transform them into test scripts which are either directly executable on the system under test, which is called online testing, or applicable using the test execution tool, which is called offline testing. The application of these test scripts outputs the final test results, i.e., whether a test case has passed the execution on the system under test.

Model-based testing and generation-based fuzzing, as introduced in Section 2.1, utilize a data model to be able to derive and execute tests. Hence, it is only fitting to compare both techniques:

- The fuzz generator of the fuzzer has the same behaviors and responsibilities as the model and test case generator of a model-based tester.

- The delivery mechanism of the fuzzer has the same behaviors and responsibilities as the test script generator and the test execution tool of the model-based tester.

- The online testing variant of model-based testing can also be directly compared to feedback-driven fuzzing. Both techniques executed their cases directly on the system and require some kind of monitoring of the system under test. The feedback of the monitoring can then be incorporated into the test cases, e.g., to check if an error message occurred, and guide the generation of the next test case.

Even though we can directly compare generation-based fuzzing to model-based testing, we can also observe three major differences: Model-based testing is much stricter concerning the model which is used to generate test cases. It has to be based on the requirements of the system under test. Furthermore, the validation of the system's output, which is a requirement for model-based testing, is not necessary for a fuzzer, but can be adopted by the monitoring system. Lastly, model-based testing strives to generate reusable test cases which are semi-executable, while fuzzing is often described of only generating inputs which can be used as parameters for test cases or the execution of the system under test.

These differences make model-based testing an attractive addition to fuzzing to completely cover the whole spectrum of software testing. As a result, this thesis introduces with the Tavor framework in Chapter 3 and the Tavor format in Chapter 4 capabilities to define and structure functional test cases which include the validation of requirements. Furthermore, Chapter 5 introduces with the Tavor CLI functionality to communicate with the systems under test. These additions allow the utilization of model-based testing which are showcased in the evaluation presented in Chapter 7.

Since both, fuzzing and model-based testing, allow to generate test cases to form test suites, a suitable method for evaluating these test suites must be employed. The subsequent section introduces *mutation testing*, which allows to measure the effectiveness of a test suite to detect faults in the system under test.

2.3 What Is Mutation Testing?

Mutation testing is, according to an extensive survey by Jia et al. [14], a "fault-based testing technique which can be used to measure the effectiveness of a test set in terms of its ability to detect faults". The technique introduces faults, which are represented as syntactical changes to the program's binary or source code called *mutations*, to create faulty programs named *mutants*. Each mutant is then tested by executing an existing test set of the program. If the result of running the test set with the mutant differs

from the result of running the test set with the original fault-free program, then the mutant has been killed, i.e., it has been detected by the test set. If the mutant has not been killed, either the test set needs to be extended to detect the modification or the mutant is equal to the original program. The effectiveness of the test set can then be calculated by the number of killed mutants by the total amount of tested mutants. This metric is called the *mutation score* which can be used to directly compare two distinct test sets for the same program. Hence it can, for instance, be used to compare handwritten to automatically generated test sets of programs. A mutation score of 1.0 is most desirable since this means that a test set killed all mutations.

Significant to the field of mutation testing are two hypothesis defined by DeMillo et al. [12]: the *Competent Programmer Hypothesis* and the *Coupling Effect*. They imply that it should be sufficient to examine only a subset of simple faults of the overall possible program faults, to be effective in finding real faults [14].

- The *Competent Programmer Hypothesis* states that "programmers are competent, which implies that they tend to develop programs close to the correct version" [14]. It can be therefore assumed that the existing faults in a program are simple. Therefore, it is sufficient to introduce simple faults by mutation testing because it mimics the faults that are made by competent programmers.

- The *Coupling Effect* states that "test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors" [12]. Therefore, it is sufficient to examine only simple errors since they are coupled with complex errors in the program under test.

The fundamental components of mutation testing can be derived from the traditional process described in [12, 14], namely: the list of `Mutation Operators` to apply modifications and a `Test Set` of the program under test. These components, as well as their typical interactions, are depicted in Figure 2.3. First the original program P is executed against the test set T, producing the test result R. Then each mutation operator M takes the original program under test P and applies its modifications. The outcome is a set of modified programs P', the so called mutants. These modified programs P' are then executed against the test set T. The produced test results R' of this step are compared to the original test result R. If the results are equal, the mutant P', and therefore its modifications, has not been detected by the test set T.

Although the definition of mutation testing states that it is applied to detect faults in programs, it can also be used to uncover implementation flaws such as dead code. Furthermore, mutation testing is mostly described to solely cover techniques for program source code. However, it has also been used to mutate program specifications [14]. The

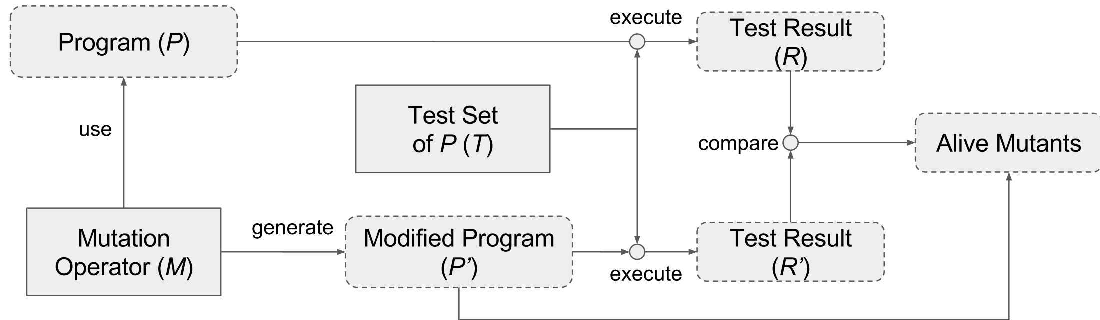


Figure 2.3: Fundamental Components of Mutation Testing

latter direction is called *specification mutation*, which is a black-box or grey-box testing technique depending on what program parts are used, while the first direction is called *program mutation* which solely focuses on the program binary or source code and is therefore a white-box testing technique [14].

This thesis utilizes mutation testing in Chapter 7 to evaluate the effectiveness of generated test suites. The direction of program mutation is used, to emphasize how much of the system under test is actually covered. Since everything that has been implemented for this thesis is written in the programming language Go, it is only fitting to include in the evaluation of this thesis at least one scenario of a Go program. Chapter 6 introduces GO-MUTESTING to perform the mutation testing part for the evaluation of the mentioned scenario.

Determining the effectiveness of test suites is just one discipline that needs to be considered while generating test cases. Another discipline arises when individual test cases are executed. The execution of a test case either verifies that the system under test behaves as defined by the test case, or the test case fails which highlights a fault that must be investigated. One possible method to aid in the time-consuming and complicated process of investigating and patching faults is the reduction of failing test cases. The subsequent section introduces *delta-debugging*, an automatic technique to systematically reduce data such as test cases.

2.4 What Is Delta-Debugging?

Delta-debugging is defined by Zeller in [22] as an “automatic technique that narrows down a cause by running automated experiments”. The typical use case is the reduction of input data which lets a program fail during execution. Less data has the benefit of having to consider less context during debugging a problem. Therefore, delta-debugging makes it possible for the user to focus only on relevant parts of the failure-inducing data. Note that finding the minimal representation, which still provokes the same program

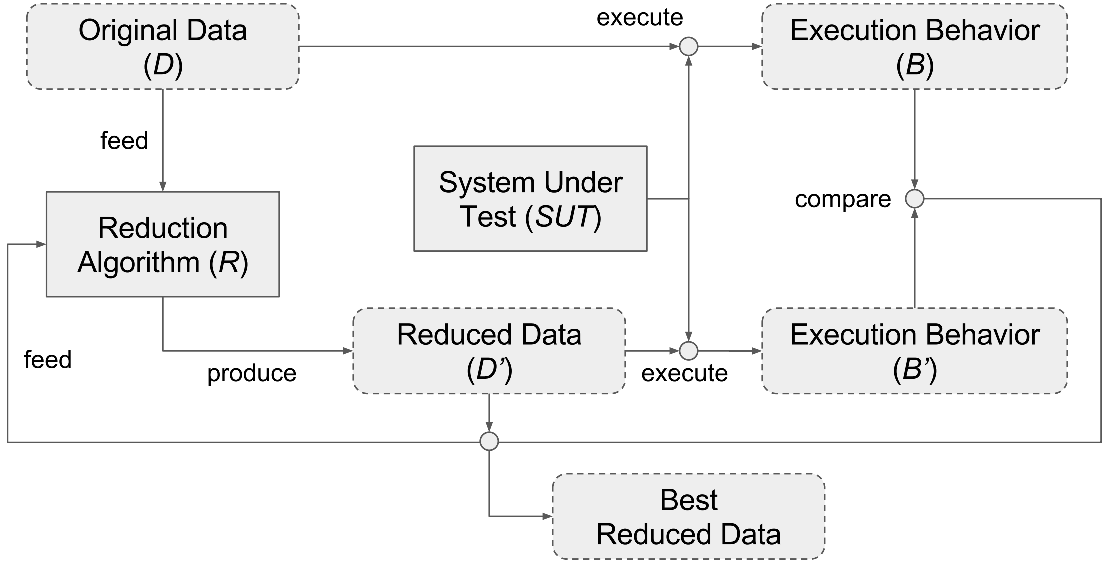


Figure 2.4: Fundamental Components of Delta-Debugging

behavior of the original data, is not guaranteed by process of delta-debugging [10, 11]. However, in practice every reduction of the context that is needed to debug a problem is already an improvement. Furthermore, delta-debugging can be used to reduce and isolate anything, given appropriate implementations of the delta-debugging phases. For instance a solution might be reduced to be more readable or a huge formula is rewritten to a smaller formula which has the same behavior.

The fundamental components of a delta-debugger, which can be derived from the original algorithm [22], are the `Reduction Algorithm` and the `System Under Test`. These components, as well as their typical interactions, are depicted in Figure 2.4. First the original data D is executed with the system under test. The resulting execution determines the original behavior B which must be preserved during the reduction. Next the original data D is fed into the reduction algorithm. The algorithm determines which parts of the data should be reduced for the current reduction iteration, and how the selected part is reduced, i.e., removed or substituted. The reduced data D' is then executed with the system under test. The resulting behavior of the execution B' is compared to the behavior of the execution with the original data B. The result of this comparison is fed back into the reduction algorithm. If both behaviors are the same, the original behavior got preserved and the reduction algorithm continues with the next reduction iteration. Otherwise the last reduction is reverted and the algorithm proceeds. This reduction loop continues until predefined conditions are met, e.g., a timeout occurs or the algorithm does not find new parts to reduce.

Considering the main components and workflow of delta-debugging, we can define the following three phases:

- The *search phase* decides which part of the data will be reduced next.
- The *reduction phase* removes repetitions and optional data or replaces data with something else, e.g., uninteresting complex functions are replaced with constant values.
- The *testing phase* checks if the reduced data still provokes the original failure.

The original delta-debugging algorithm [22] reduces the given data at the character level, ignoring the syntactical structure of the data. Hence, making it likely to produce invalid data, leading to a high amount of rejected reduction iterations. By making the search phase aware of the data's structure an immense reduction in iterations can be achieved, dramatically speeding up the delta-debugging process [18, 10, 11]. Additionally, making the reduction phase aware of the content itself, can be very efficient in reducing the original data greatly while also reducing the runtime of the process [10, 11].

Making the delta-debugging implementation aware of the data's structure and content, basically giving the implementation an almost or even complete model of the provided data, is one of the goals of this thesis. Section 3.5 describes the approach and implementation which completes the goal of combining fuzzing and delta-debugging by utilizing the same underlying model.

Chapter 3

The Tavor Framework

The main goal of TAVOR is to provide functionality for the automatic generation, alteration and reduction of test data which conforms to a predefined structure. Programs oftentimes need to process complex data structures that allow for an enormous number of possible variants. Such programs are therefore hard to test, since a tester has to consider every important variant for testing purposes. To construct the input data manually is time-intensive and error-prone. TAVOR can be used to automate the generation of this data, to alter existing data and to apply delta-debugging to systematically reduce existing data according to a predefined structure and its constraints.

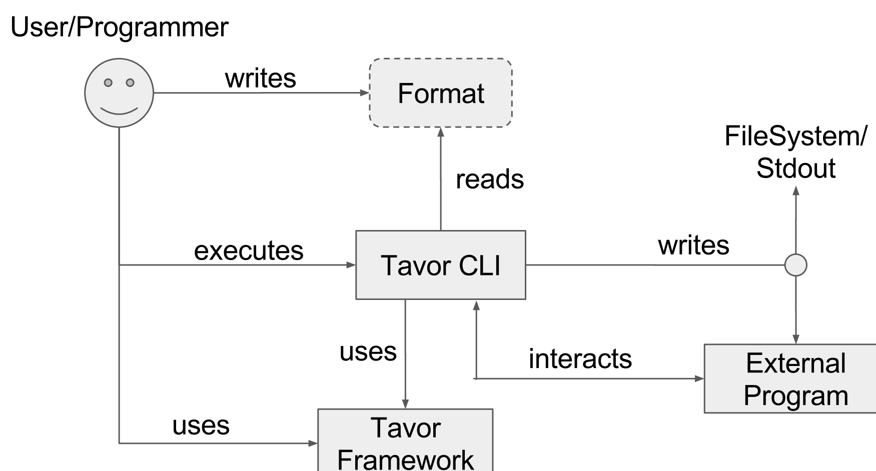


Figure 3.1: TAVOR's Subsystems

Figure 3.1 depicts the interactions of the TAVOR subsystems. In order to interact with an external program a user first needs to define a data model which describes the structure of the expected inputs to the program. The TAVOR FRAMEWORK provides its own format to define such data models, which covers all functionality of the framework. Please refer to Chapter 4 for a detailed description of the TAVOR FORMAT and its capabilities. Next, the TAVOR CLI can be used to interact with an external program. Interactions such as fuzzing or delta-debugging are also provided by the TAVOR CLI, which is described in more detail in Chapter 4. The TAVOR CLI relies heavily on the algorithms and data structures provided by the TAVOR FRAMEWORK. The framework's

design goals as well as its components and their interactions are the main focus of the subsequent sections of this chapter.

3.1 Components

The Tavor framework offers various components which are depicted in Figure 3.2. There are three utility components, i.e., `Tokens`, `Parsers` and `Logging`, as well as three components offering access to different kinds of algorithms, i.e., `Fuzzing Strategies`, `Fuzzing Filters` and `Reducing Strategies`.

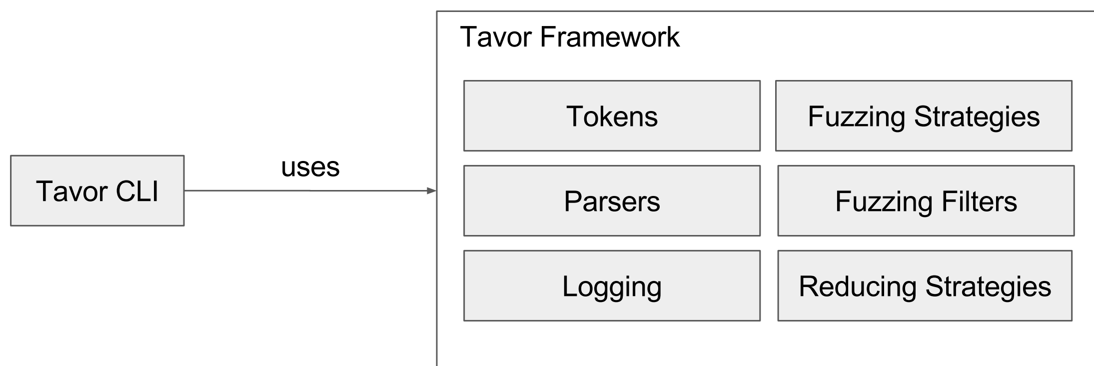


Figure 3.2: Components of the Tavor Framework

In order to combine both, fuzzing and delta-debugging, all implemented methods of the Tavor framework operate on one internal model-based structure. This structure is basically a graph of nodes that are called `token graphs` throughout the Tavor framework. Each node represents an instance of a `token` of a framework-defined type. A token graph can either be directly instantiated or by parsing a format file using the `Parsers` component of the Tavor framework. The `Logging` component provides extensive logging for debugging the handling of tokens. The granularity of logged output can be controlled by specifying the desired log level, i.e., the debug level reports the widest range of information while the error level only informs about occurred errors.

Instead of focusing on only one technique Tavor's components have been kept generic. Dedicated techniques and heuristics can be implemented and executed independently. All of these components operate on tokens. In order to generate permutations of a specific token, i.e., unique arrangements of the token's values, `fuzzing strategies` are used. A token itself is not fixed to a static definition but can be changed by `fuzzing filters` to apply additional techniques such as boundary-value analysis of ranges. For delta-debugging so called `reducing strategies` can be implemented and used.

Every algorithm of the Tavor framework has to be deterministic, since determinism has the advantage that it eases debugging problems and enables the reproduction of results that are sent to external programs. Therefore, no functionality is allowed to have its own source or seed of randomness. Instead, a common interface that defines a random generator is used throughout the framework. An implementation of this interface has to be deterministic given the same seed of randomness. The decision of keeping code deterministic was also applied to concurrent code and tests, making it possible to reproduce the same output given the same random seed and version of the framework.

The following subsections present Tavor's fundamental implementation decision as well as its main components `Tokens`, `Fuzzing Strategies`, `Fuzzing Filters` and `Reducing Strategies` in more detail.

3.2 Tokens

The basic building blocks of the Tavor framework are its *tokens*. These tokens differ from *lexical analysis tokens* in the following way: They represent not just a group of characters, but different kinds of data with additional properties and abilities. Tokens can be constant integers and strings of all kind as well as dynamic data such as integer ranges, sequences and character classes. Furthermore, tokens can encapsulate other tokens to group them together and to create building blocks that can be reused to, for example, repeat a group of tokens. Tokens can have states, conditions and can perform operations such as arithmetic. They can create new tokens dynamically and can depend on other tokens to generate data. Tokens are basically the foundation of the framework and every algorithm for parsing, fuzzing and delta-debugging is relying on them.

A *permutation* of a token is a specific arrangement of its values. Every token type has its own arrangement implementation and every token instance can have its own values. However, the set of permutations is unique for a specific token type and values, as is the order of these permutations. For example a token holding a range of numbers from “4” to “6” has 3 possible permutations: “4”, “5” and “6”. The first permutation of this token is “4” and the third permutation is “6”. Even though this demonstrates that a token can represent many permutations, every token can hold only one permutation at any given time. Therefore, every token implementation should differentiate between an internal and external representation. Since it is necessary to save the possible values internally but to forward the current value externally. This distinction is especially important for token types that generate new tokens out of their internal representation. The

original internal tokens should not be connected to the external ones, since it would be otherwise possible to change the internal representation without any contract.

Many of the algorithms in the TAVOR FRAMEWORK build upon the assumption that the processed token graphs are acyclic. This decision allows for easier implementations and usage since no guards and functionality have to implemented to deal with infinitely running algorithms and unwanted repetitions. Hence, while the internal structure allows tokens to have loops in their graphs, each loop must be unrolled before it can be used with the algorithms of the TAVOR FRAMEWORK, i.e., the bodies of repetitions are copied N-times instead of the original repetition. An example for a token graph with a loop is given in Figure 3.3 which is then unrolled twice resulting in the graph of Figure 3.4.

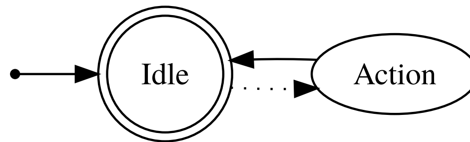


Figure 3.3: Example for a Graph With a Loop

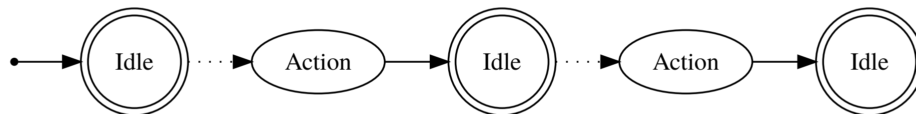


Figure 3.4: Example for an Unrolled Graph

Every token implements at least the basic *Token Interface*, which specifies methods for generating, replicating, permutating and parsing. Additional token interfaces add specific functionality to a token. The *List Interface*, for example, states that a token can have child tokens and specifies methods to access them.

The Token Interface is the base interface of all tokens. Its operations can be grouped into the following categories:

- *Generation* to output the external representation of the token.
- *Replication* to create an exact copy of the token.
- *Permutation* to permutate the token.
- *Parsing* to parse arbitrary input to permutate the token.

Subsection 3.2.1 presents an example token implementation which exemplifies the above-mentioned fundamental token concepts. Subsequently Subsection 3.2.2 elabo-

rates on advanced token concepts, which are for instance necessary to model repetitions or constraints.

3.2.1 Example Implementation - The Smiley Token

This subsection illustrates the implementation of a basic Tavor Token, thus explaining the fundamental token operations by example. Since the Tavor framework is written in the programming language Go, the following implementations are written in Go as well. Go is an open source programming language created at Google. The language is garbage collected which allows the following examples to be nearly uncluttered of memory handling. Furthermore, Go's C-like syntax should be widely readable with the exceptions of three operators: The *short variable declaration* operator ":=" declares and initializes a variable using the value and variable type of the right-hand side of the operator. The range operator allows the programmer to iterate over an array using a for-loop, e.g., `for i, p := range people` iterates over the variable `people` with `i` holding the index and `p` holding the item of the current iteration. Lastly, the "go" operator starts the encapsulated function call in a new coroutine.

Consider a token defining a smiley which has eyes, a mouth and can have a nose. The token should be able to generate different permutations of smilies and even parse them. This example uses two different kinds of eyes: "`:`" and "`;`", an optional nose "`-`" and three different kinds of mouths: "`)`", "`(`" and "`D`". Thus, we allow for instance the smiley "`;-D`", but not "`:-<`". This example should in general show how easy it is to create new token types. It must be noted that this example could be easily implemented with the available token types or with the Tavor format shown in Listing 1.

Listing 1 Smiley Tavor Format

```
1 START = [:;] ?("-") [)(D]
```

Listing 2 shows the basic data structure of the Smiley Token. Since the Smiley Token has to hold three different types of information, it is necessary to create a structure. Instead of directly using the characters for the eyes and mouth in the structure, only indices are used. This is not necessary but helps to separate the constant data from the permutation of the token. Using this data structure the smiley "`;-D`" would be represented with the composite literal `Smiley{eyes: 1, nose: true, mouth: 2}`. The structure must implement the Token Interface, which is grouped into method categories as described in Section 3.2. The implementations of these categories are now presented in the remainder of this section.

Listing 2 Smiley Data Structure

```
var (
    eyes   = ":;"
    mouths = ")(D"
)

type Smiley struct {
    eyes  int
    nose  bool
    mouth int
}
```

The *Generation* category of the Token Interface deals with the output of the current permutation. To implement this category at least the `String` method has to be implemented, which returns the textual representation of the token's current permutation. The `String` method for the Smiley Token is shown in Listing 3, and returns for the smiley `Smiley{eyes: 1, nose: true, mouth: 2}` the string "`;-D`". Since the internal representation should not be accessible by the token's user, no safeguards, e.g., for accessing array items by their indices, are necessary.

Listing 3 Smiley `String` Method

```
func (s *Smiley) String() string {
    nose := ""
    if s.nose {
        nose = "-"
    }

    return string(eyes[s.eyes]) + nose + string(mouths[s.mouth])
}
```

The *Replication* category deals with replicating tokens, i.e., with creating copies of tokens. The `Clone` method is the only replication function required by the Token Interface. Note that the new token must be independent of the original token, i.e., that each `Clone` function needs to perform a deep rather than a shallow copy of its data. For the Go programming language this means that token internal slices, maps, structures and token children must be copied as well. The implementation of the `Clone` method for the Smiley Token is shown in Listing 4.

The *Permutation* category handles the set of permutations for a token as well as its current permutation. The method `Permutations` defines how many permutations a single token holds. The Smiley Token has a constant number of 12 permutations, since the amount of eyes, mouths and noses is constant. Other tokens such as ranges

Listing 4 Smiley `Clone` Method

```
1 func (s *Smiley) Clone() token.Token {
2     return &Smiley{
3         eyes:  s.eyes,
4         nose:  s.nose,
5         mouth: s.mouth,
6     }
7 }
```

of integers depend on their configuration. The implementation of the `Permutations` method for the Smiley Token is shown in Listing 5.

