Stephan Valentan

A Comparison of Combinatorial and Random Testing for Audio Processing Software

Master's Thesis

to achieve the university degree of
Master of Science

submitted to
Graz University of Technology

Supervisor
Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa

Institute for Softwaretechnology
Head: Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa

Factulty of Computer Science and Biomedical Engineering

Graz, April 2021

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date	Signature

Abstract

The importance of testing in software development is increasing, therefore the availability of automatic testing solutions is crucial. However, little work has been conducted in the domain of digital signal processing and audio software development, and thus the focus of this work lies on testing audio plugins used in digital audio workstations. Because professional-grade audio plugins are developed for multiple operating systems and against multiple SDKs, while also providing complex input models, manual regression testing is not viable.

This thesis introduces an automatic regression testing tool for audio plugins, intended to support developers in the course of the continuos integration cycle. The tool supports automatic test case generation using combinatorial or random strategies, and performs automatic regression testing by comparing the rendered audio output of two different versions.

During evaluation, the effectiveness of combinatorial and random testing is assessed by applying the tool to mutated versions of real-world audio plugins. It is shown that for this particular problem, no relevant differences in fault-detecting capabilities of the two strategies can be observed. The results also show that random testing needs less tests to reach the maximal mutation score.

After collection of code metrics in the audio plugins it is found that the combinatorial complexity of the DSP code is rather low, explaining the good results that are obtainable with random testing. While more work on the complexity of DSP code and the fault-detecting capabilities of combinatorial testing in audio plugins is necessary, this work provides first insights on the applicability of combinatorial testing in this area of software development.

Acknowledgements

First and foremost, I want to thank Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa for the excellent supervision and guidance during the course of this thesis.

Furthermore, I thank my employer, sonible GmbH and Dipl.-Ing. Peter Sciri in particular for their support and flexibility.

I am also grateful for the support from my parents and family during my studies, as well as the many opportunities that were enabled by them.

Finally, Hannah, thank you for the many fruitful discussions and cups of coffee we had together, and for motivating and supporting me in the last years and during the time of the thesis.

Graz, April 2021 Stephan Valentan

Contents

List of Figures

List of Figures

List of Tables

1 Introduction

In this section, the motivation of this thesis and the problem statement is given, the thesis contribution is stated and the overall structure of the work is described.

1.1 Problem Statement

The importance of testing in the field of digital signal processing (DSP) software is, analogue to the majority of fields in software development, on the rise [22]. As processing hardware becomes more capable every year, software that is making use of these systems also becomes more complex, resulting in feature-rich products for end-users. This is supported by ever-improving development tools, continuous-integration servers, collaboration services and comprehensive coding frameworks. However, the increase in innovation has not gone by without consequences in the context of quality assurance in general, and testing in particular. It has been found that testing accounts for a large percentage of the overall complexity of a project [11].

Due to the shift from traditional management techniques to more flexible and iterative solutions, the need for automating large parts of the testing process is more important than ever. Because traditional management styles assume that requirements hardly change over the course of the project, ad-hoc testing at the end of the implementation of the product is often considered sufficient. This premise hardly holds up in real-world software projects. Subsequently, research shows that many projects are delayed or canceled altogether [28]. To cope with this, iterative development styles consider the possibility for frequent change in the projects requirements by heavily relying on testing [24]. As the need for constant testing requires a considerable amount of resources, more automated approaches than ad-hoc testing are necessary. The research community therefore provides valuable solutions for a variety of problems, with examples including *Combinatorial Testing*, *Symbolic Execution*, *Random Testing*, *Search-Based Testing* and *Model-Based Testing*. However, all these techniques, possibly with the exception of model-based testing, exhibit one fundamental flaw: They are not capable of automatically providing ground-truth information.

While this so-called *Oracle problem* is equally important for the automation of testing, far less research has been conducted on this topic [8]. It becomes clear that formal knowledge about the systems expected behaviour must be available, in order to automatically generate the required ground-truth data. This is the approach taken in model-based testing (MBT), where both the test cases and the expected outputs are derived from a formal model description [4]. However, by adopting this technique, instead of investing resources in writing tests, more resources are needed for developing and maintaining the model itself. This task may in turn introduce errors or inconsistencies, and not all systems may be expressible via such a model. Research suggests that MBT is mainly applicable for systems based on structures, e.g. final-state-machines or label-transition-systems [5]. For these kinds of systems, MBT has proven to be useful not only in the research community, but also in industry projects. While other useful strategies and tools exist, developers are still required to invest valuable resources for defining the properties of the system first, in order to profit from test automation.