Listing 5 Smiley `Permutations` Method

```
1 func (s *Smiley) Permutations() uint {
2     return uint(len(eyes) * 2 * len(mouths))
3 }
```

The method `PermutationsAll` returns the number of permutations of the token itself and all its children. Since the Smiley Token has no children, it is the same as `Permutations` which is shown in Listing 6. However, it is important to note that calculating the amount of permutations is not always a straightforward task. Consider for instance the *Concatenation Token* representing a sequence of tokens, which requires all of its children to be present, and the *One Token*, which requires exactly one of its children to be present. These two tokens have very different permutation calculations. For the Concatenation Token the product of its children's `PermutationsAll` result is calculated using $\prod_{i=1}^{token.NumChildren} Token.Child(i).PermutationsAll()$ while for the One Token the sum of its children's `PermutationsAll` result is calculated using $\sum_{i=1}^{token.NumChildren} Token.Child(i).PermutationsAll()$.

Listing 6 Smiley `PermutationsAll` Method

```
1 func (s *Smiley) PermutationsAll() uint {
2     return s.Permutations()
3 }
```

The `Permutation` method completes the *Permutation* category. It sets a distinct permutation of the token, i.e., calling the method with the integer 11 results in `Smiley{eyes: 1, nose: true, mouth: 2}` which represents the `;-D` smiley. The implementation of the `Permutation` method for the Smiley Token is shown in Listing 7, and has been intentionally been made inefficiently to keep the example simple.

Listing 7 Smiley `Permutation` Method

```
func (s *Smiley) Permutation(i uint) error {
    if i < 0 || i >= s.Permutations() {
        return NewPermutationErrorIndexOutOfBound()
    }

    p := uint(0)
    for eyes := range eyes {
        for _, nose := range []bool{false, true} {
            for mouth := range mouths {
                if i == p {
                    s.eyes = eyes
                    s.nose = nose
                    s.mouth = mouth

                    return nil
                }

                p++
            }
        }
    }

    return nil
}
```

Finally, the `Parsing` category, which is the last category of the Token Interface, deals with parsing the token from an input. Listing 8 shows the `Parse` method of the SMILEY Token. The implementation may look very verbose, but it is necessary to handle every syntax error to generate adequate parsing errors. Note that these messages could be further improved by not only stating that something was expected, but actually giving examples on what has been expected.

The SMILEY Token can then be used to create structures like with any other token of the framework. However, to give an easier example, it will be used alone. Listing 9 creates a new SMILEY Token, permutates over all permutations and parses a string. Each step is printed to STDOUT. Resulting in the output shown in Listing 10.

Please note, that the smiley token has been used to demonstrate, which standard methods each token type needs to implement. Of course, the TAVOR FRAMEWORK already offers a wide range of token types allowing to represent arbitrary formats. Chapter 4 gives an overview of the format options that are already implemented by the tokens of the TAVOR FRAMEWORK.

Listing 8 Smiley Parse Method

```
1  func (s *Smiley) Parse(parser InternalParser, cur int) (int, []error)
   ↪  {
2      if cur+2 > parser.DataLen {
3          return cur, []error{errors.New("Out of data for a smiley")}
4      }
5
6      if i := strings.IndexRune(eyes, parser.Data[cur]); i != -1 {
7          s.eyes = i
8      } else {
9          return cur, []error{errors.New("Expected some eyes")}
10     }
11     cur++
12
13     if parser.Data[cur] == '-' {
14         s.nose = true
15         cur++
16     } else {
17         s.nose = false
18     }
19     if cur >= parser.DataLen {
20         return cur, []error{errors.New("Out of data for a mouth")}
21     }
22
23     if i := strings.IndexRune(mouths, parser.Data[cur]); i != -1 {
24         s.mouth = i
25     } else {
26         return cur, []error{errors.New("Expected a mouth")}
27     }
28     cur++
29
30     return cur, nil
31 }
```

3.2.2 Advanced Token Concepts

While Tavor actually lacks the Smiley Token introduced in Subsection 3.2.1, it offers a wide range of different token types, which can be used to model arbitrary formats. The following categories of tokens give an excerpt of the different tokens supported by Tavor:

- *Primitive* Tokens do not reference other tokens, i.e., they are vertices in the token graph without any outgoing edges. These tokens solely implement the basic Token Interface. The plainest types are the `Integer` and `String` Tokens, which represent constant data of numbers and texts, e.g., “1234” and “”Hello World”.

Listing 9 Working With the Smiley Token

```
1  func main() {
2      s := Smiley{}
3
4      fmt.Print("Permutations:")
5      for i := uint(0); i < s.Permutations(); i++ {
6          s.Permutation(i)
7          fmt.Print(" ", s.String())
8      }
9      fmt.Println()
10
11     p := NewInternalParser(":-D")
12     _, errs := s.Parse(p, 0)
13     if errs != nil {
14         panic(errs)
15     }
16
17     fmt.Println("Parsed:", s.String())
18 }
```

Listing 10 Output of Smiley Example

```
1  Permutations: :) :( :D :-) :-( :-D ;) ;( ;D ;-) ;-( ;-D
2  Parsed: :-D
```

Other examples of this category, such as the as `RangeInt` which allows to define a range of integers, are dynamic, i.e., can have more than one permutation, but can represent only one permutation at any given time. Another example is the `CharacterClass` Token, most commonly known from regular expressions, which allows to define character class definitions such as `[ACE]` allowing either one of the characters “A”, “C” or “E”.

- *List* Tokens contain a list of child tokens. Additionally to the basic Token Interface these tokens also implement the List Interface, which provides methods to operate on the token’s children. At the moment of writing the token types `Concatenation`, `Once`, `One` and `Repeat` are currently supported. The `Concatenation` (resp. `Once`) Token requires all children to be present in the defined order (resp. an arbitrary order). The `One` Token allows to define lists were exactly one child token is present in every permutation. Finally the `Repeat` Token is used to repeat defined tokens for a specified amount of times.

- *Constraint* Tokens are used to bind tokens to constraints, e.g., the `Optional` Token expresses that the wrapped token may or may not be present.

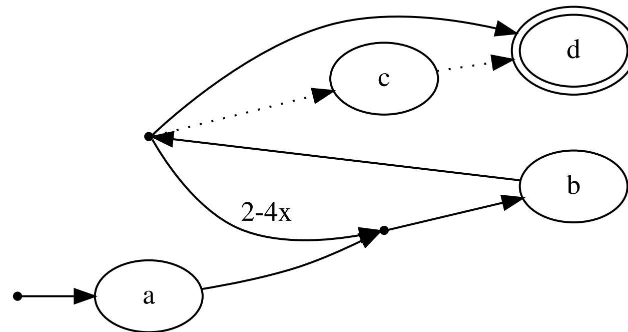


Figure 3.5: Automaton of Simple Format Definition

- *Expression* Tokens are used to define formulas, consisting of one or more token, which can be resolved to a single value, which is again a token. Such tokens can be for example arithmetically such as `ArithmeticAdd`, `ArithmeticSub`, `ArithmeticMul` and `ArithmeticDiv` which allow simple calculations of numbers. The `BooleanTrue` and `BooleanEqual` are another example of expression tokens, which belong to the scope of boolean algebra.

- *Statement* Tokens are used to handle control flows in token graphs, e.g., the `If` Token uses a boolean expression to guard a child token which is only active if the expression is true.

- *Variable* Tokens have been implemented in order to be able to reuse and interact with generated values at another place in a token graph.

Every token type and interface can have its own *token attributes*, which are used to access meta information about a token. Each List token, for instance, provides the attributes `Count` to retrieve to the number of its child tokens, `Item(i)` to refer to its i-th child and `Unique` to refer to an arbitrary but unique child of the list.

Consider Figure 3.5, it shows an automaton which allows an "a" character, followed by two to four "b", an optional "c" and a concluding "d". The token graph representing this format is shown in Figure 3.6. It is rooted in a `Concatenation` Token which has four child tokens. The starting and concluding tokens are simply constant strings, i.e., they need to be present in every permutation. The repetition of "b" is expressed using a `Repeat` Token with the configuration `from=2` and `to=4`. In order to model that the String Token "b" is optional it is wrapped in an `Optional` Token.

Most algorithms of the TAVOR FRAMEWORK traverse a token graph and perform a certain operation on each traversed token. Auxiliary functions are provided by the framework to fulfill this purpose. One of these functions is `Walk` which is depicted as pseudo code in Listing 11. It receives two input parameter: the `root` Token which needs

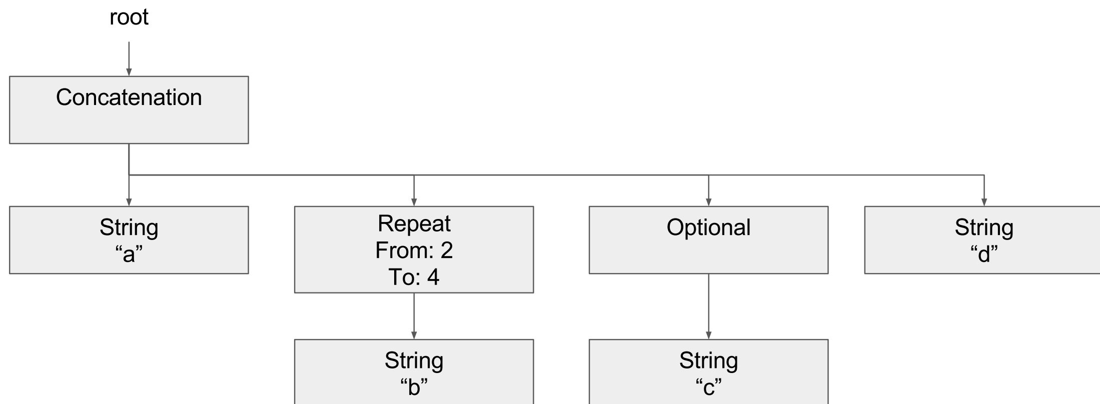


Figure 3.6: Token Graph of Simple Format Definition

to be traversed and the function `walkFunc` which is called for each traversed token. A queue is used to systematically traverse the token graph. Remember, no cycles are allowed within a token graph. Hence, no loop-detection needs to be implemented. Tokens that implement the Forward Interface, which reference a single token, are handled in Line 14, which pushes the referenced child for later processing onto the queue. The List Interface is implemented by those token types who refer to a list of child tokens, hence the loop from Lines 16-18 is used to add all referenced children onto the queue.

3.3 Fuzzing Strategies

The TAVOR FRAMEWORK provides the concept of fuzzing strategies to generate permutations of a token graph according to a specific algorithm. There are two fundamental ways of token fuzzing. One is to deterministically choose one possible permutation, the other is to choose randomly out of all permutations of a token. All fuzzing strategies need to implement the following interface:

```
type Strategy func(root Token, r rand.Rand) (chan bool, error)
```

This interface defines a single function, which initializes the strategy and then starts the first fuzzing iteration in a new coroutine. If an error is encountered during the initialization of the strategy, the error return argument is not nil. On success a channel is returned, which controls the fuzzing process. In the GO programming language channels are used to communicate among coroutines. For each completed iteration of the fuzzing strategy a value is returned by the channel. The caller of the strategy needs to put back a value into the channel, to initiate the calculation of the next fuzzing iteration. This passing of values is needed to avoid data races within the token graph.

Listing 11 Token Graph Walk Function

```go
func Walk(root Token, walkFunc func(token Token) error) error {
	queue := NewQueue()
	queue.Push(root)

	for !queue.Empty() {
		token := queue.Pop()

		if err := walkFunc(token); err != nil {
			return err
		}

		switch t := token.(type) {
		case Forward:
			queue.Push(t.Get())
		case List:
			for i := t.Len() - 1; i >= 0; i-- {
				queue.Push(t.Get(i))
			}
		}
	}

	return nil
}
```

Note that the channel must be closed either when there are no more iterations, or in case the caller of the strategy wants to end the fuzzing process.

The Tavor framework provides the following default fuzzing strategies:

- The *Random* strategy generates exactly one permutation of the passed in token graph by permuting each reachable token randomly. The determinism is dependent on the random generator and is therefore deterministic if the same random seed is used for initializing the random generator.

- The *AllPermutations* strategy deterministically generates all available permutations of a token.

- The *AllmostAllPermutations* strategy generates a subset of all available permutations by not covering all possible permutations of Repeat Tokens. This capability is especially helpful when less permutations are needed than provided by the *AllPermutations* strategy.

- The *PermuteOptionals* strategy searches the graph for tokens implementing the `Optional Interface` and permutates over them by deactivating or activating them. The permutations always start from the deactivated states in order to generate minimum data first.

In order to provide a better understanding of the concept of fuzzing strategies we walk through an example implementation of a fairly basic fuzzing strategy in Section 3.3.1. Additionally, we offer in Section 3.3.2 the pseudo code of the AllPermutations strategy, which is provided by the TAVOR FRAMEWORK, and illustrate the algorithm using a walk-through for an example token graph.

3.3.1 Basic Example Fuzzing Strategy

Consider a fuzzing strategy, which traverses the token graph looking for Integer tokens. Each integer having a value between 1 and 10 is incremented by one therefore replacing the original value. This strategy falls in the category of mutation-based fuzzing, since it does change the original model. It is also stateless since there is no need to keep track of current events between iterations. The graph is simply traversed and changed once per fuzzing iteration.

Listing 12 depicts the pseudo code of the sample fuzzing strategy. First the channel to steer the fuzzing strategy is created in Line 2. Next a coroutine is started, see Lines 4-28, which makes use of the auxiliary function `Walk` to traverse the token graph. For each traversed token the function defined in Lines 8-19 is called, which is responsible to adapt the content of the currently traversed token. In case it is a Integer token holding a value between 1 and 10 the current token is adapted. After the traversal of the token graph the completion of the current iteration is reported to the `continueFuzzing` channel if at least one token has been adapted. Otherwise the channel is closed and the coroutine terminates.

One way to execute this strategy is by using the pseudo code shown in Listing 13. Note that the implemented strategy does not need a random generator, hence, this argument for the function can be `nil`. The produced output of this program is shown in Listing 14, showing that the initial permutation `7 9` is fuzzed four times untill all constant integers have reached the value 11.

The last step when creating a fuzzing strategy is to make the strategy known to the TAVOR FRAMEWORK. In order to do so, a register function is provided, which allows to register fuzzing strategies based on an identifier. This is especially needed for the Tavor CLI, which can execute a specific fuzzing strategy defined by a command line

Listing 12 Sample Fuzzing Strategy

```
1  func NewSampleStrategy(root Token, r rand.Rand) (chan bool, error){
2      continueFuzzing := make(chan bool)
3
4      go func() {
5          for {
6              found := false
7
8              err := Walk(root, func(token Token) error {
9                  intToken, ok := token.(*Integer)
10                 if !ok { return nil }
11
12                 v := intToken.Value()
13                 if v >= 1 && v <= 10 {
14                     found = true
15                     intTok.SetValue(v++)
16                 }
17
18                 return nil
19             })
20             if err != nil { panic(err) }
21             if !found { break }
22
23             continueFuzzing <- true
24             if _, ok := <-continueFuzzing; !ok { return }
25         }
26
27         close(continueFuzzing)
28     }()
29
30     return continueFuzzing, nil
31 }
```

argument. The sample fuzzing strategy can be registered with the Tavor framework using the code shown in Listing 15.

3.3.2 The AllPermutations Fuzzing Strategy

The *AllPermutations* fuzzing strategy is used to generate all permutations of a token graph in a deterministic manner. Consider the token graph in Figure 3.7, which describes a format allowing one or two “a” characters followed by “b”, “c” or “d” and concluded by an Optional Token “e”. Every token can tell how many permutations it has, i.e., the Concatenation Token has just one permutation as it always requires each

Listing 13 Callee of Example Fuzzing Strategy

```
1  func main() {
2      var root token.Token = lists.NewConcatenation(
3          NewInteger(7),
4          NewString(" "),
5          NewInteger(9),
6      )
7
8      continueFuzzing, err := NewSampleStrategy(root, nil)
9      if err != nil { panic(err) }
10
11     for i := range continueFuzzing {
12         fmt.Println(root.String())
13         continueFuzzing <- i
14     }
15 }
```

Listing 14 Command Line Output of Example Fuzzing Strategy

```
1 8 10
2 9 11
3 10 11
4 11 11
```

of its children to be present. The Optional Token on the other hand is capable of two permutations, i.e., either its child token “e” is active or inactive.

In order to provide a better understanding on how the AllPermutations strategy operates, we first take a look at the computation of the number of possible permutations of the token graph. Consider Table 3.1, it lists the formulas for calculating the number of possible permutations per token type. Applying these formulas to the token graph shown in Figure 3.7, results in $(1^1 + 1^2) * (1 + (1 + 1)) * (1 + 1) = 2 * 3 * 2 = 12$. These 12 supported permutations are listed in Listing 16.

Table 3.1: Computing PermutationsAll for Different Token Types

Token Type	**Formula**
Concatenation	$\prod_{i=1}^{token.NumChildren} Token.Child(i).PermutationsAll()$
Repeat	$\sum_{i=From}^{To} Token.Child().PermutationsAll()^i$
One	$\sum_{i=0}^{token.NumChildren} Token.Child(i).PermutationsAll()$
Optional	$1 + token.Child().PermutationsAll()$
String	1

Listing 15 Registering the Sample Fuzzing Strategy

```
1 func init() {
2     strategy.Register("SampleStrategy", NewSampleStrategy)
3 }
```

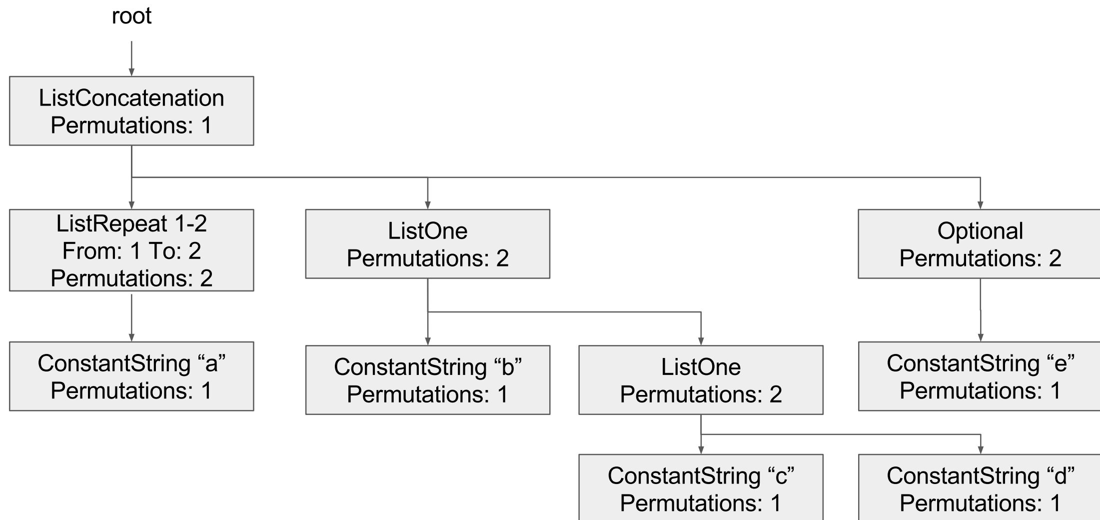


Figure 3.7: AllPermutations Example Token Graph

To iterate over all permutations of a token graph we view each token as a single digit in its own numeral system. In common numeral systems every digit has a fixed range. For instance, the decimal numeral system allows digits from 0 to 9. When incrementing a number over its range in one of those systems, a digit propagates an overflow to its neighboring higher digit and resets itself to the first digit of its range. Figure 3.8 illustrates this propagation in the decimal numeral system on the calculations $0+1 = 1$, where no propagation takes place, $9+1 = 10$, where one digit propagates and $99+1 = 100$, where two digits propagate their overflow. A very similar concept is applied on token graphs when iterating over their permutations. In contrast to common number systems, each token may have a different range of values, i.e., an Optional Token can have the values 0 and 1 while a Repeat Token from 0 to 10 may have values from 0 to 10. In common numeral systems each digit has at most two neighboring digits, in comparison each token in a token graph may have a parent, siblings as well as child tokens which all need to be considered when calculating the next permutation.

For calculating the next fuzzing iteration, i.e., the next permutation of the token graph, the AllPermutations strategy applies the following procedure on each token:

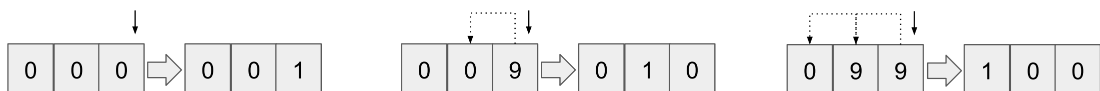


Figure 3.8: Incrementing in the Decimal Numeral System

Listing 16 Permutation of AllPermutations Example Token Graph

```
1  ab
2  aab
3  ac
4  aac
5  ad
6  aad
7  abe
8  aabe
9  ace
10 aace
11 ade
12 aade
```

- If a token has children, first try to increment the permutation of its children. In case no overflow takes place, the next permutation has been successfully calculated.

- If there are no children, or incrementing their current permutation results in an overflow, try to increment the current token's permutation. In case no overflow takes place, the next permutation has been successfully calculated.

- If incrementing the permutation of the current token results in an overflow, propagate this overflow either to the neighboring higher sibling, or in case there are no siblings, the parent token. If the current token is the root token, this means the last permutation has been reached.

Figure 3.9 depicts the individual steps of the AllPermutations fuzzing strategy for the token graph shown in Figure 3.7. The solid arrows symbolize an increment instruction for a token, and the dashed arrows are used to depict the propagation of an overflow. The token graph is initialized with value 0 for all of its tokens, which represents the permutation "ab". In order to step to the next permutation first an increment command is sent to the root token in iteration 0, which propagates this increment to its first child token the Repeat Token, which in turn also propagates to its child the String Token "a". A String Token has only one permutation, hence, the increment command overflows setting the Repeat Token to value 1, resulting in the permutation "aab". In iteration 1 the same computation takes place, except that this time also the Repeat Token overflows setting the first One token to value 1, resulting in permutation "ac". This process continues until iteration 11, where the root token itself reports an overflow, hence no more permutations are available.

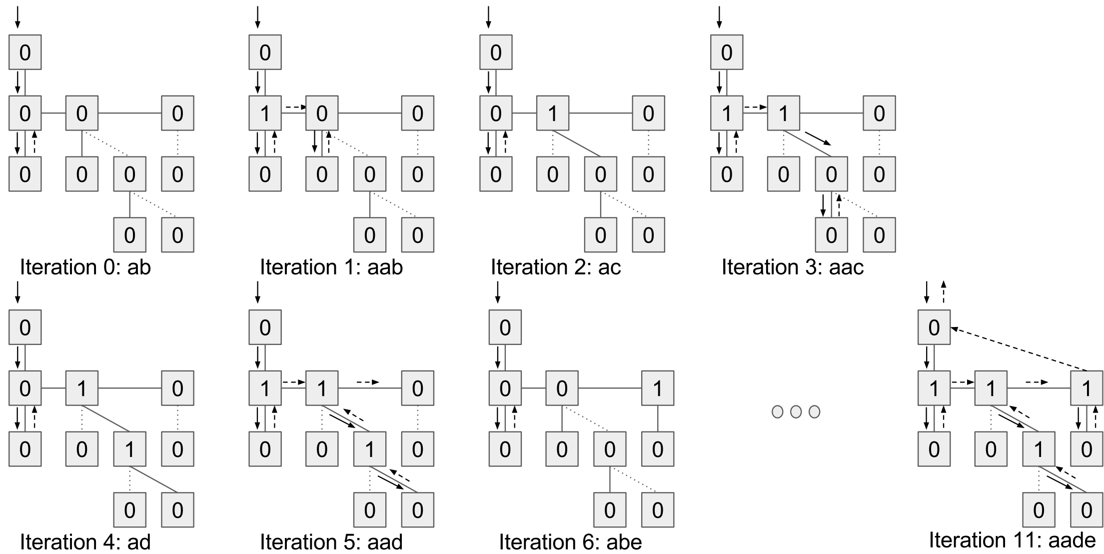


Figure 3.9: AllPermutations Iterations

Please refer to Appendix A Listing 83 and Listing 84 for the pseudo code of the AllPermutations fuzzing strategy.

3.4 Fuzzing Filters

In order to mutate a token graph, i.e., apply mutation-based fuzzing, the Tavor framework offers *fuzzing filters*. Fuzzing filters do not change a specific permutation, but alter all permutations that may be generated with a specific token graph. All fuzzing filters need to implement the following interface:

```
type Filter func(token Token) (Token, error)
```

This interface specifies a generic function, which applies the given filter onto a single token that is passed to the function. Therefore, the function's concern is only one token at a time. If an error is encountered during the filter execution, the error return argument is uninitialized. On success a replacement for the token is returned. If this replacement is not `nil`, it will replace the original token. Consider, for instance, a very basic filter that replaces each occurrence of the string "old" with the new constant string "new". A possible implementation of this filter is shown in Listing 17.

The Tavor framework offers a register function for fuzzing filters, which works along the same lines as the register function for fuzzing strategies. It allows to register filters based on an identifier, which can later on be used within the framework. The sample

Listing 17 Sample Filter

```
1 func NewSampleFilter(token Token) (Token, error) {
2     c, ok := token.(*String)
3     if !ok || c.String() != "old" {
4         return nil, nil
5     }
6
7     return NewString("new"), nil
8 }
```

filter of Listing 17 can be registered with the TAVOR FRAMEWORK using the code shown in Listing 18.

Listing 18 Registering the Sample Filter

```
1 func init() {
2     filter.Register("SampleFilter", NewSampleFilter)
3 }
```

Applying a filter can be done manually or using the *ApplyFilters* function. An excerpt of this function is shown in Listing 19. ApplyFilters traverses the graph using a queue of pairs holding the token to process and its parent. The root token of the input token graph is special, in the sense, that it does not have a parent. The inner loop in Lines 5-23 successively applies the passed in filters and alters the input token graph in case a replacement for one of its tokens has been found. Line 22 pushes the child tokens of the currently processed token onto the queue, ensuring that the whole graph is traversed.

An example use case for fuzzing filters is the boundary-value analysis software testing technique, which commonly uses the first and last values of a range of values instead of the whole range. This reduces the amount of values that need to be tested. Consider a function which has one input parameter of the type integer. The parameter's valid values range from 1 to 100, i.e., there are 100 possible test candidates and thus 100 permutations. Boundary-value analysis may reduce these permutations to 1, 50 and 100, i.e., just three instead of 100 cases are tested. Additional to the first and last values, the middlemost value. The TAVOR FRAMEWORK provides an implementation of this technique with the `PositiveBoundaryValueAnalysis` fuzzing filter. This fuzzing filter traverses the whole internal structure and replaces every range token with at most five boundary values, e.g., a range from -10 to 10 would result into -10, -1, 0, 1 and 10 therefore including the transition from negative to positive values. Another fuzzing filter, the `NegativeBoundaryValueAnalysis`, changes every range token to two integers, i.e., one representing its lower bound subtracted by one, and the other one, representing its upper bound incremented by one. Thus, resulting in values which are not valid in the original model but are useful to generate invalid data for testing. For

Listing 19 Auxiliary Function ApplyFilters

```
func ApplyFilters(filters []Filter, root Token) Token {
    queue := NewQueue()
    queue.Push(&Pair{token: root, parent: nil})

    for !queue.Empty() {
        pair := queue.Pop()

        for i := range filters {
            replacement, _ := filters[i](pair.token)

            if replacement != nil {
                if pair.parent == nil {
                    root = replacement
                } else {
                    pair.parent.Replace(pair.token, replacement)
                }

                break
            }
        }

        addChildrenToQueue(queue, pair.token)
    }

    return root
}
```

the integer range from the previous example this would mean that the two integers 0 and 101 are generated.

3.5 Reducing Strategies

A reducing strategy in the Tavor framework is a specific delta-debugging algorithm, which can be applied on the current permutation of a token graph. Individual reducing strategies may vary on the heuristic for walking through the token graph or on how the individual tokens are reduced. The reduction method is depending on the token type. For example a constant integer cannot be reduced any further but a repetition of optional strings can be minimized or even left out. All reducing strategies need to implement the following interface:

```
type Strategy func(root Token) (chan bool, chan<- Feedback, error)
```

Every reducing strategy instance has to be associated on construction with exactly one token. This allows an instance to hold a dedicated state of the given token graph, which makes optimizations for multiple reducing operations and iterations possible. During construction a reducing strategy starts its first reduction step in a new coroutine and provides the following return arguments to control the reduction process:

1. A channel to synchronize the reduction progress. The reducing strategy writes to this channel to signal that a new reduced permutation is available, i.e., the current reduction iteration is completed. The caller of the strategy function is responsible to write to this channel as soon as it is able to process the next iteration. This passing of values is needed to avoid data races within the token graph. The channel must be closed when there are no more steps or the caller of the strategy wants to end the reduction process. Note that this can also occur right after receiving the channel, i.e., when there are no reductions at all.

2. A channel to receive feedback on the current iteration. The feedback answer `Good` communicates to the reducing strategy that the current iteration produced a successful result, e.g., that the result has the same outcome when it is used or is better than the last good result. The meaning of the feedback and the response of the strategy to the feedback are purely dependent on the application. Responses could be for example to proceed with a given optimization path or to simply end the whole reduction process, since it can be enough to find just one solution. The feedback answer `Bad` communicates exactly the opposite of `Good` to the strategy. This answer is often more complicated to handle, since it means that in some scenarios a revert of the current iteration to the last good iteration has to occur before the reduction process can continue.

3. An error indicating that the initialization of the strategy has not been successful. A reducing strategy may for instance return an error, in case the input token contains loops, as they are not supported by the TAVOR FRAMEWORK.

The TAVOR FRAMEWORK offers as for the fuzzing strategies and filters a register function, which allows to register reducing strategies based on an identifier. This is especially needed for the Tavor CLI, which can execute a specific reducing strategy defined by a command line argument.

In order to provide a better understanding of the interactions between a reducing strategy and its caller, we walk through an example implementation of a fairly basic reducing strategy in Subsection 3.5.1. Additionally, we offer in Section 3.5.2 the pseudo code of the linear reducing strategy, which is provided by the TAVOR FRAMEWORK.

3.5.1 Basic Example Reducing Strategy

Consider a reducing strategy that searches the token graph for Repeat Tokens holding internally a String Token, reducing the repetition by one token for every reduction iteration. This reducing strategy is very simple and should demonstrate that reducing strategies can be added to the Tavor framework in a straight forward manner. It is a stateless strategy since every iteration can be executed independently of the previous one. Additionally, it is guaranteed to end, since only a finite amount of tokens is targeted without generating new ones.

Listing 20 shows the pseudo code for reducing repetitions of String Tokens. First the two channels for synchronization purposes are initialized. Next, a coroutine is defined in Lines 5-27, which makes use of the auxiliary function `Walk` of the Tavor framework, which receives two input parameters: the root token of the token graph that should be traversed and a function, which is called for each traversed token. This function is defined from Lines 6-24 and is responsible for the actual reduction process and synchronization with the caller. First, it checks in Line 7 whether the currently traversed token `token` is a Repeat Token referencing a String Token. If this is the case, the Repeat Token is stored in `repeat` and `ok` equals true. The loop from Lines 10-21 iterates over the number of available reductions, i.e., `repeat.Reduces()`, and tries in Line 13 to reduce the current token `repeat` by one of its referenced child tokens. This reduction is then reported in Line 16 to the caller by writing to the `continueReducing` channel. The feedback from the caller is processed in Line 17. Please note that this call blocks until the caller has written to the `feedbackReducing` channel. Finally, after the walk of the token graph has been completed the two synchronization channels are closed in Line 26.

Listing 21 shows a matching counterpart to the sample reducing strategy of Listing 20. First it initializes a concatenation list, which contains four Repeat Tokens each referencing a String Token in Lines 2-11. Next, it constructs the sample reducing strategy in Line 14. It concludes with processing the reduce steps in Lines 17-27, signaling the reducing strategy it should keep reducing until a length smaller or equal to ten is reached. When this function is executed it produces the output shown in Listing 22, which shows that the Walk function first visits the Repeat Token referencing the string "a", which is reduced until no "a" is left. Next the Repeat Token referencing the String Token "b" is processed, which is reduced until its minimum length of one is reached. This process continues until the token `doc` holds the value "bcccccccdd", which has the target length of 10.

Listing 20 Reduce String Token Repetitions

```
1  func NewSampleStrategy(root Token) (chan bool, chan<-
   ↪ ReduceFeedbackType, error) {
2      continueReducing := make(chan bool)
3      feedbackReducing := make(chan ReduceFeedbackType)
4
5      go func() {
6          err := Walk(root, func(tok Token) error {
7              repeat, ok := isRepeatingString(tok)
8              if !ok { return nil }
9
10             for i := repeat.Reduces() - 1; i >= 0; {
11                 found := false
12                 l := len(repeat.String())
13                 found, i := reduceByOne(repeat, l, i)
14                 if !found { break }
15
16                 continueReducing <- true
17                 feedback, ok := <-feedbackReducing
18                 if feedback == strategy.Good { return Done }
19                 if !ok { return nil }
20                 if _, ok := <-continueReducing; !ok { return nil }
21             }
22
23             return nil
24         })
25         if err != nil && err != Done { panic(err) }
26         close(continueReducing); close(feedbackReducing)
27     }()
28
29     return continueReducing, feedbackReducing, nil
30 }
```

Finally, the strategy can be registered as a framework-wide usable strategy using the code shown in Listing 23. This registration is, for instance, necessary in order to use the sample strategy when working with the Tavor CLI.

3.5.2 The Linear Reducing Strategy

The Linear reducing strategy reduces data based on a linear search. In contrast to the sample strategy from the previous section, this strategy does not rely on specific token types for reduction, but solely operates on the following interfaces:

Listing 21 Callee of Example Reducing Strategy

```
1  func main() {
2      aRepeat := NewRepeat(NewString("a"), 0, 100)
3      aRepeat.Permutation(6)
4      bRepeat := NewRepeat(NewString("b"), 1, 100)
5      bRepeat.Permutation(4)
6      cRepeat := NewRepeat(NewString("c"), 7, 100)
7      cRepeat.Permutation(8)
8      dRepeat := NewRepeat(NewString("d"), 1, 100)
9      dRepeat.Permutation(1)
10
11     var root Token = lists.NewConcatenation(aRepeat, bRepeat, cRepeat,
       ↪  dRepeat)
12     fmt.Println(root.String())
13
14     continueFuzzing, feedbackReducing, err := NewSampleStrategy(root)
15     if err != nil { panic(err) }
16
17     for i := range continueFuzzing {
18         out := root.String()
19         fmt.Println(out)
20
21         if len(out) <= 10 {
22             feedbackReducing <- strategy.Good
23         } else {
24             feedbackReducing <- strategy.Bad
25         }
26         continueFuzzing <- i
27     }
28 }
```

- The `Reduce Interface` is implemented by tokens which may be reduced. The method `Reduces() int` is used to determine the number of available reductions and the method `Reduce(i int)` is available to set a specific reduction.

- The `Forward Interface` is implemented by tokens which wrap other tokens. The method `Get()` is provided by this interface in order to access the wrapped token.

- The `List Interface` is implemented by tokens that wrap several tokens. The method `Len()` informs about the number of wrapped child tokens and the method `Get(i int)` returns a child specified by its index `i`.

Consider Listing 24, it shows the top level implementation of the Linear reducing strategy. It starts off by creating the two synchronization channels `contin` and `feedback`. As with the sample reducing strategy the actual reduction is performed in a coroutine.

Listing 22 Command Line Output of Example Reducing Strategy

```
1  aaaaaabbbbbcccccccccccccccdd
2  aaaaabbbbbcccccccccccccccdd
3  aaaabbbbbcccccccccccccccdd
4  aaabbbbbcccccccccccccccdd
5  aabbbbbcccccccccccccccdd
6  abbbbbcccccccccccccccdd
7  bbbbbcccccccccccccccdd
8  bbbbcccccccccccccccdd
9  bbbcccccccccccccccdd
10 bbcccccccccccccccdd
11 bcccccccccccccccdd
12 bccccccccccccccdd
13 bcccccccccccccdd
14 bccccccccccccdd
15 bcccccccccccdd
16 bccccccccccdd
17 bcccccccccdd
18 bccccccccdd
19 bcccccccdd
```

Listing 23 Registering the Sample Strategy

```
1 func init() {
2     strategy.Register("SampleStrategy", NewSampleStrategy)
3 }
```

Line 5 uses the auxiliary function `getReductionListFromTree(root)`, which traverses the token graph referenced by `root` and returns a list of tokens which can be reduced. This list of tokens is passed on to the auxiliary function `reduce(contin, feedback, list)`, which reduces these tokens and communicates over the channels `contin` and `feedback` with the caller of the Linear reducing strategy.

The pseudo code of the auxiliary function `getReductionListFromTree` is shown in Listing 25. It uses a queue to traverse the token graph rooted in the token `root`. The loop starting in Line 7 searches for tokens implementing the `Reduce Interface` and adds them to the list of reducible tokens. Tokens wrapping child tokens are dealt with in Lines 9-14 by unwrapping them and adding them to the queue for later procession.

Listing 26 shows the pseudo code of an excerpt of the `reduce` method. It walks through the tokens stored in `list` and tries to find a suitable reduction of the current token in the loop from Lines 3-12. The auxiliary function `nextStep` handles the synchronization with the reducing strategy caller via the channels `contin` and `feedback`, which is done along the same lines as for the sample reducing strategy in the previous subsection.

Listing 24 Linear Reducing Strategy

```
1  func NewLinear(root token.Token) (chan bool, chan<-
   ↪ ReduceFeedbackType, error) {
2      contin := make(chan bool)
3      feedback := make(chan ReduceFeedbackType)
4      go func() {
5          list := getReductionListFromTree(root)
6          if !reduce(contin, feedback, list) {
7              return
8          }
9          close(contin)
10         close(feedback)
11     }()
12     return contin, feedback, nil
13 }
```

Please note that `c.Reduce(c.Reduces()-1)` is always equal to the initial value of `c`, i.e., if no suitable reduction is found the last iteration of the inner loop performs the restoration of the initial value of `c`. Finally the switch-statement in Line 13 is responsible for recursively calling the `reduce` method to Forward and List tokens. Consider, for instance, a token graph which has as its root a Repeat Token referencing another reducible token. The root Repeat Token is reducible, hence it will be in the list of reducible tokens during the first call of method `reduce`. Its referenced child is not part of that list, hence a recursive call of method `reduce` is necessary to also process this token.

Listing 25 Traversing of the Token Graph

```
1  func getReductionListFromTree(root Token) (list []ReduceToken) {
2      queue := NewQueue()
3      queue.Push(root)
4      for !queue.Empty() {
5          token := queue.Pop()
6          switch t := token.(type) {
7          case ReduceToken:
8              list = append(list, token)
9          case ForwardToken:
10             queue.Push(t.Get())
11         case ListToken:
12             for i := t.Len() - 1; i >= 0; i-- {
13                 queue.Push(t.Get(i))
14             }
15         }
16     }
17     return list
18 }
```

Listing 26 Performing the Reduction

```
1  func reduce(contin chan bool, feedback <-chan ReduceFeedbackType, tree
   ↪  []ReduceToken) bool {
2      for _, c := range tree {
3          for i := 0; i < c.Reduces(); i++ {
4              c.Reduce(i)
5              contin, feedback := nextStep(contin, feedback)
6              if !contin {
7                  return false
8              } else if feedback == Good {
9                  break
10             }
11             c.reduction++
12         }
13         switch t := c.(type) {
14         case ForwardToken:
15             children := getReductionListFromTree(c.Get())
16             reduce(contin, feedback, children)
17         case ListToken:
18             for i := t.Len() - 1; i >= 0; i-- {
19                 children := getReductionListFromTree(c.Get(i))
20                 reduce(contin, feedback, children)
21             }
22         }
23     }
24     return true
25 }
```

Chapter 4

Tavor Format

The TAVOR FORMAT is an EBNF-like notation which allows the definition of data, such as file formats and protocols, without the need of programming. It is the default format of the Tavor CLI of Chapter 5 and supports every feature of the framework, which has been introduced in Chapter 3. Each of the demonstrated features of the TAVOR FORMAT is required to fully define the AIGER format.

The format is Unicode text encoded in UTF-8 and consists of terminal and non-terminal symbols which are called `tokens` throughout the TAVOR FRAMEWORK. A more general definition of `tokens` can be found in Section 3.2.

4.1 Token Definition

Every `token` in the format belongs to a definition of `non-terminal token` which consists of a unique case-sensitive name and its definition part. Both are separated by exactly one equal sign. Syntactical white spaces are ignored. Every `token definition` is by default declared in one line. A line ends with a new-line character.

To give an example, the format in Listing 27 declares the `START token` with the `String Token` "Hello World" as its definition.

Listing 27 TAVOR's Hello World

```
1 START = "Hello World"
```

`Token names` have the following constraints: Each `token name` has to start with a letter, and can only consist of letters, digits and the underscore sign "_". Additionally, a `token name` has to be unique. It is also not allowed to declare a `token` and never use it. The `START token` is the only exception, which is used as the entry point of the

format. Hence it defines the beginning of the format and is therefore required for every format definition.

Sequential listed `tokens` in the definition part of a `token definition` are automatically concatenated. The example in Listing 28 concatenates to the string “This is a String Token and this 123 was a number token.”.

Listing 28 Example for a Token Concatenation

```
1 START = "This is a String Token and this " 123 " was a number token."
```

A `token definition` can be sometimes too long or poorly readable, it can be therefore split into multiple lines by using a comma before the newline character as shown in Listing 29. The `token definition` ends at the string “definition.” since there is no comma before the newline character. Listing 29 also highlights that syntactical white spaces are ignored and can be used to make a format definition more human-readable.

Listing 29 Example for a Multi-Line Token Definition

```
1 START = "This",
2         "is",
3         "a",
4         "multi line",
5         "definition."
```

The TAVOR FORMAT supports two kinds of comments. `Line comments` start with the character sequence “//” and end at the next new-line character. `General comments` start with the character sequence “/*” and end at the character sequence “*/”. A general comment can therefore contain new-line characters and can be used between `token definition` and `tokens`. Both kinds of comments are illustrated in Listing 30.

Listing 30 Example for Different Kinds of Comments

```
 1 /*
 2
 3 This is a general comment
 4 which can have
 5 multiple lines.
 6
 7 */
 8
 9 START = "This is a string." // This is a line comment.
10
11 // This is also a line comment.
```

4.2 Terminal Tokens

`Terminal tokens` are the constants of the Tavor format. The format supports two kinds of `terminal tokens`: numbers and strings. Every other token that is not a `terminal token`, such as `tokens` of definitions, is called a `non-terminal token`.

`Number tokens` allow only positive decimal integers, which are written as a sequence of digits as shown in Listing 31.

Listing 31 Example for a Number Token

```
1 START = 123
```

`String Tokens` are character sequences between double quote characters and can consist of any UTF-8 encoded character except the new-line, the double quote and the backslash characters which have to be escaped with a backslash character. An example can be seen in Listing 32.

Listing 32 Example for a String Token

```
1 START = "The next word is \"quoted\" and next is a new line\n"
```

Since Tavor is using Go's text parser as foundation of its format parsing, the same rules for "interpreted string literals" as defined in Go's language specification [5] apply to `String Tokens`.

Empty `String Token` are forbidden and lead to a format parsing error. The reason for this exceptions is the way `tokens` are utilized during parsing and delta-debugging, and is described in the repeat groups of Paragraph 4.5 in more detail.

4.3 Embedding of Tokens

`Non-terminal tokens` can be embedded in the definition part by using the name of the referenced `token`. The example in Listing 33 embeds the `token` "Text" into the `START token`. `Token names` declared in the global scope of a format definition can be used throughout the format regardless of their declaration position. Terminal and non-terminal tokens can be mixed as illustrated in Listing 34.

The Tavor framework and therefore the Tavor format differentiate between a `token reference`, which is the embedding of a `token` in a definition, and a `token`

Listing 33 Example for Embedding Tokens

```
1 START = Text
2
3 Text = "Some text"
```

Listing 34 Example for Terminal and Non-Terminal Tokens Mixed in One Format

```
1 Dot = "."
2
3 First  = 1 Dot
4 Second = 2 Dot
5 Third  = 3 Dot
6
7 START = First ", " Second " and " Third
```

`usage`, which is the execution of a `token` during an operation such as fuzzing or delta-debugging. Listing 35 illustrates the difference between a `token reference` and a `token usage`. The format defines two `tokens` called "Choice" and "List". There exists one `token reference` of "Choice", which can be found in the "List" definition, and two for "List", which are both in the `START token` definition. Although "Choice" is in a `repeat group`, which in this example repeats the token "Choice" exactly twice, it is only referenced once. "List" has two `token usages` in this format while "Choice" has 4. Every "List" `token` does have two "Choice" usages because of the `repeat group` in the definition of "List".

Listing 35 Example for a Token Reference and a Token Usage

```
1 Choice = "a" | "b" | "c"
2
3 List = +2(Choice)
4
5 START = "1. list: " List "\n",
6         "2. list: " List "\n"
```

4.4 Alternations

`Alternations` are defined by the pipe character "|" which separates two `alternation terms`. The example in Listing 36 defines that the token `START` can either hold the strings "a", "b" or "c". An `alternation term` has its own scope which means that a sequence of `tokens` can be used.

Listing 36 Example for the Alternation Token

```
1 START = "a" | "b" | "c"
```

`Alternation terms` can be empty which allows more advanced definitions of formats. Listing 37 illustrates how to use an alternation to define a loop which can either hold the empty string or the strings “a”, “b”, “ab”, “aab” or any amount of “a” characters ending with an optional “b” character.

Listing 37 Example for Loop Using an Alternation

```
1 A = "a" A | B |
2 B = "b"
3
4 START = A
```

4.5 Groups

`Tokens` can be arranged using `token groups` by using parenthesis beginning with the opening parenthesis character “(” and ending with the closing parenthesis character “)”. A `group` is a `token` on its own and can be therefore mixed with other `tokens`. Additionally, a `group` starts a new scope between its parenthesis and can therefore hold a sequence of `tokens`. The `tokens` between the parenthesis are called the `group body`. Listing 38 illustrates the usage of a `token group` by declaring that the `START token` can either hold the string “a c” or “d c”. `Groups` can be nested as illustrated in Listing 39, which defines a number with one to three digits. `Groups` can have modifiers which give a `group` additional abilities. The following sections introduces these modifiers.

Listing 38 Example for a Group Token

```
1 START = ("a" | "d") " c"
```

Optional group

The `optional group` has the question mark “?” as modifier and allows the whole `group token` to be optional. The `START token` in Listing 40 can either hold the strings ”a” or ”a b”.

Listing 39 Example for Nested Groups

```
1 Digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
2
3 START = Digit (Digit (Digit | ) | )
```

Listing 40 Example for an Optional Group

```
1 START = "a" ?(" b")
```

Repeat group

The default modifier for the `repeat group` is the plus character "+". The repetition is executed by default at least once. In Listing 41 the string "a" is repeated and the `START token` can therefore hold any amount of "a" characters but at least one.

Listing 41 Example for a Repeat Group

```
1 START = +("a")
```

Since repeated empty `tokens` lead to infinite steps during parsing and delta-debugging, a parsing error is issued if such `tokens` are encountered. This includes `optional groups` or `alternations` with at least one empty term inside a `Repeat Token`. The `repeat group` repeats by default from one to infinite times. The repetition can be altered with arguments to the modifier. Listing 42 repeats the string "a" exactly twice. The `START token` can therefore only hold the string "aa". It is also possible to define a repetition range. Listing 43 repeats the string "a" at least twice but at most four times. This means that the `START token` can either hold the strings "aa", "aaa" or "aaaa". The "from" and "to" arguments can be empty too. They are then set to their default values. Listing 44 repeats the string "a" at least once and at most four times. Listing 45 repeats the string "a" at least twice. Listing 46 illustrates the `optional repeat group` modifier. The `START token` can either hold the strings "a", "ab", "abb" or any amount of "b" characters prefixed by an "a" character.