Developers of DSP and audio software face even more problems. Since this application domain is highly mathematical, even unit testing can be impractical to conduct. In many cases, the result of a logical unit, which could be easily covered by unit tests in other domains, is not more than a matrix filled with floating point numbers. While there are situations in which even these results may be tested thoroughly, e.g. when testing a well-known algorithm, in most cases such a matrix would not be informative to the human eye. This increases the maintenance costs of the tests and also reduces the documentation value that tests hold. Although unit testing is undoubtedly important for assessing the correctness of logical problems, it can be argued that it is not suited for testing the majority of DSP code.

However, in research only a limited amount of work has been conducted in the context of testing DSP software in general, and audio software testing in particular. Even the simplest approach of ad-hoc testing is of limited usefulness for audio development. While obviously the perceived characteristics of sound produced by the software are important, many of the subtle differences may not be recognizable for human ears. As the number of different configurations of the software is in many cases high, it is often not possible to reach a satisfactory confidence level using ad-hoc testing. This is especially true in the case of audio *plugin* development, where a single product may be used in numerous different hosting programs, multiple operating systems and up to four different plugin standards. In the face of these problems, other methods of assuring the quality of software products in this domain need to be found.

1.2 Thesis Contribution

Tackling the problems described in the last section, this thesis provides three distinct contributions for supporting the testing process.

1.2.1 Audio Plugin Regression Tool

Firstly, this thesis introduces an automatic audio regression testing tool for audio plugins. This multi-platform tool supports three of the four major plugin standards and is capable of performing regression testing across operating system bounds by using a built-in server mode. It implements two different test case generation strategies, namely combinatorial testing and random testing. This design choice ensures that as little resources as possible are necessary for integrating the tool into the testing process. The tool also includes an automated test oracle, which further reduces the resources needed for adopting the tool in a continuous integration cycle.

1.2.2 Comparison of Combinatorial and Random Testing

Due to the fact that little research is available about testing strategies of DSP code, this thesis conducts experiments with three real-world plugins, two of which are considered professional-grade. The experiments address the question whether combinatorial test case generation is beneficial compared to random test case generation. For conducting the experiments, a number of mutations are introduced into the signal processing code of the plugins. Using a number of different input models, as well as the two supported test generation strategies, it is then assessed whether or not the mutation is detected by the tool.

1.2.3 Complexity Analysis of DSP Code

Finally, by analyzing the source code of the plugins used in the experiment, useful metrics about the logical complexity of the signal-processing code are collected. These results support the findings of the test case generation comparison provided by this thesis. Using the metrics it may be possible to make informed decisions about the testing setup that could be employed on future projects in the context of audio development.

1.3 Thesis Structure

The remainder of this thesis is structured as follows. In Section 2, background information about the testing process and audio plugin development is presented. Section 3 discusses the design choices, implementation, and use of the regression testing tool itself. An evaluation of the testing tool, as well as a comparison of the test case generation strategies is given in Section 4. This section also provides metrics about the source code complexity of the plugins used during the course of the evaluation. A discussion about the results of the evaluation is given in Section 5, followed by the conclusion in Section 6.

2 Background

In this section, relevant technologies and terms which are used throughout the thesis are introduced. Specifically, a brief history of the evolution of the software development cycle and software testing is given. Next, the term *Testing* itself and its variants are described, including strategies for test case generation. Furthermore, the so-called *Oracle Problem* and proposed solutions are addressed. Finally, an overview of the Digital Audio Processing ecosystem is given.