Permutation group

The "@" is the permutation modifier which is combined with an alternation in the group body. Each alternation term will be executed exactly once but the order of execution is non-relevant. In Listing 47 the `START token` can either hold "123", "132", "213", "231", "312" or "321".

Listing 42 Example for a Fixed Repeat Group

```
1 START = +2("a")
```

Listing 43 Example for a Ranged Repeat Group

```
1 START = +2,4("a")
```

4.6 Character Classes

`Character class tokens` can be directly compared to character classes of regular expressions used in most programming languages. A `character class token` begins with the left bracket "[" and ends with the right bracket "]". The content between the brackets is called a pattern and can consists of almost any UTF-8 encoded character, escape character, special escape character and range. In general the `character class token` can be seen as a shortcut for an `alternation`. The definition in Listing 48 illustrates the usage of a `character class` by defining that the `START token` holds either the strings "a", "b" or "c".

Escape Characters

Table 4.1 holds all `escape characters` which are UTF-8 encoded characters that are not directly allowed within a `character class` pattern. Their equivalent escape sequence has to be used instead. Listing 49 defines that the `START token` can hold only white space characters.

Table 4.1: Escape Characters for Character Classes

Character	Escape sequence
"-"	"\-"
"\"	"\\"
form feed	"\f"
newline	"\n"
return	"\r"
tab	"\t"

Since some characters can be hard to type and read the "\x" escape sequence can be used to define them with their hexadecimal code points. Either only two hexadecimal characters are used in the form of "\x0A" or when more then two hexadecimal digits

Listing 44 Example for an Empty "from" Argument for a Ranged Repeat Group

```
1 START = +,4("a")
```

Listing 45 Example for an Empty "to" Argument for a Ranged Repeat Group

```
1 START = +2,("a")
```

are needed the form "\x{0AF}" has to be used. The second form allows up to eight digits and is therefore fully Unicode ready. Unicode usage is illustrated in Listing 50, which either holds the Unicode character "/" or "☺".

Character Class Ranges

`Character class ranges` can be defined using the "-" character. A `range` holds all characters defined by UTF-8 starting from the character before the "-" to ending at the character after the "-". Both characters have to be either an UTF-8 encoded or an escaped character. The starting character must have a lower value than the ending character. The usage of `Character classes` is illustrated in Listing 51 which defines a number with one digit.

Special Escape Characters

`Special escape characters` combine many characters into one `escape character`. Table 4.2 lists all implemented `special escape characters`.

Table 4.2: Special Escape Characters for Character Classes

Escape character	**Character class**	**Description**
"\d"	"[0-9]"	Holds a decimal digit character
"\s"	"[\f\n\r\t]"	Holds a white space character
"\w"	"[$a-zA-Z0-9$_]"	Holds a word character

4.7 Token Attributes

Some `tokens` define attributes which can be used in a definition by prefixing the dollar sign "$" to their name and appending a dot followed by the attribute name. Listing 52

Listing 46 Example for an Optional Repeat Group

```
1 START = "a" *("b")
```

Listing 47 Example for a Permutation Group

```
1 START = @(1 | 2 | 3)
```

illustrates `token attributes` using the "Count" attribute, which holds the count of the token's direct children. The format holds for example the string "The number 56 has 2 digits".

Some `token attributes` can have arguments. A `token argument list` begins with the opening parenthesis "(" and ends with the closing parenthesis ")", and holds "token argument parameters" which are separated by commas. All `list tokens` have for example the "Item" attribute, which holds one child of the token. The "Item" attribute has one argument which is the index of the child starting from the index zero, and is illustrated in Listing 53. The format holds the string "The letter with the index 1 is b".

A `list token` is a `token` which has in its definition either only a sequence of `tokens` or exactly one `repeat group token`. Table 4.3 shows the `token attributes` that can be utilized by `list tokens`.

Table 4.3: Token Attributes for List Tokens

Attribute	Arguments	Description
"Count"	-	Holds the count of the token's direct children.
"Item"	"i"	Holds a direct child of the token with the index "i".
"Unique"	-	Chooses at random a direct child of the token and embeds it. The choice is unique for every reference of the token.

Scope of Attributes

The Tavor format allows the usage of `token attributes` as long as the referenced `token` exists in the current scope.

Two kinds of scopes exist: The `global scope` is the scope of the whole format definition. An entry of the `global scope` is set by the nearest child `token reference` to the

Listing 48 Example for a Character Class Token

```
1 START = [abc]
```

Listing 49 Example for Escape Characters in Character Classes

```
1 START = +([ \t\n\r\f])
```

`START token`. The `local scope` is the scope held by a definition, `group` or any other `token` which holds its own scope. `Local scopes` are initialized with entries from their parent scope at the time of the creation of the new `local scope`. Listing 54 illustrates this behavior which can for example hold the string "1 a 1(1 aa 2)1 aaa 3". The `List` token is used trice in this example. The first usage leads to the value "a", the second to "aa" and the third to "aaa". Depending on which list is visible in the current scope "$List.Count" results in another value. This example showcases that the scope of token `Inner` is not visible in the scope of the `START token`.

4.8 Typed Tokens

`Typed tokens` are a functional addition to regular `token definitions` of the TAVOR FORMAT. They provide specific functionality which can be utilized by embedding them like regular `tokens` or through their additional `token attributes`. `Typed tokens` can be defined by prefixing the dollar sign "$" to their name. They do not have a format definition on their right-hand side. Instead, a type and optional arguments written as key-value pairs, which are separated by a colon, define the `token`. Listing 55 illustrates `typed tokens` with the definition of an `integer token`. The format definitions holds additions with random integers as numbers such as "47245 + 6160 + 6137".

The range of the "Int" type can be bounded using arguments for the definition as shown in Listing 56, where the `token` "Number" is an integer within the range one to ten.

Typed Token `Int`

The `Int` type implements a random integer. Its optional `token arguments` are listed in Table 4.4 and its `token attributes` are listed in Table 4.5.

Listing 50 Example for Hexadecimal Code Points in Character Classes

```
1 START = [\x2F\x{263A}]
```

Listing 51 Example for a Character Class Range

```
1 START = [0-9]
```

Table 4.4: Optional Token Arguments for the "Int" Typed Token

Arguments	Description
"from"	First integer value (defaults to 0)
"to"	Last integer value (defaults to $2^{31} - 1$)

Table 4.5: Token Attributes for the "Int" Typed Token

Attribute	Arguments	Description
"Value"	-	Embeds a new token based on its parent

Typed Token `Sequence`

The `Sequence` type implements a generator for integers. Its optional `token arguments` are listed in Table 4.6 and its `token attributes` are listed in Table 4.7.

Table 4.6: Optional Token Arguments for the "Sequence" Typed Token

Arguments	Description
"start"	First sequence value (defaults to 1)
"step"	Increment of the sequence (defaults to 1)

Listing 52 Example for a Token Attribute Using the "Count" Attribute

```
1 Number = +([0-9])
2 START = "The number " Number " has " $Number.Count " digits"
```

Listing 53 Example for Attribute Parameters Using the "Item" Attribute

```
1 Letters = "a" "b" "c"
2 START = "The letter with the index 1 is " $Letters.Item(1)
```

Table 4.7: Token Attributes for the "Sequence" Typed Token

Attribute	Arguments	Description
"Existing"	-	Embeds a new token holding one existing value of the sequence
"Next"	-	Embeds a new token holding the next value of the sequence
"Reset"	-	Embeds a new token which on execution resets the sequence

4.9 Expressions

`Expressions` can be used in `token definitions` and allow dynamic and complex operations using `operators` who can have different numbers of `operands`. An `expressions` starts with the dollar sign "$" and the opening curly brace "{" and ends with the closing curly brace "}". Listing 57 illustrates the simplest expression which is an addition.

Every `operand` of an `operator` can be a token. The usual dollar sign for a `token attribute` can be omitted inside an `expression` as illustrated in Listing 58.

Arithmetic Operators

`Arithmetic operators` which are shown in Table 4.8 have two `operands` between the `operator` sign. `Operators` always embed the right side `operand` which means that that "2 * 3 + 4" will result in "2 * (3 + 4)" and not "(2 * 3) + 4".

Listing 54 Example for Scopes

```
1 List = +,10("a")
2
3 Inner = "(" $List.Count " " List " " $List.Count ")"
4
5 START = $List.Count " " List " " $List.Count,
6         Inner,
7         $List.Count " " List " " $List.Count
```

Listing 55 Example Typed Token Using an Integer Token

```
1 $Number Int
2
3 Addition = Number " + " (Number | Addition)
4
5 START = Addition
```

Listing 56 Example for Typed Token Attributes

```
1 $Number Int = from: 1,
2               to:   10
3
4 Addition = Number " + " (Number | Addition)
5
6 START = Addition
```

Listing 57 Example Expression

```
1 START = ${1 + 2}
```

Listing 58 Example for a Token Attribute Inside an Expression

```
1 $Number Int
2
3 START = ${Number.Value + 1}
```

Table 4.8: Arithmetic Operators

Operator	Description
"+"	Addition
"-"	Subtraction
"*"	Multiplication
"/"	Division

Include Operator

The `include operator` parses an external TAVOR FORMAT file and includes its `START token`. It takes a string as its only `operand` which defines the file path of the to be included TAVOR FORMAT file. The file path can be absolute or relative. Listing 59 includes the TAVOR FORMAT file "number.tavor".

Listing 59 Example for Include Operator

```
1 START = ${include "number.tavor"}
```

Path Operator

The `path operator` traverses a `list token` based on the described structure. The structure defines the starting value of the traversal, the value which identifies each entry of the `list token`, how the entries are connected and which values are ending values for the traversal. The `path operator` has the format "path from (<starting value>) over (<entry identifier>) connected by (<entry connections>) without(<ending values>)". All values are `expressions`. Furthermore, the `entry connections` and `ending values` are lists of `expressions`. The `entry identifier`, `entry connections` and `ending values` have the variable "e" in their scope which holds the currently traversed entry of the `list token`.

Listing 60 defines a list of connections called "Pairs". Each entry in the list "Pairs" defines the identifier as its first `token` and the connection as its second `token`. The used `path operator` arguments define that all entries are traversed beginning from the value "2" and ending at the value "0". The example holds the string "103123->231".

Listing 60 Example for Path Operator

```
1 START = Pairs "->" Path
2
3 Path = ${Pairs path from (2) over (e.Item(0)) connect by (e.Item(1))
   ↪  without (0)}
4
5 Pairs = (,
6         (1 0),
7         (3 1),
8         (2 3),
9 )
```

Not-in Operator

The `not-in operator` queries the "Existing" `token attribute` of a sequence to not include the given list of `expressions`. A list of `expressions` begins with the opening parenthesis "(" and ends with the closing parenthesis ")". Each `expression` is defined without the expression frame "${...}". `Expressions` are separated by a comma. Listing 61 illustrates the `not-in operator` by issuing two sequence entries and excluding the entry with the value "2". The format therefore holds the string "1, 2 -> 1".

Listing 61 Example for the Not in Operator

```
1 $Id Sequence
2
3 START = $Id.Next ", " $Id.Next " -> " ${Id.Existing not in (2)}
```

4.10 Variables

Every `token` of a `token definition` can be saved into a `variable` which consists of a name and a reference to a `token usage`. `Variables` follow the same scope rules as `token attributes`. It is therefore possible to, for example, define the same `variable name` more than once in one `token sequence`. They also do not overwrite `variable definitions` of parent scopes. `Variables` can be defined by using the "<" character after the `token` that should be saved, then defining the name of the `variable` and closing with the ">" character. `Variables` have a range of `token attributes` which are listed in Table 4.9. Listing 62 illustrates the usage of `variables` by saving the `String Token` "text" into the `variable` "var". The `token` "Print" uses this `variable` by embedding the referenced `token`. The format therefore holds the string "text->text". `Tokens` which are saved to `variables` are by default relayed to the generation. This

Table 4.10: Operators for the If Statement Condition

Operator	Usage	Description
“==”	“<token> == <token>”	Returns true if the value of op1 is equal to op2
“defined”	“defined <name>”	Returns true if name is a defined variable

means that their usage generates data as usual. Since this is sometimes unwanted, an equal sign “=” after the “<” character can be used to omit the relay.

Listing 62 Example for a Token Variable

```
1 START = "text"<var> "->" Print
2
3 Print = $var.Value
```

Table 4.9: Token Attributes of Variable Tokens

Attribute	Arguments	Description
“Count”	-	Holds the count of the referenced token’s direct child
“Index”	-	Holds the index of the referenced token in relation to its parent
“Item”	“i”	Holds a child of the referenced token with the index “i”
“Reference”	-	Holds a reference to a token which is needed for some operators
“Value”	-	Embeds a new token based on the referenced token

4.11 Statements

`Statements` allow the TAVOR FORMAT to have a control flow in its `token definitions` depending on the used tokens and values. All `statements` start with the opening curly brace “{” and end with closing curly brace “}”. The `statement operator` must be defined right after the closing curly brace.

The `if statement` allows to embed conditions into `token definitions` and defines an `if body` which is a scope on its own. The `token body` lies between an opening `{if <condition>} statement` and an ending `{endif} statement`. The condition can be formed using the `if statement’s operators` which are listed in Table 4.10. Each `if body` creates a new scope. Listing 63 illustrates the `if statement` by generating the character “A” only if the variable “var” is equal to “1”.

Listing 63 Example for the If Statement

```
1 Choose = 1 | 2 | 3
2
3 START = Choose<var> "->" {if var.Value == 1}"A"{endif}
```

Additional to the `if statement`, the statements `else` and `else if` can be used. They can only be defined inside an `if body` and always belong to the `if statement` that the body belongs to. Both statement operators create a new `if body`.

Chapter 5

Tavor CLI

The TAVOR CLI is the command line interface (CLI) for the TAVOR FRAMEWORK. It is therefore the user interface for non-programmers to utilize the capabilities of the framework for fuzzing and delta-debugging. Commands and arguments have to be specified for the binary using the following format:

```
<general arguments> <command> <command arguments>
```

General arguments, which are listed in detail in Listing 85 of Appendix B, are mostly just activating the output of textual logs for informing the user or to set global constants. Arguments are applied during initialization or the invocation of a CLI command. Examples for global constants are the `--seed` argument, which sets the seed for the random generator of the CLI to make all executions deterministic, and the `--max-repeat` argument, which sets the maximum repetition for unrolling loops. The CLI acts on a TAVOR FORMAT file which is specified by the `--format-file` argument.

The TAVOR CLI's main purpose is the invocation of the following CLI commands:

- The `graph` command converts the given format file into a graphical representation.
- The `fuzz` command produces fuzzed outputs from the given format file.
- The `validate` command applies the given format file to a specified input file.
- The `reduce` command applies delta-debugging for the given format file to a specified input.

The following sections describe these commands, their arguments, workflow and usage in more detail.

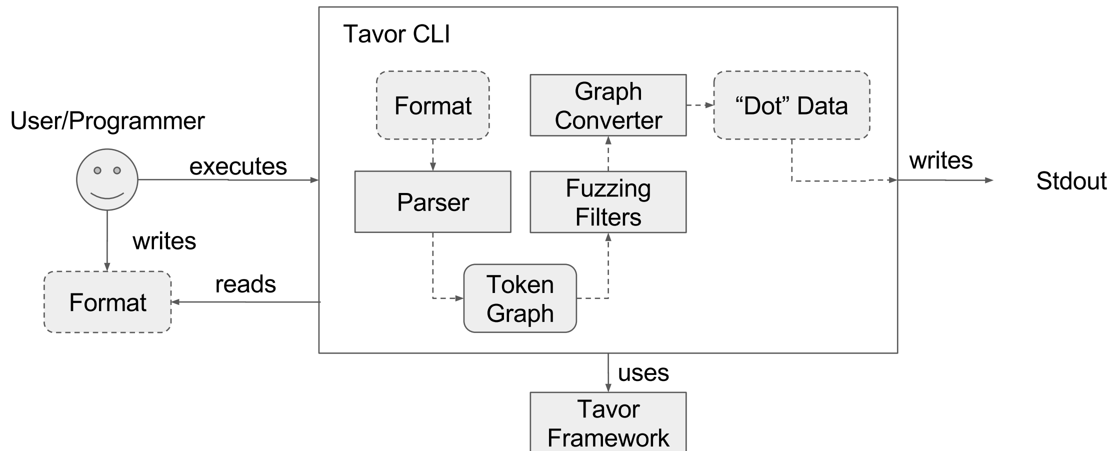


Figure 5.1: Workflow for the **Graph** Command of the TAVOR CLI

5.1 Command `graph`

The `graph` command of the TAVOR CLI converts a given TAVOR FORMAT file to a graphical representation of a finite-state machine (FSM). This functionality is needed because textual representations of graphs, such as the TAVOR FORMAT, are often too difficult to mentally visualize. Listing 90 of Appendix B presents the available arguments for this command. Most notable is the `--filter` argument which allows to define `fuzzing filters` before the graphical conversion is executed. The command outputs the textual DOT graph description language [1], which can be further processed into image formats, e.g., by using the Graphviz toolset [2].

Figure 5.1 illustrates the workflow and the interactions of components for the `graph` command. First the user-specified TAVOR FORMAT file is read in from the file system and parsed using the format parser of the framework, which results into a `token graph`. Then optional `fuzzing filters` are applied to the graph. As last step in the workflow the graph converter component of the framework is used, to convert the `token graph` to textual DOT formated data for the FSM which is streamed to the STDOUT file descriptor of the CLI.

The algorithm for the conversion of the `token graph` to DOT data is composed by the following two phases:

[1] http://www.graphviz.org/content/dot-language
[2] http://www.graphviz.org

Listing 64 Convert Token Graph to Simple Graph Structure

```
1  func buildGraph(token, graph) start, end {
2    start, end = {}, {}
3    switch type(token) {
4      Optional:
5        token' = token.Child()
6        start', end' = buildGraph(token', graph)
7
8        start.AddOptional(start')
9        end.AddOptional(end')
10     Concatenation:
11       prev = {}
12       for i = 0; i < token.NumChildren; i++ {
13         token' = token.Child(i)
14         start', end' = buildGraph(token', graph)
15
16         graph.AddEdges(prev, start')
17
18         if start'.ContainsOptional() {
19           start'.Add(prev)
20         }
21         prev = start'
22
23         if i == 0 {
24           start = start'
25         } else if i == token.NumChildren-1 {
26           end = end'
27         }
28       }
29     Scope:
30       token' = token.Child()
31       start' end = buildGraph(token', graph)
32     String:
33       graph.AddState(token)
34
35       start = {token}
36       end = {token}
37   }
38   return start, end
39 }
```

- Phase one traverses the *token graph* using a recursive depth-first search, which builds upon the assumption, that the used data structure does not contain any loops. As mentioned in Subsection 3.2.2, a loop unrolling step is performed after the format has been parsed to make sure that there are no loops in the token graph. Since each *token type* has its own representation and internal data structure, each type has to be treated differently for the graphical representation using its own implementation. The pseudo code for handling *Optional*, *Concatenation*, *Scope* and *String Tokens* is shown in Listing 64. The traversal function receives the current token to process as well as the graph data structure which needs to be extended. As return values it delivers the set of start and end states of the extended graph data structure. When traversing an Optional Token, the returned start and end states need to be marked as optional. In case of processing a Concatenation Token, edges need to be added to the graph data structure which connect the individual children of the Concatenation Token. The returned start (resp. end) states are the start (resp. end) states of the first (resp. last) child of the Concatenation Token. When traversing a Scope Token a call to process its only child is performed. Whenever a String Token is processed, a state is added to the resulting graph. As String Tokens are leaves of the token graph no further recursive calls are necessary.

- Phase two uses the extended graph structure as well as the returned start and end states of phase one to print a DOT representation of the input token graph. End states are depicted using double lines around their label. Optional vertices are shown by using dashed incoming and outgoing arrows. Additional labels and states, which are not included in the pseudo code, are used to depict other tokens such as repeating groups and ranges.

The TAVOR FORMAT in Listing 65 exemplifies how the `graph` command works. The format results in the DOT data depicted in Listing 87 of Appendix B, which is convertible into the FSM illustrated in Figure 5.2. All tokens of the format, which are depicted in the graphics as states, are sequential but the tokens B and F are optional, and the group DE is repeated at least twice but at most four times. States are connected using different types of edges, which are depicted as differently styled arrows in the graphics. The FSM depicted in Figure 5.2 starts with the small dot at the top of the graphics. Since B is optional, its incoming and outgoing arrows are dotted. In contrast, the arrow from A to C is solid and is therefore mandatory. However, since A has two outgoing arrows, only one of them has to be taken. Figure 5.2 also illustrates repetition of the DE group, which is indicated by the small dot with an ingoing arrow that has a label with the repetition amount. This arrow causes a loop, therefore marking the repetition. Finally, the accepting state after the repetition has a double bordered circle. Each path of a graph must end in such an accepting state, or else it is not a valid path, i.e., it

Listing 65 Example TAVOR FORMAT for the `Graph` Command

```
1  START = A ?(B) C +2,4(D E) ?(F)
2
3  A = "a"
4  B = "b"
5  C = "c"
6  D = "d"
7  E = "e"
8  F = "f"
```

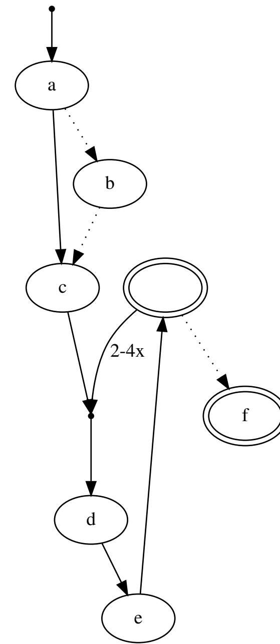


Figure 5.2: Example Graphics for the `Graph` Command

is not considered by the fuzzing and delta-debugging processes. In this example, it is also possible to go from the accepting state after the repetition to the *F* token which is optional but also ends in an accepting state.

5.2 Command `fuzz`

The `fuzz` command of the TAVOR CLI generates permutations, which conform to a specified format. Additionally, it is capable of forwarding these permutations to external programs, thus allowing to systematically execute a system under test with a stream of inputs for some predefined structure.

Figure 5.3 illustrates the workflow and the interactions of components for the `fuzz` command. It starts along the same lines as the `graph` command, by first parsing the TAVOR FORMAT file into a `token graph` and by applying optional `fuzzing filters` on that graph. It then proceeds with its fuzzing loop, which uses a specified `fuzzing strategy` to generate consecutive permutations. These permutations are written either to the file system or the STDOUT file descriptor. External programs may process these permutations and in turn interact with the TAVOR CLI using their exit codes or via STDIN.