2.1 Software Development Life Cycle

In this section, the history of the software development life cycle (SDLC) is presented. The SDLC denotes the process involved with planning, creating, testing, and maintaining a piece of software, and hence covers the whole lifetime of a software product. From the early days of software development on, one of the main project management techniques has been the *Waterfall-Model* and, to a lesser extend, the *V-Model*. Despite the heavy use of these traditional approaches in other industries, it has been found that only 34% of software projects have been completed successfully [28]. This revealed the shortcomings of the traditional approaches and led to a change in mind of software project executives. It has been found that the static succession of project phases is not reflecting the real-world processes very well. To counteract these shortcomings, more flexible solution have been proposed, giving rise to the idea of *Agile Programming* [24]. One of the key realizations of the agile methods is, that it seemed to be very difficult for customers to concisely communicate their needs, and furthermore to translate the customers problems into technical terms. Therefore, the underlying concept of all agile methods is to develop the project in an iterative manner, instead of passing through all project stages linearly. With the traditional approaches, a customer would only see the final product after all phases of the project are completed. This bears a great risk, since it has been found that changes become increasingly expensive as the project is being implemented, and delays in the project completion become more likely [47]. However, because customer needs are often only formulated vaguely, miscommunication is relatively likely, and project requirements may change during the implementation of the project. Therefore, the presentation of the final product to the customer at such a late stage can be considered risky. Agile approaches try to mitigate this fact by iteratively developing a product, and presenting the current state of the project to the customer on a regular basis. This has proven to have benefits for both sides: The customer can closely monitor the progress of the product and point out misconceptions early on, increasing confidence in the product and transparently outlines the projects progress. On the other hand, for software developers it is possible to reduce the risk of disappointing the customer at the end of the development process, since the continuously held meetings and small releases require the customer to approve the direction and state of the project on a regular basis. It can be argued that this method embraces dialogue, improves customer satisfaction and the chance of future cooperation, while the risk of litigation and even development staff dissatisfaction is reduced at the same time [16] [40]. As a result, studies have found that software projects using agile methods are less likely to fail, which is a reason for the continuing adoption of these techniques in modern software development [26] [40].

In the following sections both the traditional and modern approaches to software project management are described in greater detail.

2.1.1 Traditional Strategies

In the following, the most noteworthy approaches, the *Waterfall-Model* and *V-Model* are explained.

The traditional waterfall model as presented in [9] consists of five distinct phases. Each successor phase may only start once the previous phase is completed. The model proposes a linear advance, it is discouraged to re-open a previously completed phase. At the beginning of a software project stands the *Requirement Phase*, at which all functional and non-functional requirements of the projects are collected in accordance with the customer. As a result of this stage, a requirement document is created, which, in the best case, stays valid for the remainder of the project. As a second step, in the *Design Phase*, software engineers and system architects translate the requirements collected in the previous stage into technical terms and draft the design and architecture of the resulting software. This is the basis for the third stage, the *Implementation Phase*, where the requirements are implemented with the agreed-upon design and architecture. In the traditional workflow, this is followed by the *Testing Phase*, where functional and non-functional tests are performed to ensure that the requirements set by the customer are met. The last test that is performed is usually the *Acceptance Test*, which ultimately determines if the customer is satisfied with the delivered product in general. At the end of the test phase stands the release of the product, which at the same time marks the beginning of the final *Maintenance Phase*. This phase describes all activities regarding updates and bugfixes, which are performed after the product has already been shipped to end-users. Because the earlier phases of the project, especially the implementation phase, are often delayed, while the release date is fixed, in many cases the testing is cut short. This results in a reduction in software quality and increases maintenance costs later on.

The *V-Model* can be seen as an extension to the waterfall-model, originating around 1990 [38]. Its main benefit is to emphasize the importance of testing that is required for advancing through the stages. Instead of just describing a monolithic testing stage like the waterfall model, it clearly states that different types of testing are required to ensure that all requirements are met. Furthermore, it proposes that testers are involved in the project early on, in order to develop testing strategies before the implementation phase starts. However, it faces the same challenges as the waterfall-model, as its similarly inflexible and does not provide an effective strategy for dealing with problems during any later phase in the development process [12].

2.1.2 Agile Development

In opposite to the static and linear approach that the traditional project management strategies follow, agile approaches are, as the names already suggests, more flexible. In fact, it is hard to summarize precisely what the term *agile* means in this context, as a multitude of different realizations of this concept have been developed over the years. However, all of these strategies share a common goal, which is to reduce the risk of a project failing by introducing shorter feedback loops. Since traditional management approaches only contain one feedback cycle, the customer is informed about the product at the end of the testing phase. Therefore, by regularly showing and discussing the current state of the project with the customer, agile strategies try to reduce the probability of misunderstandings and miscommunication. The key concepts of the agile development are captured in the so-called *Agile Manifesto*, formulated by software experts in 2001 [21]. Among other things, it emphasizes the following four points:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

The manifesto states that both sides are valuable, but when in doubt, it makes clear which concept should be preferred. In the following three widely-used variants of the agile development movement are described [1].

- Kanban: The Kanban strategy is one of the more lightweight management strategies. It primarily tries to provide means for organizing the issue backlog in projects, while respecting established responsibilities within the team.
- Scrum: While Kanban only focuses on the way the work is visualized, Scrum also defines roles and processes within the team. If following the Scrum technique, the performance (or *velocity*) of a team is optimized over a number of development cycles (or *sprints*). In order to do so, the team engages in multiple mandatory meetings, where the performance of the previous sprints is discussed, and the next sprint is planned

[1]https://manifesto.co.uk/kanban-vs-scrum-vs-xp-an-agile-comparison/, accessed 19.2.2021

together with the team. Scrum additionally defines roles within the team, most prominently the *Scrum Master*, which is responsible for maintaining and coordinating the process itself, as well as the *Product Owner*, which is acting as a representative of the customer within the development team.

- Extreme Programming: The Extreme Programming (XP) variant has been developed by Beck in 1996 [33]. It proposes that best-practice techniques in software development should be taken to the extreme, resulting in a low-risk management strategy. While it also defines the visualization of issues and introduces several mandatory roles within the team, it also requires the team to follow a number of development guidelines, e.g. pair-programming or test-driven development.

2.2 Software Testing

With the rise of agile software development, new ways of managing tests needed to be introduced. When previously manual testing strategies like ad-hoc testing were sufficient, the short iterations required more automated testing techniques [50]. In this section, the term of *Software Testing* and several different categories of software testing are described. A brief history of software testing is given, focussing on the changes in perception and priority of testing with the transition from traditional software management strategies to more modern, agile approaches.

The definition of software testing has evolved over the decades. In the early days, testing has been indistinguishable from debugging. However, Alan Turing realized that better means to analyze whether programs are exhibiting the desired behaviour are necessary. He proposed an operational test, which should determine whether human and program behaviour are similar, given the same task. In later years, researchers' interest in software testing theory increased, and more pronounced concepts and differentiations were developed. Notably, the difference between debugging and testing was defined more precisely. It was stated that the goal of testing is to show the absence of errors, while debugging is the action of eliminating errors that have been discovered previously. Still, in these days testing was seen as a

Name	Purpose
Unit	Test isolated logical unit, no dependencies
Integration	Test logical unit with its collaborators, real world scenario
System	End-to-End tests of whole system, real world scenario
Acceptance	Customer approval of the product
Regression	Test if a change broke existing behaviour

Table 2.1: Goals of different testing techniques.

stage following the implementation for detecting implementation faults. It was only in later years after 1983 [34], that the ability of testing to detect not only mistakes made in the implementation phase, but in all previous phases, was discovered. This phase of the software testing evolution was later called *Evaluation-Oriented* [22]. However, in all evolution stages before 1988, software testing was mainly seen as a way of detecting faults. This changed when entering the current phase, which instead focuses on preventing faults. In modern days, the main goal of testing is seen as preventing any faults to even reach the product in the first place.

2.2.1 Software Testing Distinctions

As the professional focus on software testing grew, a number of different software testing distinctions were established, a few of which are presented in the following. The differences between this distinctions concern the focus of the testing technique, its collaborators, the purpose and the point in time when the testing is conducted. These properties can be seen in Table 2.1.

Unit Testing

Unit testing describes the process of assessing the correctness of a single logical unit within a program. As a rule of thumb, a unit test should be written as soon as a logical block of code exhibit non-trivial behaviour. A typical candidate for a unit test could be a single function in procedural programming languages, or a method of a class in object-oriented languages.