The fuzzing loop of the `fuzz` command is highly configurable via command line options, which are listed in Listing 88 of Appendix B. It may either interact with executables

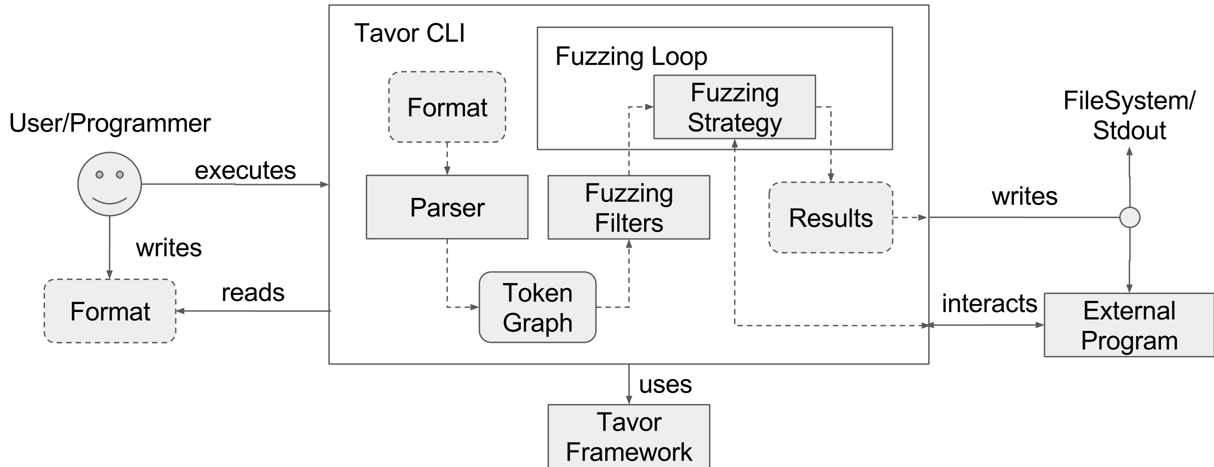
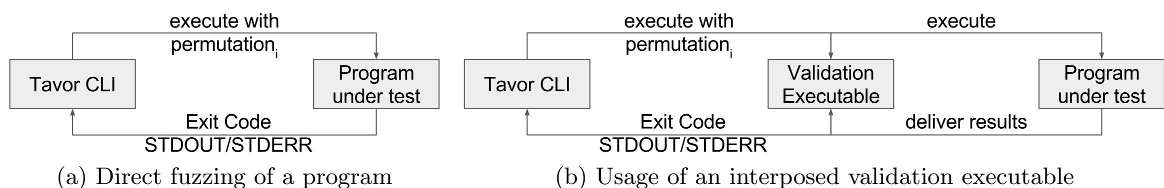


Figure 5.3: Workflow for the `Fuzz` Command of the TAVOR CLI

or scripts. When working with executables their exit codes as well as their STDOUT and STDERR file descriptors are processed by the fuzzing loop. In case the `--exit-on-error` option is set the fuzzing loop will terminate in case an unexpected output is encountered in any of the afore mentioned communication channels. When using scripts the communication with the fuzzing loop is solely performed via STDIN and STDOUT. The fuzzing loop writes permutations separated by a predefined separator to STDIN of the script and waits until the script signals via its STDOUT that it has processed the current permutation. Success or failure, are signaled by using the constants "Yes" and "No".

When using the TAVOR CLI to immediately fuzz an external program there are two ways to proceed, which are depicted in Figure 5.4. Either the program under test is directly passed on to the TAVOR CLI, or an interposed validation executable is used. To communicate directly with the program under test has the advantage that only the expected input format is required to start fuzzing. But also has the downside that the built-in validating capabilities of the TAVOR CLI are restricted to specifying return codes as well as regular expressions on delivered outputs in STDOUT and STDERR. In case a more thorough validation is required, which cannot be accomplished by the afore mentioned validation capabilities, we advise to use interposed executables or scripts. Consider for instance a program which sorts CSV files by specific columns. An interposed validation function or script could not only check that the program under test



(a) Direct fuzzing of a program (b) Usage of an interposed validation executable

Figure 5.4: Fuzzing of External Programs

exits without errors, but could additionally verify that the output CSV is indeed sorted and its content corresponds to the input CSV file. Several examples of customized scripts and executables are provided at [3].

5.3 Command validate

The validate command of the Tavor CLI checks that a given input file conforms to a specified format file. This functionality can be helpful since for instance the reduce command only applies delta-debugging on valid inputs, or in the general case it can be used to examine an input, which was not generated through the given format file.

Figure 5.5 illustrates the workflow and the interactions of components for the validate command. It starts off by parsing the specified Tavor format file to its corresponding token graph. In the next step this token graph is used to parse the specified input file. In case the input file can be parsed successfully, it conforms to the specified format file. The CLI exits with the exit status 0 if the input file conforms to the format file, or exits with an exit status unequal to 0 if it does not conform.

Please refer to Listing 86 of Appendix B for a list of available arguments of the validate command.

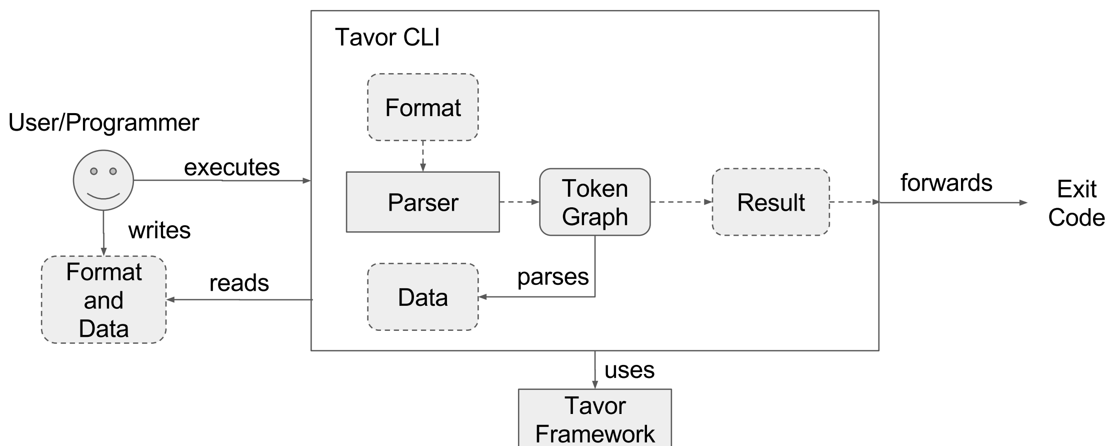


Figure 5.5: Workflow for the Validate Command of the Tavor CLI

5.4 Command reduce

The reduce command applies delta-debugging to a given input according to a specified Tavor format file, i.e., given an input which results in certain program behavior, this

[3] https://github.com/zimmski/tavor/tree/master/examples/fuzzing

capability may be used to systematically reduce the given input while preserving the same behavior.

Figure 5.6 depicts the workflow and the interactions of components for the `reduce` command. It starts similar to the `validate` command by first parsing the TAVOR FORMAT file to its associated token graph. This graph is then used to parse the given input data, i.e., the initial input needs to correspond to the specified format. Next, the reduction loop starts by feeding the initial input data to an external resource and capturing its responses. These initial responses are consecutively used by the reduction loop to guide the generation of reduced inputs, i.e., only reductions resulting in these responses are investigated for further reductions.

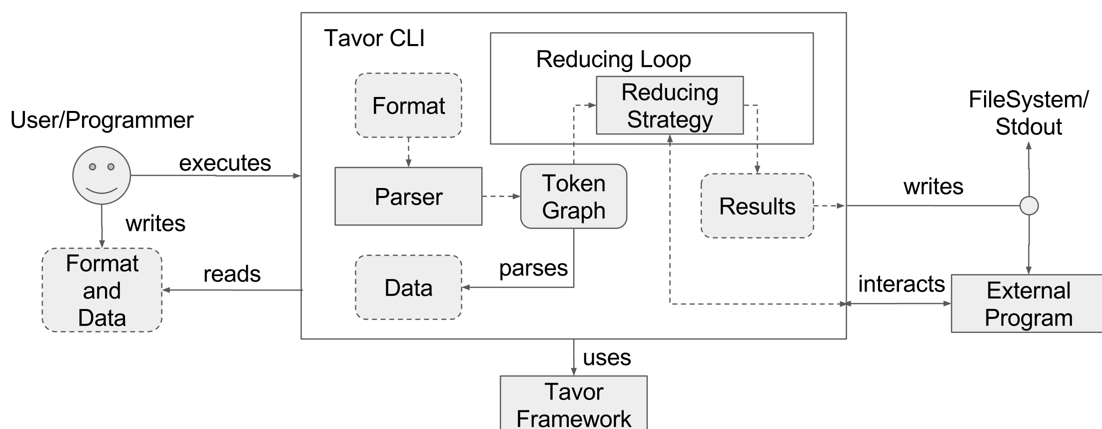


Figure 5.6: Workflow for the `Reduce` Command of the TAVOR CLI

Various options are available for the `reduce` command, which are listed in Listing 89 of Appendix B. The `reduce` command works along the same lines as the `fuzz` command, i.e., it may interact with scripts or executables using exit codes as well as STDERR, STDOUT and STDIN as communication channels. Options such as `--exec-match-stderr` allow to define the expected structure of outputs written to those channels. Please refer to [4] for example scripts and executables for performing delta-debugging.

[4] https://github.com/zimmski/tavor/tree/master/examples/deltadebugging

Listing 66 Delta-Debugging Pseudo Code for Scripts

```
1  func deltaDebugScript(tokenGraph, reduceStrategy) {
2    continue, feedback = reduceStrategy(tokenGraph)
3
4    for i = range continue {
5      stdin.Write(tokenGraph.String())
6      stdin.Write(inputSeparator)
7
8      result = stdout.ReadLine()
9      switch result {
10       case "YES\n":
11         feedback <- Good
12       case "NO\n":
13         feedback <- Bad
14     }
15
16     continue <- i
17   }
18 }
```

Consider Listing 66, which outlines the pseudo code of the reduction loop when working with scripts. A token graph, which holds the parsed input data, as well as the reducing strategy, for reducing the input data, are passed on to function `deltaDebugScript`. The reducing strategy receives in Line 2 the token graph to operate on, and returns two channels. In order to communicate that the current processing step is finished the channel `continue` is used, i.e., the call in Line 4 blocks until the reducing strategy has produced another permutation. The channel `feedback` is used to signal whether the current permutation still triggers the expected behavior, thus steering the reducing strategy.

Chapter 6

The go-mutesting Framework

This chapter presents GO-MUTESTING, a framework for performing mutation testing on source code of the programming language GO. Its main purpose is to find source code, which is not covered by any tests. The implementation of GO-MUTESTING with all its assets has been open sourced and is freely available at [4].

6.1 Motivation

The generation of test suites for existing software systems is a major use case of TAVOR. One way of evaluating the quality of these test suites is to use mutation testing [14], i.e., the software under test gets modified and the generated test suite is verified by checking whether at least one test case fails and therefore if it catches the modifications. A more thorough description of mutation testing can be found in Section 2.3.

At the time of implementation of TAVOR, there has not been any adequate mutation testing tool for the programming language GO. All three existing tools MANBEARPIG [7], MUTATOR [8] and GOLANG-MUTATON-TESTING [6] have severe disadvantages:

- Only one type or even just one case of mutation is implemented.
- Only one mutation operator can be applied per execution. (MANBEARPIG, GOLANG-MUTATON-TESTING)
- The mutation of code is done by directly modifying the characters composing the code. This can lead to lots of invalid mutations, e.g., because of syntax errors. (GOLANG-MUTATON-TESTING)
- New mutation operators are not easily implemented and integrated since no common functionality nor API can be utilized.

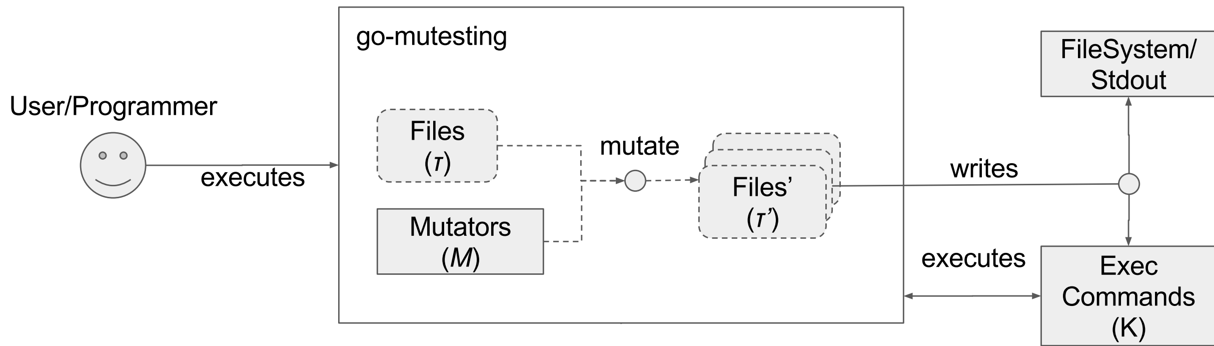


Figure 6.1: Architecture of GO-MUTESTING

- Only one package or file can be analyzed per execution.
- Other scenarios than `go test` cannot be applied.
- Proper cleanup or the handling of fatal failures is not implemented.
- No automatic tests exist to ensure that the algorithms are working at all.
- Another language than GO is used for the implementation and it is therefore not possible to utilize the up-to-date language parser. (GOLANG-MUTATON-TESTING)

Due to these insufficiencies we created the GO-MUTESTING framework, which as described in the following sections, overcomes all highlighted disadvantages.

6.2 Components

Figure 6.1 depicts the main components of the GO-MUTESTING framework. The user defines a set of files τ which should be mutated, and optionally a set of `exec commands` κ which should be executed for each modification. Such modifications during mutation testing are called mutants. The framework applies its individual mutation operators called mutators μ on each file τ, resulting in a set of mutated files τ' for each mutator-file pair. Each mutated file τ' is then written to the file system and executing the individual `exec commands` κ. After these executions are finished the framework prints the total number of mutants, the number of passing mutants and each failing mutant with their associated source code patches. We consider a mutant to pass in case the test suite failed for its mutation, which means that the mutant has therefore been killed.

Additional to the statistical output of mutants, GO-MUTESTING calculates and outputs the mutation score. Which is a metric on how many mutants have been killed by

the test suite and therefore states the quality of the test suite. The mutation score is calculated by dividing the number of passed mutants by the number of total mutants. If we had for example a total of eight mutants, where six are passing then the mutation score would be $6/8 = 0.75$. A score of 1.0, which is most desirable, means that all mutants have been killed.

6.3 Mutators

The mutation operators of the GO-MUTESTING framework are called mutators, and are used to introduce small deltas into the source files at hand. Mutators operate directly on the AST of a file and must offer two kinds of operations:

- The `Change` operation adapts the AST node at hand.
- The `Reset` operation restores the original AST node.

Working directly on the AST has the advantage, that the introduced mutations are syntactical valid, i.e., compilable source code. Currently the following mutators are supported by the framework:

- The `if`-mutator replaces the body of an if-branch with a NOOP statement, which creates an empty usage for every identifier of the substituted body. This is necessary, since the programming language GO marks unused identifiers as syntactical errors.
- The `else`-mutator replaces the body of an else-branch with a NOOP statement.
- The `case`-mutator replaces the body of a case-clause with a NOOP statement.
- The `remove-expression`-mutator modifies the binary logical operators `AND` and `OR`. The `AND` operator is mutated by replacing its left and right operands with the constant `true`. The `OR` operator is dealt with along the same lines by using the constant `false`. For instance, the binary logical expression `var1 && var2` results in the two mutations `true && var2` and `var1 && true`.
- The `remove-statement`-mutator modifies statements, such as assignments and the increment/decrement statement, by replacing them with a NOOP statement.

The above list of available mutators can easily be extended. A new mutator simply needs to provide the two mutation operations `Change` and `Reset`. Additionally, each mutator needs to be registered with the mutator registry of GO-MUTESTING in order to be applied during mutation testing.

6.4 Exec Commands

`Exec commands` are used by GO-MUTESTING to define the actions which should be taken for each individual mutant. Consider for instance a package with the following files: `LinkedList.go` and `LinkedList_test.go`, where `LinkedList.go` is a linked list implementation and `LinkedList_test.go` is the corresponding file for its tests. For each mutant that GO-MUTESTING produces for the file `LinkedList.go`, a temporary file with the modifications of the mutant is written. Afterwards the specified `exec command` is called, executing the tests within `LinkedList_test.go`.

A built-in `exec command` is provided by GO-MUTESTING, which implements the following behavior:

1. Temporarily overwrite the original file, in our example `LinkedList.go`, with its mutated content.

2. Execute all tests of the package under test, which are in our example located in `LinkedList_test.go`.

3. Restore the original file.

4. Report whether the mutant has been successfully killed.

Customized `exec commands` can be specified as command line parameters of GO-MUTESTING. Environment variables such as `MUTATE_CHANGED` and `MUTATE_ORIGINAL` are used to communicate the path of the original as well as the mutated file to the `exec command`. In order to report success or failure, the following exit codes need to be used by `exec commands`:

- Exit code 0 indicates that the mutant was killed, i.e., that the test led to a failed test after the mutation was applied.

- Exit code 1 indicates that the mutant is alive, i.e., that this could be a flaw in the test suite or even in the implementation.

- Exit code 2 indicates that the mutant was skipped, since other problems have been found such as compilation errors.

- An exit code greater than 2 indicates that the mutant produced an unknown exit code, which might be a flaw in the `exec command`.

Two examples of customized **`exec commands`** are provided by GO-MUTESTING at [1]: `test-current-directory.sh` may be used to execute all tests of the current directory and `test-mutated-package.sh` to execute all tests originating from the specified package.

[1] `https://github.com/zimmski/go-mutesting/tree/master/scripts/exec`

Chapter 7

Evaluation

The evaluation performed in the context of this thesis is threefold: First we present a case study, which demonstrates the major capabilities of the Tavor framework and how the framework can be applied to software programs using the example of a coin vending machine. Next we evaluate the generic fuzzing capabilities of Tavor by comparing it with *aigfuzz* [2], a dedicated fuzzer for the *AIGER* formats [9]. Finally we fuzz the JSON format and compare these generations with the manually written test suite of Go's JSON package.

7.1 Case Study: Coin Vending Machine

A *coin vending machine* is the typical example for showcasing the practicality of model-based testing. In this section we introduce such a coin vending machine to give a step-by-step guide to the techniques presented in this thesis and to show how they can be applied to real world applications. The example is intentionally kept simple so that the basic functionality of each technique can be demonstrated. Source code that is not relevant for the demonstration but necessary for completeness can be found in Tavor's "A Complete Example" documentation [1].

The description of the demonstration is divided into the following subsections:

- Subsection 7.1.1 defines our use-case of the coin vending machine and its implementation.
- Subsection 7.1.2 defines the keyword-driven testing approach for defining test cases and to test the implementation.
- Subsection 7.1.3 defines our test cases using the Tavor Format, and generates and executes a test suite using the `fuzz` command of Tavor.

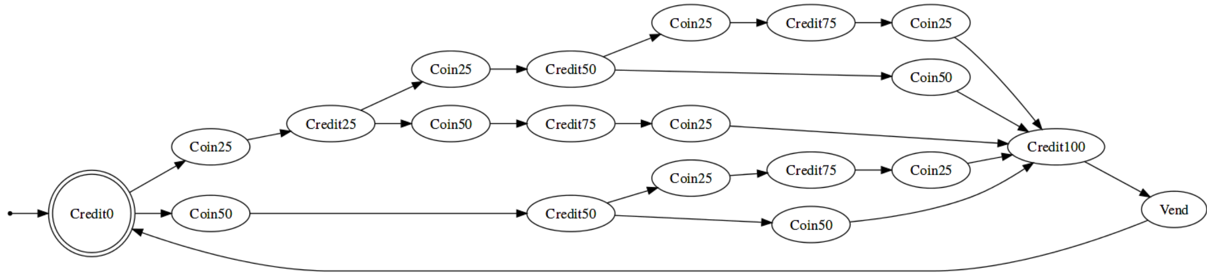


Figure 7.1: Example Automaton of the Coin Vending Machine

- Subsection 7.1.4 executes the generated test suite and identifies missing test cases by applying code coverage metrics and GO-MUTESTING.
- Subsection 7.1.5 introduces some intentional bugs and reduces failing test cases that trigger these bugs using the `reduce` command of TAVOR.

7.1.1 Definition of the Coin Vending Machine

In this case study we consider a basic coin vending machine, which accepts coins of two different kinds: `coin25` and `coin50` representing credits of value 25 and 50. The coin vending machine keeps track of the currently accepted credit and vends if a credit of 100 is reached. One option to model this behavior, using a finite state machine, is shown in Figure 7.1. Please note, that the given state machine could be defined more efficiently using state variables commonly used in model-based testing. While the TAVOR FRAMEWORK supports such state variables, the TAVOR FORMAT does not yet fully implement them. One possible direction of future work are such advancements of the TAVOR FORMAT. Please refer to Chapter 8.2 for more details on proposed future extensions.

In order to test the coin vending machine of our case study, an approach called keyword-driven testing is applied, which is explored in the next section.

7.1.2 Keyword-Driven Testing

The software testing technique *keyword-driven testing*, introduced by Fewster et al. in [13], also known as table-driven testing or action word based testing, separates the documentation of a test case from its implementation. Usually a sequence of keywords is used to specify the course of a test case, i.e., the sequence of keywords `Credit0, Coin50, Credit50, Coin50, Credit100, Vend, Credit0` specifies a sequence of actions to be executed for testing the coin vending machine. The implementation of the

individual actions for each keyword, such as Coin50, is separated from the test case. Thus, making it possible to write test cases without any programming knowledge.

In order to apply keyword-driven testing a format needs to be defined, which specifies the structure of a keyword-driven test case. A test case can then be saved into a single file called a *keyword-driven file*. Additionally, an executor is needed, which maps the individual keywords of such tests to concrete actions.

The Tavor framework offers the package `keydriven`, which provides various functionality to effectively apply keyword-driven testing. The function `ReadKeyDrivenFile` is used to parse a keyword-driven file and the type `Executor` is used to map keywords to their actions. Please consider Listing 67, which outlines the most important types and functions of package `keydriven`. Next, we use this functionality to define an executor for the coin vending machine, whose interface is shown in Listing 68. The interface of coin vending machine offers the functions `Credit`, `Coin` and `Vend`. For testing such a machine we introduce three keywords for our keyword-driven files with the following semantics: The keyword `credit` is used for validating the credit amount currently held by the vending machine, the keyword `coin` triggers the action of inserting a coin into the vending machine and finally the keyword `vend` invokes the vending action. Please note, that the keywords `coin` and `credit` need integer arguments specifying the coin credit amount.

Listing 67 Functionality of Package `Keydriven`

```
1  // ReadKeyDrivenFile reads in a keyword-driven file.
2  func ReadKeyDrivenFile(file string) ([]Command, error)
3
4  type Executor interface {
5      // Register adds a given key-action pair to the executor.
6      Register(key string, action Action) error
7      // Execute executes a set of keyword-driven commands.
8      Execute(cmds []Command)
9  }
10
11 // NewExecutor initializes and returns a new executor.
12 func NewExecutor() *Executor
```

The executor connects the keyword-driven files, each representing a test scenario, with the implementation under test. It reads, parses and validates keyword-driven files, executes sequentially each key with its arguments by invoking actions of the implementation and validates these actions. A test passes if each action executes without any problems. The pseudocode for an executor for the coin vending machines of the case study is shown in Listing 69. Please note, that the pseudocode does not include any error handling for the keyword-driven files and command line arguments in order to

Listing 68 Interface of the Coin Vending Machine Module

```
1 type VendingMachine interface {
2     // Credit returns the current credit of the coin vending machine.
3     Credit() int
4     // Coin inserts a coin into the coin vending machine and increases
       ↪  its credit.
5     Coin(credit int) error
6     // Vend executes a vend of the machine if enough credit (100) was
       ↪  put in and returns true.
7     Vend() bool
8 }
```

keep the code listing short. The main method subsequently parses the keyword-driven file, initializes the executor and executes the read commands. The actual connection between keywords and actions is made in function `initExecutor`, which registers for each keyword a callback function using the functionality of the provided package `keydriven`. Consider for instance the registration of the keyword `credit`, which includes one parameter denoting the expected amount of credit.[1] When this keyword is encountered, the defined expected credit is compared with the currently held credit of the coin vending machine. In case these two values diverge, an error is returned, indicating that the test scenario failed.

The next step for testing the coin vending machine is to generate keyword-driven files, which is done in the next section by applying TAVOR'S format and fuzzing capabilities.

7.1.3 Tavor Format and Fuzzing

A valid keyword-driven file for the TAVOR FRAMEWORK needs to adhere to the following conditions: Each line starts with and holds at most one keyword. Each keyword may be followed by zero, one or more arguments, where each argument is preceded by a tab character. Finally each line ends with the new-line character. Combining the state machine shown in Figure 7.1, the defined keys and the rules for the keyword-driven format together, results in the TAVOR FORMAT shown in Listing 70.

This format file can now be easily fuzzed using the TAVOR binary, resulting in outputs such as Listing 71. Since there is a loop in the specified format, the graph can be traversed more than once, resulting in longer keyword-driven files which execute the action `vend` more than once. The default fuzzing strategy `Random` can create all possible permutations of a format but since it is random, it will need enough time to do so.