It is important to note, that unit tests should be independent from other functionality provided by the program, as well as being independent from environment changes. This means that they should be reliable and repeatable, regardless of the context they are run in. This can be achieved via so-called *Mocking* [35], where external dependencies, e.g. HTTP-requests to an external server, are replaced with simple, reproducible, and fast counterpart implementations. Doing so eliminates the risk of a unit test failing because of some external failure, e.g. an unstable internet connection. As a side-effect, this also improves the performance of unit tests, which is crucial for certain techniques such as *Test-Driven-Development*.

Integration Testing

While unit testing focuses on a single unit, integration testing is applied to a number of modules that interact with each other. Depending on the context, integration tests can also make use of real-world data and assess whether the system performs correctly, even in the presence of external dependencies. In opposite to unit testing, integration testing may also be used to test the compliance of the software with non-functional requirements, e.g. performance criteria. Partly because of the external dependencies, integration tests usually run slower than unit tests, and hence they will not be run as frequently as unit tests.

System Testing

Also known under the term *End-to-End testing*, system testing examines a programs behaviour from an user-centric standpoint. Instead of focusing on certain parts of the source code, its primary goal is to model how users or clients of the software will interact with the system, and assess whether the corresponding output is correct. Due to this fact, end-to-end tests can also be modeled as so-called *Black-Box-Tests*, where no knowledge of the actual implementation is necessary.

Regression Testing

Regression testing is complementary to the other testing techniques. While it makes use of tests already written, e.g. unit-, integration- or system-tests, it serves a different purpose. Instead of assessing the correctness of the newly written code, it determines if the already-existing functionality still performs as intended. This means that regression testing is performed after integrating new functionality into the existing codebase. In many cases when developing new functionality that is backed up with unit tests, a developer would only run the relevant tests for the modified sections of code, speeding up the development process. However, with regression testing all available tests are run, in order to catch errors that were missed or caused by non-trivial interactions.

Acceptance Testing

In opposite to the testing techniques described in the previous chapters, acceptance testing is performed by the customer, rather than the development team. It is determined, if the software delivered to the customer meets the functional and non-functional requirements that were put down and agreed upon in the contract.

2.3 Test Case Generation

While it has been widely accepted that software testing is an integral part of the development process, it still poses a significant challenge on development teams. It has been found that testing takes up as much as 50% of a projects budget [3], resulting in increasing interest of the research community. This chapter describes ways for improving the testing process by automatically generating test cases, instead of requiring developers to invest valuable resources for manually creating tests.

2.3.1 Model-based Testing

The goal of model-based testing (MBT) is to derive test cases from an abstract, but formal model specification. An example for such a specification is a final state machine (FSM), where the system is defined by states, state changes, inputs, and outputs. Using a formal model and user-defined test requirements, abstract test cases can be derived. For a FSM this could be a sequence of state changes, that should result in a defined sequence of outputs. The next step is to translate the generated abstract test cases into the systems language, such that they can be run against the system under test (SUT). Depending on the modeling language, the so-called *concretion* of abstract test cases can, in some cases, be performed by simple text substitution. This results in a test suite at the same abstraction level as the SUT, therefore the tests can be run against the SUT directly [1]. Although the use of MBT may provide benefits to software projects [4], its use in the industry poses some challenges. Foremost, the formalization of requirements into a model is a highly complex task that requires special care. If the model is incorrect, no valid test cases can be derived. Furthermore, it is necessary to choose the correct level of abstraction for the model: If the model is too abstract, MBT may not be able to detect implementation errors in the SUT. On the other hand, if the model is too concrete, the effort required to validate the model becomes increasingly similar to validating the SUT directly, resulting in only limited benefits.

2.3.2 Search-based Testing

Search-based testing is a method to automatically and iteratively generate test data. Instead of relying on a model, a so-called *fitness function* is used to determine whether a test should be included in the final test suite. Over multiple iterations and by making use of the fitness function, this approach searches for tests that maximize the fitness function. The implementation of the fitness function depends on the application, a common metric is to maximize the branch coverage. One promising approach to generate test data is the field of genetic algorithms, which is inspired by the naturally occurring processes of evolution. Genetic algorithms start with an initial test suite, in this context called *population*, which contains test cases called *individuals*. The population at a specific iteration is called *generation*. Starting with the initial population, each individual is subject to a number of transformations, commonly used are the *mutation* and the *crossover* operations. When performing a mutation operation, parts of an individual are modified, while keeping the rest of the individual unchanged. For a crossover, two distinct individuals, called *parents* in this case, are used to form two *child* individuals. These children then consist of parts of both parents' information, furthering the diversity of the population as a whole. After applying these or more transformations, a selection process takes place, where only a limited number of individuals featuring the highest fitness scores are kept. This process is then repeated, until a certain number of iterations, or a previously-defined fitness score is reached. Search-based approaches in general and genetic algorithms in particular have been successfully applied to a variety of problems, including image processing and environmental research [51]. However, it is notable that all search-based optimization approaches use probabilistic rather than deterministic rules, therefore they might not be applicable in certain fields [37].

2.3.3 Combinatorial Testing

In fields where probabilistic rules, like the ones used in search-based testing, are undesired, one can turn to combinatorial testing (CT). CT focuses on testing interactions of parameter [45]. While it is often unfeasible to test all parameter-value combinations, CT can provide a tradeoff between efficiency and completeness. This approach employs a sampling mechanism to reduce the problem space, and generates combinations of sampled values as input for test cases. With this sampling and by cleverly combining individual parameter-value combinations into a single test case, advanced algorithms are, to a certain extent, able to avoid the test case explosion. In modern, real-world software it is unfeasible to exhaustively test all interaction combinations, since the number of test cases grows exponentially. Therefore, combinatorial testing provides means for covering scenarios that have a higher probability of causing an error. Studies have found that most programming errors are triggered by a combination of 6 or less distinct parameter-values, and that 2 distinct parameter-value combinations are already covering as much as 60% of programming errors [29]. Therefore it is unpractical, inefficient, and unnecessary to exhaustively covering all combinations, and the focus on lower parameter combination degrees is sufficient in many cases. Having a systematic measure of generating test cases is especially important in highly safety-critical fields of the industry, e.g. in aircraft development. Therefore, a project conducted by Lockheed Martin alongside the National Institute of Standards and Technology (NIST) found, that the introduction of combinatorial testing mechanisms improved test coverage, while at the same time reducing development costs [25]. The paper also claims that adoption of CT strategies is likely to to increase, since the technique has been introduced into the Software Testing Standard ISO/IEC/IEEE 29119 [2]. The US government agency NIST has been a driving force behind the development of CT. In 2013 a tool called *Advanced Combinatorial Testing System* (ACTS) is published by NIST, implementing several algorithms for generating covering arrays, which provide the basis for combinatorial testing. NIST also claims that some of the algorithms developed by the institute, for example *IPOG* and its variants, are more efficient than previously known algorithms for test case generation in this context. In the following, the mathematical foundations of covering arrays and combinatorial testing are explained in greater detail.

Covering Arrays

Covering arrays are connected to the definition of *Orthogonal Arrays*. An orthogonal array $OA_\lambda = (N; t, k, v)$ is a $N \times k$ 2-dimensional array, where t is called the *strength*, k is named *factor*, and v is called the *order* and specifies the number of levels per factor. The definition states that for every $N \times t$ sub-array, each t-tuple must occur exactly λ times. In testing, usually only the case of $\lambda = 1$ is considered. Table 2.2 displays an orthogonal array $OA_1 = (4; 2, 3, 3)$. Notice that all 2-tuples aa, ab, ba and bb are present exactly once, regardless which two columns are considered. Therefore the definition of the orthogonal arrays is satisfied.

However, in practical settings the requirement that each tuple is present exactly the same number of times proved to be too strict. Therefore the

a	a	a
b	b	a
a	b	b
b	a	b

Table 2.2: Orthogonal array $OA_1 = (4; 2, 3, 2)$. Each 2-tuple *aa*, *ab*, *ba*, *bb* is present exactly once when considering any two columns.

a	a	a
b	b	a
a	b	b
b	a	b
a	b	a

Table 2.3: Covering array $CA_1 = (5; 2, 3, 2)$. Each 2-tuple *aa*, *ab*, *ba*, *bb* is present *at least* once when considering any two columns.

relaxed notation of *Covering Arrays* $CA_\lambda = (N; t, k, v)$ was introduced. With covering arrays, each t-tuple needs to be present *at least* a λ times, making clear that an orthogonal array is a special case of a covering array. Table 2.3.3 depicts a covering array $CA_1 = (5; 2, 3, 3)$. The *covering array number* $CAN(t, k, v)$ specifies the minimum number of test cases (rows) in a covering array that are necessary for fulfilling the formal definition. They can be seen as the lower bound of test cases. Furthermore, there exists a more general form of the covering array that is relevant in practice, called the *mixed level covering array*.