[1] Validating the presence and type of keyword parameters has awas skipped to keep the pseudocode short.

Since even random events often lead to choosing the same path in a graph, many duplicated results will be generated using the `Random` fuzzing strategy. To work around this problem the `AllPermutations` strategy can be used which, as its name suggests, generates all possible permutations of a graph. This strategy should be used wisely since even small graphs can have an enormous amount of permutations. Also, since the example graph has a loop, we can state that there is an infinite amount of permutations. To work around this additional problem, the `--max-repeat` argument, which enforces a maximum for traversals and repetitions of loops, is used with a suitable value. Choosing a good value for `--max-repeat` is challenging, because high values may result in many repetitive permutations that will not improve the testing process. Choosing a small value on the other hand can lead to a bad coverage, which means that some scenarios are not tested.

The chosen Tavor Format for testing the coin vending machine of this case study is fairly easy to understand as it contains only a single loop. We choose the value 2 for `--max-repeat` to ensure that the repetitive part of the state machine is executed at least once. By executing the following command: `tavor --format-file vending.tavor --max-repeat 2 fuzz --strategy AllPermutations --result-folder testset --result-extension ".test"`. A total of 31 keyword-driven files is generated and stored in folder `testset`. Each written test file is named by Tavor according to its MD5 hash with the specified extension `.test`.

Since all components for testing the given state machine are now defined the next step is to execute the actual tests using the executor. One option to do this, is by invoking the program of Listing 69 for each test file, e.g., `go run executor.go testset/fba58bb35d28010b61c8004fadcb88a3.test`.

Executing each keyword-driven file separately is tedious. A better solution would be to extend the executor, but this would also mean more restrictions and more flaw possibilities in the executor code. Alternatively a simple Bash script which executes each keyword-driven file of the folder `testset` and immediately exits if the execution of a file fails can be used. An example for such a Bash script is shown in Listing 72.

Executing this script reveals no errors, i.e., all tests passed. In the next section we introduce some defects into the implementation of the coin vending machine and check whether the test suite is able to reveal them.

Listing 69 Example Executor for a Coin Vending Machine

```
1  func main() {
2    cmds := keydriven.ReadKeyDrivenFile(os.Args[1])
3    executor := initExecutor()
4    if err := executor.Execute(cmds); err != nil {
5      os.Exit(exitFailed)
6    }
7    os.Exit(exitPassed)
8  }
9
10 func initExecutor() *keydriven.Executor {
11   executor := keydriven.NewExecutor()
12   machine := implementation.NewVendingMachine()
13   executor.Register("credit", func(key string, parameters ...string)
       ↪  error {
14     expected := strconv.Atoi(parameters[0])
15     got := machine.Credit()
16     if expected != got {
17       return fmt.Errorf("Credit should be %d but was %d", expected,
           ↪  got)
18     }
19     return nil
20   })
21   executor.Register("coin", func(key string, parameters ...string)
       ↪  error {
22     if err := machine.Coin(strconv.Atoi(parameters[0])); err != nil {
23       return err
24     }
25     return nil
26   })
27   executor.Register("vend", func(key string, parameters ...string)
       ↪  error {
28     if vend := machine.Vend(); !vend {
29       return fmt.Errorf("Could not vend")
30     }
31     return nil
32   })
33   return executor
34 }
```

Listing 70 Tavor Format for Coin Vending Machine

```
1  START = Credit0 *(Coin25 Credit25 | Coin50 Credit50)
2
3  Credit0  = "credit" "\t" 0 "\n"
4  Credit25 = "credit" "\t" 25 "\n" (Coin25 Credit50 | Coin50 Credit75)
5  Credit50 = "credit" "\t" 50 "\n" (Coin25 Credit75 | Coin50 Credit100)
6  Credit75 = "credit" "\t" 75 "\n" Coin25 Credit100
7  Credit100 = "credit" "\t" 100 "\n" Vend Credit0
8
9  Coin25 = "coin" "\t" 25 "\n"
10 Coin50 = "coin" "\t" 50 "\n"
11
12 Vend = "vend" "\n"
```

Listing 71 Key-Driven File for the Coin Vending Machine Test Scenario

```
1 credit  0
2 coin    25
3 credit  25
4 coin    50
5 credit  75
6 coin    25
7 credit  100
8 vend
9 credit  0
```

Listing 72 Key-Driven Test Script

```
1  #!/bin/bash
2  shopt -s nullglob
3  for file in testset/*.test
4  do
5    echo "Test $file"
6    ./executor $file
7    if [ $? -ne 0 ]; then
8      echo "Error detected, will exit loop"
9      break
10   fi
11 done
```

7.1.4 Mutation Testing

Although the test suite introduced in Subsection 7.1.3 passes, it is no guarantee that it verifies all functionality of the implementation of the coin vending machine. The first step to determine the quality of the test suite is to look at its code coverage. By converting the given test cases to ordinary Go test cases we can record their code coverage using the command`go test -coverprofile=coverage.out` followed by the command `go tool cover -html=coverage.out` to display the coverage information. Using this approach we determined that only three lines are not covered of an overall of 18 coverable lines. These three lines handle two negative cases of the coin vending machine: the first is the insertion of an unknown coin, i.e., an unknown coin amount, and the second is the invocation of the vending action, even though an amount of 100 credits has was reached. Both scenarios are not implemented in our model for generating test cases and are not implemented in the executor. Hence, new keywords would need to be introduced to generate and handle these scenarios. The shown code coverage and our interpretation of the results therefore match our model. However, code coverage only determines that a statement was executed but does not prove that a statement has actuawas verified.

The main purpose of mutation testing is to determine the quality of a test suite at hand by determining whether statements are tested by a given test suite. This technique, as well as GO-MUTESTING, a framework for applying it, was introduced in detail in Chapter 6. We apply mutation testing on our implementation and test suite by invoking the command `TESTSET=./testset/ go-mutesting coin`. Please note, that "./testset/" states the relative path to the directory holding the generated test cases and "coin" defines the Go package that should be analyzed. The relevant parts of the output for this command can be found in Listing 92 of Appendix C. Since the implementation is rather simple, only a small amount of mutations can be applied. A total of five mutations were applied by GO-MUTESTING, of whom three were killed by the generated test suite. This results in a mutation score of $3/5 = 0.6$. Investigating the two alive mutations shows the same result as with the code coverage analysis, i.e., only two negative cases are not covered. Hence, the used model to generate test cases applies to all possible positive test scenarios.

The remainder of this section discusses different flaws that can be introduced into the implementation of the coin vending machine, and checks, whether the previously generated test suite catches them. The following defects were introduced separately to the coin vending machine implementation:

1. **The `Coin` method no longer increases the credit:** This flaw can easily be introduced either by removing the addition in the `Coin` method, or by using a

non-pointer type as receiver for the `Coin` method which leaves the state of the machine untouched.

2. **The `Vend` method no longer decreases the credit:** This flaw can be introduced along the same lines as the previous one, by either removing the subtraction in method `Credit` or by using a non-pointer type as a receiver.

3. **Every second 25 coin no longer increases the credit:** The defect type "works once but not twice" can be found in many programs. To emulate this kind of defect an additional state member was introduced to the coin vending machine implementation in order to trigger such a defect on every second call of method `Coin` with value 25.

Please note, that the first two defect types are automatically introduced using GO-MUTESTING. The third defect type on the other hand is not yet supported by the framework and needs to be introduced by hand.

The test suite from Section 7.1.3 was executed individually for each of the above flaws. Each defect type was successfully revealed, showcasing that the TAVOR FRAMEWORK can be used to generate test suites with little effort, that are able to catch these implementation flaws. Besides its support for fuzzing and keyword-driven testing, the TAVOR FRAMEWORK may be also used to apply delta-debugging in order to decrease debugging times. Applying delta-debugging to the coin vending machine case study is the content of the subsequent section.

7.1.5 Delta-Debugging

Delta-debugging, introduced in Section 2.4, is a technique to automatically reduce error revealing inputs in order to make debugging easier for developers. Consider, for instance, a very long keyword-driven file revealing a defect. A developer would need to walk through a very long path through the state machine before she is able to find the cause of the problem. Often not the whole keyword-driven file is necessary to successfully reproduce the problem. When this is the case delta-debugging comes in handy to automatically reduce these keyword-driven files. The final result of the delta-debugging process should be a minimal test case, which still triggers the same defect as the original test case. This can be automatically or semi-automatically done by the `reduce` command of the TAVOR binary. The binary uses our TAVOR FORMAT file to parse and validate the given keyword-driven file and for reducing its data according to the rules defined by the format file. For instance optional content like repetitions

can be reduced to a minimal repetition. In the coin vending machine case study the iterations of the vending loop can be reduced.

When executing test case `testset/fba58bb35d28010b61c8004fadcb88a3.test`,[2] for the third defect type of Section 7.1.4, then the output in Listing 73 is generated. The introduced defect is triggered in the second vending iteration, because every second 25 coin does not increase the machine's credit counter.

Listing 73 Key-Driven Test Output

```
1  credit [0]
2  coin [50]
3  credit [50]
4  coin [50]
5  credit [100]
6  vend []
7  credit [0]
8  coin [50]
9  credit [50]
10 coin [25]
11 credit [75]
12 coin [25]
13 credit [100]
14 Error: Credit should be 100 but was 75
```

First the semi-automatic method of the Tavor `reduce` command is applied to the test case at hand. The given format file is used to reduce the given input. Every reduction step displays the question "Do the constraints of the original input still hold for this generation?" to the user. The user's task is to inspect and validate the reduced output of the original data and decide by giving feedback if the defect is triggered (`yes`) or not (`no`). The following command starts this process: `tavor -format-file vending.tavor reduce -input-file testset/fba58bb35d28010b61c8004fadcb88a3.test`. Please refer to Listing 91 of Appendix C for the generated outputs of this command.

Semi-automatic processes can be tedious for big data especially due to the manual validation. The Tavor binary does therefore provide several methods to reduce the given inputs in a fully automated manner. The executor written in Section 7.1.2 was reused for this process. The reduction process is aided by the executor by exiting with different status codes on success or failure. The following command starts the fully automated delta-debugging process: `tavor -format-file vending.tavor reduce -input-file testset/fba58bb35d28010b61c8004fadcb88a3.test -exec "./executor TAVOR_DD_FILE" -exec-argument-type argument -exec-exact-exit-code`. Each exit status code of the executor is compared to the original

[2] This test case was generation in Section 7.1.3.

exit status code. If it is not equal, the reduction process will try an alternative reduction step until a reduction path is found that which leads to the minimum. The output of the execution of this command is shown in Listing 74.

Listing 74 Fully Automated Delta-Debugging for Coin Vending Machine

```
1 credit  0
2 coin    50
3 credit  50
4 coin    25
5 credit  75
6 coin    25
7 credit  100
8 vend
9 credit  0
```

The fully automated delta-debugging step concludes our case study, which demonstrates the major capabilities of the Tavor framework and how they can be applied to facilitate model-based testing, fuzzing and delta-debugging on software programs.

7.2 Fuzzing the AIGER ASCII Format

One of the major claims of the Tavor framework is to be a generic fuzzing tool, i.e., by providing the respective Tavor format, inputs of any format may be fuzzed. To evaluate this claim we compare the generations of *aigfuzz* [2], a dedicated fuzzer for the AIGER formats [9], with the generations of Tavor. In Subsection 7.2.1 we shortly introduce the AIGER ASCII format, subsequently Subsection 7.2.2 outlines the details of the experimental setup and finally Subsection 7.2.3 presents the results and conclusions we draw from this evaluation.

7.2.1 Introducing the AIGER ASCII Format

The AIGER ASCII format is used to model and-inverter graphs, containing inputs, outputs, latches, and-gates and inverters. We will discuss the structure and constraints of this format using its Tavor format definition shown in Listings 75 and 76.[3]

[3]Please note, that the Tavor format definition was split up into two listings due to its length.

Listing 75 Tavor Format Denoting the AIGER ASCII Format Part One

```
1  $Variable Sequence = start: 2,
2                       step: 2
3  ExistingLiteral = 0, // false
4                  | 1, // true
5                  | $Variable.Existing,
6                  | ${Variable.Existing + 1} // +1 means a NOT for this
                   ↪  input
7  Inputs = *(Input)
8  Input = $Variable.Next "\n"
9  Latches = *(Latch)
10 Latch = $Variable.Next " " ExistingLiteral "\n"
11 Outputs = *(Output)
12 Output = ExistingLiteral "\n"
13 ExistingLiteralAnd = 0, // false
14                    | 1, // true
15                    | ${Variable.Existing not in (AndCycle)},
16                    | ${Variable.Existing not in (AndCycle) + 1} // +1
                      ↪  means a NOT for this input
17
18 // AndCycle finds all paths beginning from the variable andLiteral
19 AndCycle = ${andList.Reference path from (andLiteral) over (e.Item(0))
   ↪  connect by (e.Item(2) / 2 * 2, e.Item(4) / 2 * 2) without (0, 1)}
20 Ands = *(And)
21 And = $Variable.Next<andLiteral> " " ExistingLiteralAnd " "
   ↪  ExistingLiteralAnd "\n"
22
23 Header = "aag ",
24          (,                                                   // M
25            ${Inputs.Count + Latches.Count + Ands.Count},
26          | ${Inputs.Count + Latches.Count + Ands.Count + 1}, // M does
            ↪  not have to be exactly I + L + A there can be unused
            ↪  Literals
27          ) " ",
28          $Inputs.Count " ",                                   // I
29          $Latches.Count " " ,                                 // L
30          $Outputs.Count " ",                                  // O
31          $Ands.Count "\n"                                     // A
```

Listing 76 Tavor Format Denoting the AIGER ASCII Format Part Two

```
1  Body = Inputs,
2         Latches,
3         Outputs,
4         Ands<andList>
5
6  Comments = "c\n",
7             *(Comment)
8  Comment = *([\w ]) "\n"
9
10 Symbols = +0,$Inputs.Count(SymbolInput),
11           +0,$Latches.Count(SymbolLatch),
12           +0,$Outputs.Count(SymbolOutput)
13
14 SymbolInput = "i" $Inputs.Unique<=e> $e.Index " " +([\w ]) "\n"
15 SymbolLatch = "l" $Latches.Unique<=e> $e.Index " " +([\w ]) "\n"
16 SymbolOutput = "o" $Outputs.Unique<=e> $e.Index " " +([\w ]) "\n"
17
18 START = $Variable.Reset,
19         Header,
20         Body,
21         ?(Symbols),
22         ?(Comments)
```

Variables in the AIGER ASCII format are indexed using positive even integer values greater or equal to two, this coherence is modeled by using a sequence in the token `Variable`. Inverters in the AIGER ASCII format are denoted by setting the least significant bit of a literal, due to this reason only even integers are used for variables. Let us assume literal 2 denotes an input to the and-inverter graph, then the literal 3 is used to denote the negation of this input. All existing literals for the and-inverter graph are defined in token `ExistingLiteral`. The constants `TRUE` and `FALSE` are denoted using the literals 0 and 1. Additionally, all variables and their negations are part of all available existing literals.

The inputs and outputs of the and-inverter graph are represented by a single valid variable. A latch is defined by first listing its current state followed by its next state separated by a whitespace character. The AIGER ASCII format poses by far the most constraints on used and-gates. An and-gate is defined by first listing its left-hand side literal followed by its two right-hand side literals each separated by whitespace characters, e.g., for storing the result of the AND operation on the two variables 2 and 4 in variable 6 one would need to write `6 2 4`. This coherence is captured by token `And`. The AIGER ASCII format allows the connection of several and-gates, i.e., the left-hand side literal of an and-gate, or its negation, may be used in the right-hand side literal of another and-gate. However, it is prohibited to model cycles of and-gates

in the and-inverter graph. Hence, the tokens `ExistingLiteralAnd` and `AndCycle` are necessary to ensure no cycles are generated by TAVOR.

The header of the AIGER ASCII format starts with the string `aag` followed by five integers `M I L O A` separated by whitespace characters. Where `M` denotes the number of variables, `I` the number of inputs, `L` the number of latches, `O` the number of outputs and `A` the number of and-gates. The header is followed by the body of the format, listing consecutively the definitions of inputs, latches outputs and and-gates. The body is succeeded by an optional symbol table and an optional comment section. A symbol table is used to connect symbols, which are ASCII strings, to specific inputs, outputs or latches. Please note, that at most one symbol can be connected to a specific variable. In order to connect the first input with symbol `my_input` we need to denote `i0 my_input`. For more details on the constraints and structure of the AIGER ASCII format, please refer to [9].

During the conduction of this evaluation we observed, that while it is possible to represent the AIGER ASCII format with the TAVOR FRAMEWORK, it is a challenging task to come up with the correct format definition and semantics. Of course, it is still more time consuming to write a dedicated AIGER ASCII format fuzzer than defining its TAVOR FORMAT, since a lot of boilerplate code and algorithms have to be created to handle the fuzzing part of the defined structures. Additionally, data structures as well as the architecture of such a dedicated fuzzer have to be defined and implemented instead of simply defining the syntax and semantics of a format, as can be done using the TAVOR FORMAT. In the next subsections we will compare the generations of TAVOR using the format defined in this subsection with the generations of the *aigfuzz* fuzzer.

7.2.2 Experimental Setup

This subsection describes the experimental setup of this evaluation in detail. First we describe the hardware of the evaluation, then we outline how the test sets were generated and finally we describe the compilation and execution details.

Hardware
The generation of the test sets as well as their compilation and execution, have been executed in a KVM VM with OpenSUSE 42.3 as the hypervisor and guest operating system. The VM consisted of 4 virtual cores of an Intel Xeon CPU E3-1275 v5 with 3.60GHz, 16GB of DDR4 ECC RAM with a clock frequency of 2400 MHz and a dedicated volume to two Toshiba XG3 M.2 NVMe drives in a software RAID1 configuration.

Test Set Generation

In order to compare the generations of *aigfuzz* with the generations of the TAVOR FRAMEWORK, three different test sets were created. The first test set was generated using the command `aigfuzz -a`, in order to generate only AIGER ASCII format files using the aigfuzz tool. The remaining two test sets were created with the TAVOR FRAMEWORK using the format introduced in Listings 75 and 76. One is generated using the *random* fuzzing strategy, where the parameter defining the maximum number of loop unrollings `max-repeat` is set to `10`, in order to ensure that also large files are being generated. The other test set is generated using the *AlmostAllPermutations* fuzzing strategy, with `max-repeat` set to `1`[4].

For the randomly generated test sets a maximum of unique 10.000 test cases were generated. And for the *AlmostAllPermutations* test set of TAVOR all tests were created, i.e., 204 tests. Please note, that the number of possible tests for this fuzzing strategy is finite as it is bounded through the definition of `max-repeat`. Each of the generated test sets is stored in its own folder, where each generated file is stored named by its MD5 checksum using the file extension `.test`, e.g., `b6fe6a6049c39ab112f778769e92cc76.test`.

Compilation and Execution

The comparison of the three test sets is performed by measuring the code coverage they reach in the AIGER toolset, which provides various executables operating on AIGER ASCII files. For this experiment we used the AIGER toolset in version 1.9.4. from `http://fmv.jku.at/aiger/`.

In order to compile the toolset, we first created the default `Makefile` of the toolset by running `./configure`. Next this `Makefile` has been adapted to do compilations using Clang in version 4.0.1., by setting `CC=clang-4.0.1`. Additionally, we changed the environment variable `CFLAGS` to `-O3 -DNDEBUG -fprofile-instr-generate -fcoverage-mapping` to make the coverage information for each execution available, and the address sanitizer option `-fsanitize=address` was appended to `CFLAGS`, enabling the detection and reporting of memory errors during runtime. Finally we ran `make` to compile all binaries of the toolset with the defined configuration.

The LLVM developer tools were used to make the coverage information for this evaluation available. To run for instance the `aigand` tool on test case `12666cece4dbf5bfc1e1d46c02819da` of the `Random` fuzzing strategy the following command needs to be executed: `LLVM_PROFILE_FILE=./aiger/outa.profraw ./aiger/aigand ./aiger/tavor-random/12666cece4dbf5bfc1e1d46c02819da.test`. This command stores the coverage information of this execution in file `./aiger/outa.profraw`. Before this file can be used, it needs to be indexed

[4] Higher values for `max-repeat` are currently not supported for the AIGER format in combination with the *AlmostAllPermutations* fuzzing strategy due to an error in the generation of permutations.

Filename	Regions	Missed Regions	Cover	Lines	Missed Lines	Cover
er.c	1687	821	51.33%	3121	1456	53.35%
er.h	0	0	-	0	0	-
or.c	56	17	69.64%	116	39	66.38%
TOTAL	1743	838	51.92%	3237	1495	53.82%

Figure 7.2: Code Coverage Example

using the following command `llvm-profdata merge -sparse outa.profraw -o outa.profdata`.

Figure 7.2 depicts the relevant data which has been gathered for a single run. Code regions may span multiple lines, i.e., for blocks without any control flow. But it is also possible that a single line contains several regions, e.g., in `if(a || b)`. Lines inform about the lines of code that have been covered. Please refer to [3], for a detailed description of the used coverage tool and its workflows.

Since we run each test case separately per tool, we need to merge the code coverages for the individual runs in order to gain the cumulative coverage of a test set. To merge two code coverages the following command was executed: `llvm-profdata merge -sparse foo1.profraw foo2.profdata -o foo3.profdata`.

For automatically running each generated test on the tools of the AIGER toolset and computing their cumulative coverage per test set, a Go script has been devised. The pseudocode outlining the contents of this script is shown in Listing 77. It iterates over the three test sets and cleans up their execution directory using the auxiliary function `cleanupExecutionDirectory`. For each of the AIGER tools used for this evaluation the tests of a test set are executed using the auxiliary function `execute`, which returns the coverage `c` as well as exit status `e` of the execution. These code coverages and exit states are stored and cumulated per tool and test set. Finally, the results of the evaluation are printed out for examination.

7.2.3 Results and Conclusions

The statistic of the generated test sets is presented in Table 7.1, where column `Name` denotes the name of the test set using the naming convention <tool>-<fuzzing strategy>, `Cases` holds the number of unique tests in the test set, `Invalid` informs about the number of invalid generations, i.e., those not conforming to the format specification, `Size(MB)` holds the required disk space of the test set in megabytes and `Time(s)` informs about the time needed to generate the test set using the respective tool in seconds. When comparing the test set `aigfuzz-random` with `tavor-random` we observe that Tavor's generation is almost twice as fast as the generation of *aigfuzz*, which may be a result of the generated test set size. The tests generated by *aigfuzz* are 3.5 times

Listing 77 Pseudo Code for the Script for Performing the AIGER Evaluation

```
1  testSets := [...]string{"aigFuzz -a", "tavarRandom", "tavorAllPerms"}
2  tools := [...]string{"aigand", "aigbmc -m", "aigflip", "aiginfo", ...}
3
4  for _, ts := range testSets {
5    cleanupExecutionDirectory(ts)
6    for _, tool := range tools {
7      for _, t := range testsOfTestSet(ts) {
8        c, e := execute(tool, t)
9
10       cumulateCoverage(ts, tool, c)
11       cumulateExitStatus(ts, tool, e)
12
13       cleanup(t)
14     }
15   }
16 }
17
18 printResults()
```

Table 7.1: Comparison of the Test Set Generation of the AIGER Evaluation

Name	**Cases**	**Invalid**	**Size(MB)**	**Time(s)**
aigfuzz-random	10000	0	1125	143.90
tavor-random	10000	918	40	82.70
tavor-aap	204	0	0.8	1.40

larger than the ones generated by TAVOR. Additionally, TAVOR generated 918 invalid tests which has not been done on purpose. Hence, either the format specification or the TAVOR FRAMEWORK contains a bug. By far the smallest test set, with 204 tests, is represented by `tavor-aap`, which has been generated with the *AlmostAllPermutations* fuzzing strategy.

The 16 commands which were used for this evaluation are listed in Table 7.2. On average these commands span 3400 lines of code and contain 1830 code regions per command.

The execution results of this experiment are shown in Table 7.3. The `lines-missed` resp. `regions-missed` inform about source code lines resp. regions which have not been covered by the test set. Columns `lines-percentage` resp. `regions-percentage` inform about the covered lines resp. regions in percent. Next, the column `Address-Sanitizer` denotes the number of executions for which LLVM's AddressSanitizer has been able to detect a problem. The number of runs for which an AIGER command exited with an exit status not equal to zero are captured in column `ExitStatusNotZero`.

Table 7.2: Comparison of the Commands of the AIGER Evaluation

Command	**Lines**	**Regions**
aigand	3259	1749
aigbmc	3477	1934
aigflip	3264	1751
aiginfo	3164	1704
aigmiter	3332	1825
aigmove	3272	1783
aignm	3164	1704
aigor	3237	1743
aigsim	3832	2101
aigsplit	3342	1768
aigtoaig	3370	1794
aigtoblif	3491	1887
aigtocnf	3323	1799
aigtodot	3348	1796
aigtosmv	3388	1820
aigunroll	4123	2167

Finally, column `FileNotApplicable` denotes the number of test cases that have not been accepted, because their structure is not suitable for the respective command.

The experiment shows that Tavor's *random* fuzzing strategy reaches `2.6% to 15.26%` more coverage than *aigfuzz*. Even though, Tavor generated substantially smaller files in less time. Since the coverage difference is considerable more than a few lines, the assumption that the invalid test cases generated by Tavor are solely responsible for the higher coverage can be neglected. Additionally, Tavor's *random* fuzzing strategy also led to more executions with exit states unequal to zero, which sometimes were due to the invalid test cases. For the commands `aigtocnf` and `aigunroll` there were cases which are valid but still led to non-zero exit states. Hence, `Tavor` did cover substantially more paths than *aigfuzz*. Furthermore, a fault found by LLVM's AddressSanitizer was discovered during the execution of Tavor's test suites for the command `aigflip` which *aigfuzz* did not find.

In comparison Tavor's *AlmostAllPermutations* fuzzing strategy reached for 13 commands in average `3.32%` more line coverage than *aigfuzz*. For the command `aigunroll` a coverage of `61.38%` was reached which even surpasses Tavor's *random* fuzzing strategy by `34.68%`. However, the line coverage of the commands `aigflip` and `aigmiter` did not exceeded the coverage of *aigfuzz*. Since these results have been generated with only `max-repeat` set to `1`, it can be assumed that a better coverage can be reached than with the *random* fuzzing strategy when setting `max-repeat` to a higher value. However, the results for this test set are already astonishing since the coverage, surpassing

Table 7.3: Comparison of the Test Set Execution of the AIGER Evaluation

Command	**Test Set**	**LM**	**LP**	**RM**	**RP**	**AS**	**ESNZ**	**FNA**
aigand	aigfuzz-random	1822	44.09	972	44.43	0	0	0
aigand	tavor-aap	1682	48.39	908	48.08	0	0	0
aigand	tavor-random	1326	59.31	753	56.95	0	918	0
aigbmc	aigfuzz-random	2185	37.16	1243	35.73	0	0	0
aigbmc	tavor-aap	2096	39.72	1209	37.49	0	0	0
aigbmc	tavor-random	1716	50.65	1043	46.07	0	918	0
aigflip	aigfuzz-random	1857	43.11	994	43.23	0	0	0
aigflip	tavor-aap	3142	0.00	1692	0.00	99	99	0
aigflip	tavor-random	1367	58.12	780	55.45	4099	5017	0
aiginfo	aigfuzz-random	2104	33.50	1165	31.63	0	0	0
aiginfo	tavor-aap	2050	35.21	1148	32.63	0	0	0
aiginfo	tavor-random	1997	36.88	1114	34.62	0	918	0
aigmiter	aigfuzz-random	1552	53.42	880	51.78	0	0	0
aigmiter	tavor-aap	1580	52.58	891	51.18	0	10	10
aigmiter	tavor-random	1342	59.72	781	57.21	0	1763	845
aigmove	aigfuzz-random	1903	41.84	1024	42.57	0	0	0
aigmove	tavor-aap	1761	46.18	953	46.55	0	0	0
aigmove	tavor-random	1414	56.78	810	54.57	0	918	0
aignm	aigfuzz-random	2092	33.88	1158	32.04	0	0	0
aignm	tavor-aap	1996	36.92	1116	34.51	0	0	0
aignm	tavor-random	1985	37.26	1107	35.04	0	918	0
aigor	aigfuzz-random	1816	43.90	968	44.46	0	0	0
aigor	tavor-aap	1678	48.16	905	48.08	0	0	0
aigor	tavor-random	1322	59.16	749	57.03	0	918	0
aigsim	aigfuzz-random	2459	35.83	1404	33.17	0	0	0
aigsim	tavor-aap	2354	38.57	1359	35.32	0	0	0
aigsim	tavor-random	1980	48.33	1200	42.88	0	918	0
aigsplit	aigfuzz-random	1850	44.64	993	43.83	0	0	0
aigsplit	tavor-aap	1673	49.94	905	48.81	0	0	0
aigsplit	tavor-random	1360	59.31	779	55.94	0	918	0
aigtoaig	aigfuzz-random	2042	39.41	1078	39.91	0	0	0
aigtoaig	tavor-aap	1952	42.08	1030	42.59	0	0	0
aigtoaig	tavor-random	1588	52.88	879	51.00	0	918	0
aigtoblif	aigfuzz-random	2242	35.78	1268	32.80	0	0	0
aigtoblif	tavor-aap	2102	39.79	1194	36.72	0	0	0
aigtoblif	tavor-random	2050	41.28	1173	37.84	0	918	0
aigtocnf	aigfuzz-random	2326	30.00	1284	28.63	0	9967	9967
aigtocnf	tavor-aap	2257	32.08	1255	30.24	0	201	201
aigtocnf	tavor-random	2219	33.22	1233	31.46	0	9933	9015
aigtodot	aigfuzz-random	2229	33.42	1230	31.51	0	0	0
aigtodot	tavor-aap	2126	36.50	1192	33.63	0	0	0
aigtodot	tavor-random	2095	37.43	1168	34.97	0	918	0
aigtosmv	aigfuzz-random	2189	35.39	1222	32.86	0	0	0
aigtosmv	tavor-aap	2085	38.46	1176	35.38	0	0	0
aigtosmv	tavor-random	2063	39.11	1160	36.26	0	918	0
aigunroll	aigfuzz-random	3130	24.08	1648	23.95	0	9758	9758
aigunroll	tavor-aap	1593	61.36	912	57.91	0	10	10
aigunroll	tavor-random	3023	26.68	1597	26.30	0	9161	8243

LM=lines-missed, **LP**=lines-percentage, **RM**=regions-missed, **RP**=regions-percentage, **AS**=AddressSanitizer, **ESNZ**=ExitStatusNotZero, **FNA**=FileNotApplicable

aigfuzz for 14 out of 16 commands, has been reached with substantially less test cases and generation time than with the other test sets.

All things considered, one explicit bug has been found in the AIGER toolset and continuous better line and regional coverage has been reached with TAVOR's *random* fuzzing strategy in direct comparison to *aigfuzz* a dedicated fuzzer for the AIGER format. Additionally, TAVOR's *AlmostAllPermutations* fuzzing strategy reached better coverage for 14 out of 16 commands with its specific test case generation. Both fuzzing strategies reached these achievements with substantially smaller sized test cases and far less time for generating their test sets. In summary, this evaluation proved that TAVOR as a generic fuzzer can keep up against a dedicated fuzzer, and that even a small amount of small-sized test cases can outperform a bigger test set.

7.3 Fuzzing the JSON Format

Handling sophisticated formats, such as the AIGER format presented in Section 7.2, is just one area where a generic fuzzer must excel. Another area are simple formats without any semantics but with lots of variety and recursive data structures. This section takes a look at the widely used JSON format, to evaluate that the TAVOR FRAMEWORK can also efficiently define and fuzz such formats. The evaluation compares 3 generated test sets of TAVOR with the manually written test suite of the JSON implementation `encoding/json` of the programming language GO. In Subsection 7.3.1 we shortly introduce the JSON format, subsequently Subsection 7.3.2 outlines the details of the experimental setup and finally Subsection 7.3.3 presents the results and conclusions we draw from this evaluation.

7.3.1 Introducing the JSON Format

The JSON (JavaScript Object Notation) format is a human-readable text format to depict JavaScript's data structures such as strings, numbers, arrays and name-value pairs. Even though JSON's name might suggest that it is bound to the programming language JavaScript, it is language-independent. Nowadays JSON is widely used for saving data and most commonly utilized for the communication between services. The format has been officially defined in RFC 7159[5]. We will discuss the structure and constraints of this format using its TAVOR FORMAT definition shown in Listing 78. Please note, that reading the RFC and defining this format took about 3 hours.

[5] https://tools.ietf.org/html/rfc7159

Listing 78 Tavor Format Denoting the JSON Format

```
1  START = Value
2  Array = WS beginArray ?( Value *(valueSeparator Value) ) endArray WS
3  Object = WS beginObject ?( Member *(valueSeparator Member) ) endObject
   ↪  WS
4  Member = WS String WS nameSeparator Value // Member name, should be
   ↪  unique per object
5  Number = WS ?("-") (0 | [1-9]*([0-9])) ?("." +([0-9])) ?([eE] ?("-" |
   ↪  "+") +([0-9])) WS
6  String = WS "\"" *(Char) "\"" WS
7  Char = CharUnescaped | CharEscaped
8  CharUnescaped = [\x20-\x21] | [\x23-\x5B] | [\x5D-\x{10FFFF}] //
   ↪  10FFFF -> 21bit of unicode
9  CharEscaped = "\\" (,
10                       "\"", // quotation mark
11                     | "\\", // reverse solidus
12                     | "/", // solidus
13                     | "b", // backspace
14                     | "f", // form feed
15                     | "n", // line feed
16                     | "r", // carriage return
17                     | "t", // tab
18                     | "u" +4([a-fA-F0-9]), // 4 hex digits
19                     )
20 Value = (,
21           Object | Array  | Number | String,
22         | WS "false" WS, // must be lower case
23         | WS "null"  WS, // must be lower case
24         | WS "true"  WS, // must be lower case
25         )
26 // Helper
27 beginArray     = "["
28 beginObject    = "{"
29 endArray       = "]"
30 endObject      = "}"
31 nameSeparator  = ":"
32 valueSeparator = ","
33 WS = *([ \t\n\r])
```

The basic building block of the JSON format is a `Value` which represents every possible data structure of the format. A `Value` can be simple data such as a `Number` or a `String`, it can be a boolean value (`false`, `true`), the empty value `null` or it can contain combined values such as an `Array` or an `Object`, which defines name-value pairs. All of these definitions can be surrounded by whitespace characters. Even though, the value types `Number` and `String` store just simple data values, their representation can be rather complex, since they allow a variety of definitions, e.g., a `String` can consist of escaped and unescaped characters spanning the complete Unicode spectrum. For more details on the constraints and structure of the JSON format, please refer to RFC 7159.

During the conduction of this evaluation we observed that while Tavor is capable of fuzzing the format defined in Listing 78, it generates lots of test cases which look different to a human but exercise the same paths in a JSON implementation. To work around this problem, we defined a smaller format shown in Listing 79. The following reductions were made in comparison to the original format: first whitespace characters were completely removed from the generation, second the character generation was reduced to hold only the min and max permutations of the different character sets and lastly the number representation was reduced to a minimum set of interesting values. These manual reductions allowed to generate a smaller set of interesting test cases in a short amount of time. In the next subsections we will compare the generations of Tavor using the format defined in this subsection with the manually written test suite of the `encoding/json` package, the official implementation of the JSON format of the programming language Go.

7.3.2 Experimental Setup

This subsection describes the experimental setup of this evaluation in detail. First we outline how the test sets were generated and finally we describe the execution details. The hardware for the evaluation has the same setup as with the AIGER evaluation described in Subsection 7.2.2.

Test Set Generation

In order to compare the manually written test suite of the JSON implementation `encoding/json` of the programming language Go with the generations of the Tavor framework, three different test sets were created. The fuzzing strategy *AlmostAllPermutations* was used for all three test sets with `max-repeat` set to 1, 2 and 3 using the format introduced in Listing 79. Since this fuzzing strategy tries to generate targeted test cases and is bound to max-repeat, and finite set of test cases will be generated. Each of the generated test sets is stored in its own folder, where each gen-

Listing 79 Tavor Format Denoting a Minimum of the JSON Format

```
1  START = Value
2  Array = beginArray ?( Value *(valueSeparator Value) ) endArray
3  Object = beginObject ?( Member *(valueSeparator Member) ) endObject
4  Member = String nameSeparator Value
5  Number = "0" | "-0" | "949" | "999" | "-544" | "0.0" | "-0.0" | "0.4"
   ↪  | "0.00" | "0.40" | "-0.04" | "0e0" | "0e4" | "0E9" | "0e-9" |
   ↪  "0e+0" | "0e+9" | "0E-9" | "0E+0" | "0E+4" | "0e00" | "100E+99" |
   ↪  "140E+99" | "-999.99E+99"
6  String = "\"" *(Char) "\""
7  Char = CharUnescaped | CharEscaped
8  CharUnescaped = [\x20\x21\x23\x5B\x5D\x{10FFFF}]
9  CharEscaped = "\\" ( "\"" | "\\" | "/" | "b" | "f" | "n" | "r" | "t"
   ↪  | "u" ("0000" | "FFFF") )
10 Value = Object | Array | Number | String | "false" | "null" | "true"
11 beginArray     = "["
12 beginObject    = "{"
13 endArray       = "]"
14 endObject      = "}"
15 nameSeparator  = ":"
16 valueSeparator = ","
```

erated file is stored named by its MD5 checksum using the file extension `.test`, e.g., `833f52bdb291f5915a6620fdaefe48bc.test`. Additionally to the three generated test set, we also included a combination of these three test sets and the manually written test suite as the fifth "combined" test set.

Execution

The comparison of the generated test sets to the manually written test suite of `encoding/json` is twofold. First we measure the code coverage they reach in the implementation of `encoding/json`, afterwards we perform mutation testing using go-mutesting, which was introduced in Chapter 6. For this experiment we used Go in version 1.7.1 from https://golang.org/dl/.

Each generated test set is executed using its own Go test function, which loads all test cases and executes each of them separately using the function `testCase` of Listing80. The test case function `testCase` calls only two functions of the `encoding/json` implementation, namely `Marshal` and `Unmarshal`, and checks their execution for errors. Therefore, it can be assumed that the generated test cases will cover far less of the implementation as the manually written test suite. Furthermore, since the generation only include valid cases, no error paths will be executed. Hence, a direct comparison of the code coverage is not appropriate. However, the results of the killed mutations during mutation testing can be directly compared, since they indicate tested paths in the implementation.

Listing 80 Test Function for Generated JSON Data

```
1 func testCase(t *testing.T, data []byte) {
2   var o interface{}
3   err := Unmarshal(data, &o)
4   if err != nil {
5     t.Fatal(err)
6   }
7   _, err = Marshal(o)
8   if err != nil {
9     t.Fatal(err)
10  }
11 }
```

Since all test sets as well as the manually written test suite are implemented using Go test functions, the execution of the evaluation can be done using Go's testing tool and GO-MUTESTING. The `encoding/json` implementation was copied in its own `GOPATH` environment and the test sets have been added to each execution. To execute a test set and measure its coverage the command `GOPATH=$PWD/testset go test json -coverprofile=testset.coverage` was used, where `$PWD/testset` indicates the folder to the given test set, `json` defines the package which should be tested and `testset.coverage` determines the coverage file for the test set. The execution of mutation testing was similarly performed using the command `GOPATH=$PWD/testset go-mutesting json`.

7.3.3 Results and Conclusions

The statistic of the generated test sets is presented in Table 7.4, where column `Name` denotes the name of the test set, `Cases` holds the number of unique tests in the test set, `Size(MB)` denotes the required disk space of the test set in megabytes and `Time(s)` informs about the time needed to generate the test set in seconds. The *max-repeat-1* includes no recursive data structures, while *max-repeat-2* and *max-repeat-3* include such structures and will therefore presumably reach more coverage. However, the later two test sets also include far more test cases and took more time to generate. Especially *max-repeat-3* is a substantially larger test set and consumed a big amount of computational power for the generation of its test cases.

The implementation of `encoding/json` consists of `1649` coverable statements, and GO-MUTESTING found `1284` exercisable mutation of whom `147` are skipped because of various reasons, e.g., some mutations lead to uncompilable code. The execution results of this experiment are shown in Table 7.5. The `Missed Statements` column informs about statements that are not covered by the test set. Next, the column

Table 7.4: Comparison of the Test Set Generation of the JSON Evaluation

Name	**Cases**	**Size(MB)**	**Time(s)**
max-repeat-1	46	0.18	0.015
max-repeat-2	1277	5.1	0.912
max-repeat-3	71061	283	457.003

Table 7.5: Comparison of the Test Set Results of the JSON Evaluation

Name	**MS**	**SCP**	**KM**
original	166	89.93	838
max-repeat-1	1211	26.56	148
max-repeat-2	1124	31.84	181
max-repeat-3	1124	31.84	181
combined	166	89.93	840

MS = Missed Statements, **SCP** = Statement Coverage in Percentage, **KM**=Killed Mutations

`Statement Coverage in Percentage` detonates in percent how many statements have been covered. Finally, column `Killed Mutations` states how many mutations out of `1284` have been killed and are therefore checked by the test set.

The experiment shows that TAVOR's generated test sets reach `26.56%` and `31.84%` statement coverage solely by execution two functions of the underlying implementation, while the original manually written test suite covers `89.93%`. Combining the original test suite with the generated test sets does not lead to any additional statement coverage. Both test set *max-repeat-2* and *max-repeat-3* have the same coverage and mutation testing result. Hence, the additional computational time for generating the third test set with `max-repeat` set to 3 did not accomplish any additional coverage for the JSON format. Looking at the mutation testing results reveals that the *max-repeat-1* test set covered `11.52%` and *max-repeat-2* covered even `14.09%` of the overall mutations. Surprisingly, combining the original test suite with the generated test sets resulted in `2` additionally killed mutations, i.e., two code paths that were not checked by the original test suite. Investigating this result reveals that these mutations are already killed by the *max-repeat-1* test set. The first mutation shown in Listing 81 reveals that no test case of the original test suite checks upper-case characters in the third character of a Unicode escaped character, e.g., the escaped character `\uAC3C` belongs to this class of characters since the character "3" is not an upper-case character. The second mutation shown in Listing 82 would be checked by parsing an empty JSON object `{}` into an object of an empty GO interface. These mutations manifest that a behavioral change for these two cases would not be caught by the existing test suite.

Listing 81 Mutant Killed by Tavor Part One

```
1  --- /home/symflower/json/json/original/src/json/scanner.go  2017-12-01
   ↪  18:53:42.663580118 +0100
2  +++ /tmp/go-mutesting-
   ↪  288106446//home/symflower/json/json/original/src/json/scanner.go.119
   ↪  2017-12-03 10:43:45.026538957 +0100
3  @@ -378,7 +378,7 @@
4   // stateInStringEscU12 is the state after reading `"\u12` during a
    ↪  quoted string.
5   func stateInStringEscU12(s *scanner, c byte) int {
6  - if '0' <= c && c <= '9' || 'a' <= c && c <= 'f' || 'A' <= c && c <= 'F' {
7  + if '0' <= c && c <= '9' || 'a' <= c && c <= 'f' || false {
8      s.step = stateInStringEscU123
9      return scanContinue
10   }
11 FAIL "/tmp/go-mutesting-
   ↪  288106446//home/symflower/json/json/original/src/json/scanner.go.119"
   ↪  with checksum 3b91ecf37738b985eb6fa9a64bb9fcf9
```

Listing 82 Mutant Killed by Tavor Part Two

```
1  --- /home/symflower/json/json/original/src/json/decode.go 2017-12-01
   ↪  18:53:43.147579808 +0100
2  +++ /tmp/go-mutesting-
   ↪  288106446//home/symflower/json/json/original/src/json/decode.go.183
   ↪  2017-12-03 10:14:54.687623329 +0100
3  @@ -998,7 +998,7 @@
4      op := d.scanWhile(scanSkipSpace)
5      if op == scanEndObject {
6        // closing } - can only happen on first iteration.
7  -     break
8  +
9      }
10     if op != scanBeginLiteral {
11       d.error(errPhase)
12 FAIL "/tmp/go-mutesting-
   ↪  288106446//home/symflower/json/json/original/src/json/decode.go.183"
   ↪  with checksum 684f5c20122977ca389bb307ee0db6f7
```

All things considered, no additional coverage but two additional killed mutations have been achieved by this quick evaluation of the JSON format and the official JSON implementation of the Go programming language. Additionally, one inconsistency has been cleaned up and one behavioral change has been made due to this evaluation[6] in the Go project. These problems have been found, even though this implementation has been thoroughly tested since 2009. In summary, this evaluation proved that it is important to not solely rely on code coverage as a metric for stating the quality of a test suite but also to use more detailed analysis such as mutation testing. Additionally, we showed that Tavor fuzzing capabilities can be applied to diverse and highly recursive formats, and that its generations can lead with little effort to high coverage in a short amount of time.

[6] https://codereview.appspot.com/162340043/

Chapter 8

Conclusion

This chapter provides the conclusions we draw from this thesis, by first summarizing its results and by outlining the directions for future work on Tavor.

8.1 Summary

In general both testing and debugging software are cumbersome as well as error prone tasks which would strongly benefit from automated techniques. Fuzzing and delta-debugging are such techniques, which are very well known in literature. In this thesis we devised the Tavor framework that utilizes both approaches by operating on the same data model.

Especially in times where programming skills are such a rare trait, tools and techniques are needed which enable also non-programmers to apply automated testing techniques. For this reason, we designed the Tavor Format, which is used to specify the data model on which the Tavor framework operates on. This format enables also testers without any programming skills to define the data models that are needed for fuzzing and delta-debugging. Additionally, the Tavor CLI can be used to utilized the capabilities of the Tavor framework without the need to write any source code.

In our evaluation we demonstrated how the Tavor framework, format and CLI may be combined to efficiently test and debug software programs using the example of a simple coin vending machine. We used go-mutesting, a by-product of this thesis, to introduce bugs into the coin vending machine and validate the generated test suite. In addition to the coin vending machine case study we compared the fuzzing capabilities of the Tavor framework for the AIGER ASCII format with the *aigfuzz* command. In total 16 commands of the AIGER toolset were evaluated to compare the generated test sets. On average the random fuzzing strategy of the Tavor Framework reached `9.16%` more line coverage than *aigfuzz*. The best result has been obtained for the `aigor`

command, where *aigfuzz* covered `43.9%` and the Tavor Framework `59.16%`. Additionally to the random fuzzing strategy the *AlmostAllPermutations* fuzzing strategy was used to accurately generate a small test suite, which reached for 13 commands on average `3.32%` more coverage than *aigfuzz* and for the command `aigunroll` `37.28%` more coverage, which surpassed even Tavor's random fuzzing strategy. Our third and last evaluation compared the fuzzing capabilities of the Tavor framework to the manually written test suite of Go's JSON implementation `encoding/json`. No additional statement coverage was achieved. However, using go-mutesting to apply mutation testing revealed two killed mutations with the test sets generated by Tavor, i.e., the code paths of these mutations are not checked by the original test suite. Therefore, if the behavior of the code changes, these cases will not be caught by the manually written test suite.

The Tavor project as well as go-mutesting have been published as open source repositories. Both enjoy great popularity in the open source and Go community, even though they focus on specialized techniques. At the time of writing Tavor is cloned on average 3.5 times per day by unique persons and holds a rating of 172 stars, i.e., 172 persons value the publication of this repository. The go-mutesting repository is cloned on average twice a day and holds 117 stars. Similar popularity can be seen for the other contributions outlined in Section 1.3. Especially the major contributions to the *go-flags* package should be emphasized, since they led to the maintainership of this package, which holds now a rating of 918 stars and is cloned on average over 750 times per day. The *go-flags* package is also now one of the de facto standard packages to parse command line options and configurations. All of these statistics attest that the contributions of this master thesis are actively used and therefore significant.

In conclusion we successfully showed that this master thesis meets its goals of providing a generic fuzzing and delta-debugging framework, which can also be utilized by non-programmers. However, there are still many options for future improvements of the Tavor framework which are outlined in the next section.

8.2 Future Work

This master thesis touches a wide variety of topics. Of course, not all of them have been explored completely. Indeed there is a great number of options for future improvements of the Tavor Framework. In this section we chose to outline those with the most impact to the applicability of Tavor.

A substantial improvement to the Tavor framework would be to ensure that the Tavor format as well as the fuzzing and delta-debugging capabilities of the framework

offer the same functionality. The fuzzing capability of the framework is currently the most advanced concept as it is capable of handling all token types[1] of the framework. While the delta-debugging capability of the framework does support reducing all sorts of groups, repetitions and character classes, it does not yet support more advanced token concepts such as variables and attributes. Also the Tavor format is not expressive enough to model all token concepts, e.g., state variables. However, please keep in mind that the format does meet its original goal of being expressive enough to model the AIGER ASCII format.

Another direction of future work is the redesign of the used data structures. Currently, graph structures are used to represent the formats to operate on. These graph structures are very well suited for storing formats, but it is cumbersome to ensure that all dependencies are met while performing operations on them. An example of such dependencies are token attributes, where changing the value of one token needs to be propagated to all tokens referencing that value by a token attribute. Rather than using graph structures for representing formats, we propose to switch to an event-driven architecture. Whenever a certain token value changes, this change can then be propagated to all registered listeners of a token. This redesign has also the advantage that most fuzzing and reducing strategies are easier to implement.

Today the goal of fuzzing is mainly to find faults in programs under test. A Tavor format may be used to describe the structure of the data which should be used as an input. We strongly believe that a dedicated fuzzing strategy, which aims to execute all paths through the token graph of a format, is likely to gain higher code coverages with its generated test suites. Consider for instance a format for fuzzing command line options of a command. Repeatedly fuzzing a single parameter with different arguments may soon exhaust all paths that are reachable with this certain parameter. However, all other parameters of the command would not have been tested yet. The proposed dedicated fuzzing strategy aims to first visit each token in the token graph at least once, so that all paths of the token graph are covered, and therefore all parameters of the command of this example. Hence, resulting in a high code coverage for the program under test. Please note, that such a strategy is costly to implement. It requires, for instance, the use of a constraint solver in order to be able to specifically target each path through the token graph.

Currently a dedicated executor needs to be provided to enhance fuzzing and delta-debugging in an automated fashion. The addition of an execution layer to the Tavor framework would have the advantage that these executors no longer need to be provided by programmers. Such an execution layer in combination with the support for real loops, instead of unrolling loops, would enable online fuzzing, i.e., the continu-

[1]Tokens, introduced in Section 3.2, are the basic building blocks of the Tavor Framework.

ous feeding of data to a program under test. As long as loop unrolling is used as a replacement for real loops it is not possible to provide an executable with an endless stream of randomly generated data. Another useful extension on top of online fuzzing is feedback-driven fuzzing, where profiling techniques are used to determine which permutations led to further code coverage in the program under test. This information is then used to guide the fuzzing strategy.

The future extensions presented in this section are just the highlights, which would bring the most benefits to the end-users of the Tavor framework. There are also lots of minor feature suggestions in the issue tracker of the Tavor repository, which can be easily implemented even by novice programmers. We would be delighted if the work started with this master thesis would be carried on, in order to provide an even better tool for applying fuzzing and delta-debugging.

Appendix A

Tavor Framework Pseudo Codes

Listing 83 Pseudo Code of AllPermutations Fuzzing Strategy

```
1  func fuzz(contin chan struct{}, list []permLevel) {
2  Step:
3      for {
4          if len(list[0].children) > 0 {
5              fuzz(contin, list[0].children)
6          } else {
7              reportPermutation(contin)
8          }
9
10         list[0].perm++
11
12         if list[0].perm >= list[0].token.Permutations(){
13             for i:=1; i < len(list); i++ {
14                 incremented := incrementChild(list[i].children)
15                 if incremented {
16                     resetTokensBeforeIndex(list, i)
17                     continue Step
18                 }
19
20                 list[i].perm++
21
22                 if list[i].perm < list[i].token.Permutations() {
23                     resetTokensBeforeIndex(list, i)
24                     setEntry(list, i)
25                     continue Step
26                 }
27             }
28             break Step
29         } else {
30             setToken(list, 0)
31         }
32     }
33 }
```

Listing 84 Pseudo Code of AllPermutations Fuzzing Strategy Helper Function

```
func incOne(list []permLevel) bool {
    for {
        if len(list[0].children) > 0 {
            if incOne(list[0].children) {
                return true
            }
        }

        list[0].perm++

        if list[0].perm >= list[0].token.Permutations(){
            for i:=1; i < len(list); i++ {
                if incrementChild(list[i].children) {
                    resetTokensBeforeIndex(list, i)

                    return true
                }

                list[i].perm++

                if list[i].perm < list[i].token.Permutations() {
                    resetTokensBeforeIndex(list, i)
                    setEntry(list, i)

                    return true
                }
            }
            break Step
        } else {
            setToken(list, 0)

            return true
        }
    }

    return true
}
```

Appendix B

Tavor CLI Command Line Arguments

Listing 85 General Arguments of the Tavor CLI

```
1  General options:
2    --debug             Debug log output
3    --help              Show this help message
4    --verbose           Verbose log output
5    --version           Print the version of this program
6
7  Global options:
8    --seed=             Seed for all the randomness
9    --max-repeat=       How many times loops and repetitions should be
     ↪  repeated (default: 2)
10
11 Format file options:
12   --check             Checks the syntax of the format file and exits
13   --format-file=      Input Tavor format file
14   --print             Prints the AST of the parsed format file and
     ↪  exits
15   --print-internal    Prints the internal AST of the parsed format
     ↪  file and exits
```

Listing 86 Arguments for `Validate` the Command of the Tavor CLI

```
1  [validate command options]
2        --input-file=   Input file which gets parsed and validated via
         ↪  the format file
```

Listing 87 Example DOT Format for the `Graph` Command

```
1  digraph Graphing {
2    node [peripheries = 2]; xc4200e7810 xc4200e75b0; node [peripheries =
     ↪  1];
3    node [shape = point] START;
4    node [shape = point] xc4200e7800;
5    node [shape = point] xc4200e7810;
6    node [shape = ellipse];
7
8    xc4200e75b0 [label="f"]
9    xc4200e7470 [label="a"]
10   xc4200e74b0 [label="b"]
11   xc4200e74f0 [label="c"]
12   xc4200e7800 [label=""]
13   xc4200e7810 [label=""]
14   xc4200e7530 [label="d"]
15   xc4200e7570 [label="e"]
16
17   START -> xc4200e7470;
18   xc4200e7470 -> xc4200e74b0[ style=dotted];
19   xc4200e7470 -> xc4200e74f0;
20   xc4200e74b0 -> xc4200e74f0[ style=dotted];
21   xc4200e7530 -> xc4200e7570;
22   xc4200e7800 -> xc4200e7530;
23   xc4200e7570 -> xc4200e7810;
24   xc4200e7810 -> xc4200e7800[ label="2-4x"];
25   xc4200e74f0 -> xc4200e7800;
26   xc4200e7810 -> xc4200e75b0[ style=dotted];
27 }
```

Listing 88 Arguments for the `Fuzz` Command of the Tavor CLI

```
1  [fuzz command options]
2        --exec=                                     Execute this binary with possible arguments to test a generation
3        --exec-exact-exit-code=                     Same exit code has to be present (default: -1)
4        --exec-exact-stderr=                        Same stderr output has to be present
5        --exec-exact-stdout=                        Same stdout output has to be present
6        --exec-match-stderr=                        Searches through stderr via the given regex. A match has to be
          ↪ present
7        --exec-match-stdout=                        Searches through stdout via the given regex. A match has to be
          ↪ present
8        --exec-do-not-remove-tmp-files              If set, tmp files are not removed
9        --exec-do-not-remove-tmp-files-on-error     If set, tmp files are not removed on error
10       --exec-argument-type=                       How the generation is given to the binary (default: stdin)
11       --list-exec-argument-types                  List all available exec argument types
12       --script=                                   Execute this binary which gets fed with the generation and should
          ↪ return feedback
13       --exit-on-error                             Exit if an execution fails
14       --filter=                                   Fuzzing filter to apply
15       --list-filters                              List all available fuzzing filters
16       --strategy=                                 The fuzzing strategy (default: random)
17       --list-strategies                           List all available fuzzing strategies
18       --result-folder=                            Save every fuzzing result with the MD5 checksum as filename in this
          ↪ folder
19       --result-extension=                         If result-folder is used this will be the extension of every filename
20       --result-separator=                         Separates result outputs of each fuzzing step (default: "\n")
```

Listing 89 Arguments for the `Reduce` Command of the Tavor CLI

```
[reduce command options]
      --exec=                           Execute this binary with possible arguments to test a generation
      --exec-exact-exit-code            Same exit code has to be present
      --exec-exact-stderr               Same stderr output has to be present
      --exec-exact-stdout               Same stdout output has to be present
      --exec-match-stderr=              Searches through stderr via the given regex. A match has to be present
      --exec-match-stdout=              Searches through stdout via the given regex. A match has to be present
      --exec-do-not-remove-tmp-files    If set, tmp files are not removed
      --exec-argument-type=             How the generation is given to the binary (default: stdin)
      --list-exec-argument-types        List all available exec argument types
      --script=                         Execute this binary which gets fed with the generation and should return
       ↪ feedback
      --input-file=                     Input file which gets parsed, validated and delta-debugged via the format file
      --strategy=                       The reducing strategy (default: Linear)
      --list-strategies                 List all available reducing strategies
      --result-separator=               Separates result outputs of each reducing step (default: "\n")
```

Listing 90 Arguments for the `Graph` Command of the Tavor CLI

```
1 [graph command options]
2       --filter=         Fuzzing filter to apply
3       --list-filters    List all available fuzzing filters
```

Appendix C

Case Study: Coin Vending Machine

Listing 91 Semi Automated Delta-Debugging of Coin Vending Machine Case Study

```
1  credit  0
2
3  Do the constraints of the original input still hold for this
   ↪  generation? [yes|no]: no
4  credit  0
5  coin    50
6  credit  50
7  coin    50
8  credit  100
9  vend
10 credit  0
11
12 Do the constraints of the original input still hold for this
   ↪  generation? [yes|no]: no
13 credit  0
14 coin    50
15 credit  50
16 coin    25
17 credit  75
18 coin    25
19 credit  100
20 vend
21 credit  0
22
23 Do the constraints of the original input still hold for this
   ↪  generation? [yes|no]: yes
```

Listing 92 Partial Log of Applying Go-Mutesting for the Coin Vending Machine Case Study

```
1 PASS "/tmp/go-mutesting-
  ↪ 291931067//home/zimmski/go/src/coin/implementation.go.0" with
  ↪ checksum 59ff7a970964f4a5bdaaac92d09b8600
2 PASS "/tmp/go-mutesting-
  ↪ 291931067//home/zimmski/go/src/coin/implementation.go.1" with
  ↪ checksum 9bc8a0e74dd28e7377aec58c336d0852
3 --- /home/zimmski/go/src/coin/implementation.go 2017-11-22
  ↪ 11:33:03.020149906 +0100
4 +++ /tmp/go-mutesting-
  ↪ 291931067//home/zimmski/go/src/coin/implementation.go.2
  ↪ 2017-11-22 11:39:25.736642702 +0100
5 @@ -39,7 +39,8 @@
6         case coin50:
7                 v.credit += credit
8         default:
9 -               return ErrUnknownCoin
10 +              _ = ErrUnknownCoin
11        }
12        return nil
13 FAIL "/tmp/go-mutesting-
  ↪ 291931067//home/zimmski/go/src/coin/implementation.go.2" with
  ↪ checksum 355ad77c0828056fde05b28d923e440c
14 --- /home/zimmski/go/src/coin/implementation.go 2017-11-22
  ↪ 11:33:03.020149906 +0100
15 +++ /tmp/go-mutesting-
  ↪ 291931067//home/zimmski/go/src/coin/implementation.go.3
  ↪ 2017-11-22 11:39:26.040652417 +0100
16 @@ -48,7 +48,7 @@
17  // Vend executes a vend of the machine if enough credit (100) has
    ↪ been put in and returns true.
18  func (v *VendingMachine) Vend() bool {
19        if v.credit < 100 {
20 -              return false
21        }
22        v.credit -= 100
23 FAIL "/tmp/go-mutesting-
  ↪ 291931067//home/zimmski/go/src/coin/implementation.go.3" with
  ↪ checksum 9bf76c4dd255f2c388659111af37a790
24 PASS "/tmp/go-mutesting-
  ↪ 291931067//home/zimmski/go/src/coin/implementation.go.6" with
  ↪ checksum a42a707de248e2cda066676e24484c6b
```

List of Figures

2.1 Fundamental Components of Fuzzing 8
2.2 Fundamental Components of Model-Based Testing 10
2.3 Fundamental Components of Mutation Testing 13
2.4 Fundamental Components of Delta-Debugging 14

3.1 Tavor's Subsystems 17
3.2 Components of the Tavor Framework 18
3.3 Example for a Graph With a Loop 20
3.4 Example for an Unrolled Graph 20
3.5 Automaton of Simple Format Definition 27
3.6 Token Graph of Simple Format Definition 28
3.7 AllPermutations Example Token Graph 33
3.8 Incrementing in the Decimal Numeral System 33
3.9 AllPermutations Iterations 35

5.1 Workflow for the `Graph` Command of the Tavor CLI 64
5.2 Example Graphics for the `Graph` Command 67
5.3 Workflow for the `Fuzz` Command of the Tavor CLI 68
5.4 Fuzzing of External Programs 68
5.5 Workflow for the `Validate` Command of the Tavor CLI 69
5.6 Workflow for the `Reduce` Command of the Tavor CLI 70

6.1 Architecture of Go-Mutesting 74

7.1 Example Automaton of the Coin Vending Machine 80
7.2 Code Coverage Example 94

List of Listings

1 Smiley TAVOR FORMAT . . . 21
2 Smiley Data Structure . . . 22
3 Smiley `String` Method . . . 22
4 Smiley `Clone` Method . . . 23
5 Smiley `Permutations` Method . . . 23
6 Smiley `PermutationsAll` Method . . . 23
7 Smiley `Permutation` Method . . . 24
8 Smiley `Parse` Method . . . 25
9 Working With the SMILEY Token . . . 26
10 Output of Smiley Example . . . 26
11 Token Graph Walk Function . . . 29
12 Sample Fuzzing Strategy . . . 31
13 Callee of Example Fuzzing Strategy . . . 32
14 Command Line Output of Example Fuzzing Strategy . . . 32
15 Registering the Sample Fuzzing Strategy . . . 33
16 Permutation of AllPermutations Example Token Graph . . . 34
17 Sample Filter . . . 36
18 Registering the Sample Filter . . . 36
19 Auxiliary Function ApplyFilters . . . 37
20 Reduce String Token Repetitions . . . 40
21 Callee of Example Reducing Strategy . . . 41
22 Command Line Output of Example Reducing Strategy . . . 42
23 Registering the Sample Strategy . . . 42
24 Linear Reducing Strategy . . . 43
25 Traversing of the Token Graph . . . 44
26 Performing the Reduction . . . 44

27 TAVOR's Hello World . . . 45
28 Example for a Token Concatenation . . . 46
29 Example for a Multi-Line Token Definition . . . 46
30 Example for Different Kinds of Comments . . . 46
31 Example for a Number Token . . . 47
32 Example for a String Token . . . 47
33 Example for Embedding Tokens . . . 48
34 Example for Terminal and Non-Terminal Tokens Mixed in One Format . . . 48

35 Example for a Token Reference and a Token Usage 48
36 Example for the Alternation Token 49
37 Example for Loop Using an Alternation 49
38 Example for a Group Token 49
39 Example for Nested Groups 50
40 Example for an Optional Group 50
41 Example for a Repeat Group 50
42 Example for a Fixed Repeat Group 51
43 Example for a Ranged Repeat Group 51
44 Example for an Empty "from" Argument for a Ranged Repeat Group . . 52
45 Example for an Empty "to" Argument for a Ranged Repeat Group . . . 52
46 Example for an Optional Repeat Group 53
47 Example for a Permutation Group 53
48 Example for a Character Class Token 54
49 Example for Escape Characters in Character Classes 54
50 Example for Hexadecimal Code Points in Character Classes 55
51 Example for a Character Class Range 55
52 Example for a Token Attribute Using the "Count" Attribute 56
53 Example for Attribute Parameters Using the "Item" Attribute 56
54 Example for Scopes 57
55 Example Typed Token Using an Integer Token 57
56 Example for Typed Token Attributes 57
57 Example Expression 57
58 Example for a Token Attribute Inside an Expression 57
59 Example for Include Operator 58
60 Example for Path Operator 59
61 Example for the Not in Operator 59
62 Example for a Token Variable 60
63 Example for the If Statement 61

64 Convert `Token Graph` to Simple Graph Structure 65
65 Example TAVOR FORMAT for the `Graph` Command 67
66 Delta-Debugging Pseudo Code for Scripts 71

67 Functionality of Package `Keydriven` 81
68 Interface of the Coin Vending Machine Module 82
69 Example Executor for a Coin Vending Machine 84
70 TAVOR FORMAT for Coin Vending Machine 85
71 Key-Driven File for the Coin Vending Machine Test Scenario 85
72 Key-Driven Test Script 85
73 Key-Driven Test Output 88
74 Fully Automated Delta-Debugging for Coin Vending Machine 89

75 Tavor Format Denoting the AIGER ASCII Format Part One 90
76 Tavor Format Denoting the AIGER ASCII Format Part Two 91
77 Pseudo Code for the Script for Performing the AIGER Evaluation . . . 95
78 Tavor Format Denoting the JSON Format 99
79 Tavor Format Denoting a Minimum of the JSON Format 101
80 Test Function for Generated JSON Data 102
81 Mutant Killed by Tavor Part One 104
82 Mutant Killed by Tavor Part Two 104

83 Pseudo Code of AllPermutations Fuzzing Strategy 111
84 Pseudo Code of AllPermutations Fuzzing Strategy Helper Function . . . 112

85 General Arguments of the Tavor CLI 113
86 Arguments for `Validate` the Command of the Tavor CLI 113
87 Example DOT Format for the `Graph` Command 114
88 Arguments for the `Fuzz` Command of the Tavor CLI 115
89 Arguments for the `Reduce` Command of the Tavor CLI 116
90 Arguments for the `Graph` Command of the Tavor CLI 117

91 Semi Automated Delta-Debugging of Coin Vending Machine Case Study 119
92 Partial Log of Applying Go-Mutesting for the Coin Vending Machine Case Study 120

List of Tables

3.1 Computing PermutationsAll for Different Token Types 32

4.1 Escape Characters for Character Classes 51
4.2 Special Escape Characters for Character Classes 52
4.3 Token Attributes for List Tokens 53
4.4 Optional Token Arguments for the "Int" Typed Token 55
4.5 Token Attributes for the "Int" Typed Token 55
4.6 Optional Token Arguments for the "Sequence" Typed Token 55
4.7 Token Attributes for the "Sequence" Typed Token 56
4.8 Arithmetic Operators 58
4.10 Operators for the If Statement Condition 60
4.9 Token Attributes of Variable Tokens 60

7.1 Comparison of the Test Set Generation of the AIGER Evaluation 95
7.2 Comparison of the Commands of the AIGER Evaluation 96
7.3 Comparison of the Test Set Execution of the AIGER Evaluation 97
7.4 Comparison of the Test Set Generation of the JSON Evaluation 103
7.5 Comparison of the Test Set Results of the JSON Evaluation 103

Bibliography

[1] A complete example of TAVOR @ONLINE (2017), https://github.com/zimmski/tavor/blob/master/doc/complete-example.md

[2] The aigfuzz fuzzer for the AIGER ASCII format @ONLINE (2017), https://github.com/johnyf/aiger_tools/blob/master/aigfuzz.c

[3] The LLVM source-based code coverage documentation @ONLINE (2017), https://clang.llvm.org/docs/SourceBasedCodeCoverage.html

[4] The GO-MUTESTING source code repository @ONLINE (2017), https://github.com/zimmski/go-mutesting

[5] The GO programming language specification @ONLINE (2017), https://golang.org/ref/spec

[6] The GOLANG-MUTATON-TESTING source code repository @ONLINE (2017), https://github.com/StefanSchroeder/Golang-Mutation-testing

[7] The MANBEARPIG source code repository @ONLINE (2017), https://github.com/darkhelmet/manbearpig

[8] The MUTATOR source code repository @ONLINE (2017), https://github.com/kisielk/mutator

[9] Biere, A.: The AIGER And-Inverter Graph (AIG) Format Version 20070427. Tech. rep., Johannes Kepler University (2007)

[10] Brummayer, R., Biere, A.: Fuzzing and delta-debugging SMT solvers. In: Proceedings of the 7th International Workshop on Satisfiability Modulo Theories. pp. 1–5. ACM (2009)

[11] Brummayer, R., Lonsing, F., Biere, A.: Automated testing and debugging of SAT and QBF solvers. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 44–57. Springer (2010)

[12] DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: Help for the practicing programmer. Computer 11(4), 34–41 (1978)

[13] Fewster, M., Graham, D.: Software test automation. Addison-Wesley Professional (1999)

[14] Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE transactions on software engineering 37(5), 649–678 (2011)

[15] McNally, R., Yiu, K., Grove, D., Gerhardy, D.: Fuzzing: the state of the art. Tech. rep., DTIC Document (2012)

[16] Miller, B.P., Koski, D., Lee, C.P., Maganty, V., Murthy, R., Natarajan, A., Steidl, J.: Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Tech. rep., Technical Report CS-TR-1995-1268, University of Wisconsin (1995)

[17] Miller, C., Peterson, Z.N.: Analysis of mutation and generation-based fuzzing. Independent Security Evaluators, Tech. Rep (2007)

[18] Misherghi, G., Su, Z.: HDD: hierarchical delta debugging. In: Proceedings of the 28th international conference on Software engineering. pp. 142–151. ACM (2006)

[19] Sutton, M., Greene, A., Amini, P.: Fuzzing: brute force vulnerability discovery. Pearson Education (2007)

[20] Takanen, A., Demott, J.D., Miller, C.: Fuzzing for software security testing and quality assurance. Artech House (2008)

[21] Utting, M., Legeard, B.: Practical model-based testing: a tools approach. Morgan Kaufmann (2010)

[22] Zeller, A.: Why programs fail: a guide to systematic debugging. Elsevier (2009)

Vollständiger Lebenslauf auf Anfrage
markus.zimmermann@symflower.com

Lebenslauf

Name Markus Zimmermann, B.Sc.

Wohnort Linz

Auswahl an Erfahrungen

Seit 02/2015	**Firma:** Symflower (in Gründung), Linz **Position:** Geschäftsführer
04/2012 - 02/2017	**Firma:** nethead - Markus Zimmermann EU, Linz **Position:** Freelancer und Consultant
04/2011 - 01/2017	**Firma:** Software Quality Lab GmbH, Linz **Position:** Senior DevOp, Senior Test und Security Consultant
10/2008 - 03/2012	**Firma:** nethead - Antipa & Zimmermann GesbR, Linz **Position:** Geschäftsführer
05/2008 - 09/2008	**Firma:** Atikon EDV & Marketing GmbH, Linz **Position:** Software Engineer
2003 - 2008	**Position:** Freelancer
07/2005 - 09/2005	**Firma:** STIWA Fertigungstechnik Sticht GmbH, Attnang-Puchheim **Position:** Praktikant

Ausbildung

Seit 2013	Masterstudium Informatik an der Johannes Kepler Universität, Linz **Master Thesis:** "Tavor - A Generic Fuzzing and Delta-Debugging Framework"
2008 - 2013	Bachelorstudium Informatik an der Johannes Kepler Universität, Linz (mit Auszeichnung abgeschlossen) **Bachelor Thesis:** "Tirion - A Complete Infrastructure for Monitoring Applications during Benchmarks"
2007 - 2008	Bundesheer
2002 - 2007	HTL für EDV und Organisation, Grieskirchen (mit ausgezeichnetem Erfolg abgeschlossen) **Diplomarbeit:** "ALOIS - Verwaltungssoftware für die HTL-Grieskirchen"

Linz, am 26.03.2017

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, am

Markus Zimmermann, BSc