In normal covering arrays, all factors have the same number of levels. A mixed level covering array $MCA_\lambda = (N; t, k, (v_1, v_2, ..., v_k))$ provides more flexibility, since factors may have different values. For example, the mixed level covering array depicted in Table 2.3.3 can be written as $MCA_1 = (6; 2, 3, (2, 2, 3))$, or in a more concise notation $MCA_1 = (6; 2, 3, 2^2 3))$.

Algorithms and Implementations

The construction of optimal covering arrays for 2-way interactions is a NP-Complete problem [36] [30], therefore no optimal algorithm is known for

a	a	a
b	b	a
a	b	b
b	a	b
a	b	c
b	a	c

Table 2.4: Mixed level covering array $MCA_1 = (6; 2, 3, (2, 2, 3))$. In opposite to covering arrays and orthogonal arrays, mixed level covering arrays allow factors to have different number of levels.

the general case. However, some research on more constraint problems is available, and even exact mathematical solutions have been produced for certain numbers of factors, interactions, and levels. For example, the exact covering array number (e.g. number of required test cases) is known for 2-wise combinations and 2 levels per factor, if N is even. Furthermore, it has been found that the covering array number grows logarithmically in the number of factors, and a probabilistic bound has been established for 2-way interactions as [18] and [36] observe:

$$N = \frac{v}{2} \log k (1 + o(1)) \tag{2.1}$$

This results however applies only to fixed-level covering arrays, which have limited applicability in practice [48].

In later research, an extended version of covering arrays called *detecting arrays* was introduced by [19], which put further constraints on the covering arrays. By removing irrelevant configurations in the test set, it is possible using detecting arrays to clearly identify the location of a fault [48]. This is done by ensuring that each t-way interaction existing in a system is covered by an *unique* set of tests. More precisely, on a covering array A and any t-way interaction of any of its factors, it defines the function $\rho(A, T)$ to be the set of rows in A, in which this particular interaction T is contained. In order for a covering array to be a detecting array, the following property, defined by [19], must hold true:

$$\rho(A, T_1) = \rho(A, T_2) \Leftrightarrow T_1 = T_2 \tag{2.2}$$

Conversely, the set of rows covering any particular t-way interaction is unique. This means, that if one interaction results in an failure when executing the test suite, an unique set of individual tests will fail, which can be used to identify the failed interaction. While theoretical foundations and even algorithms for creation of covering arrays have been developed [19] [48], the main focus in the research community have been covering arrays and, to a lesser extend, mixed-level covering arrays. In the following, the concepts behind popular algorithms and tools for combinatorial test case generation are summarized.

- AETG: Published in 1994 by [17], the *Automatic Efficient Test Generator* (AETG) is based on previous work on using orthogonal arrays for test suite construction. However, they observed that strictly using orthogonal arrays proved to be difficult, as it cannot be guaranteed that for any combination of factors and levels an orthogonal array exists. Furthermore, it is observed that the construction of such arrays is a NP-Complete problem as shown in [30]. Therefore, different mechanisms and heuristics had to be developed. As [31] states, AETG uses a greedy algorithm choosing from a random pool of test cases to construct the test suite, resulting in non-deterministic test suites [18]. One by one, individual tests are added, until all required interactions between the factors are met. Each new test tries to cover as many uncovered interactions as possible. At the time of publication, the authors were able to show that by using AETG in real-world scenarios, the number of test cases was reduced by up to 92%, compared to then-popular ad-hoc methods for test suite generation.
- IPO: Three years later, in 1997 the *In-Parameter-Order* (IPO) strategy for 2-way test suite generation was published in [30]. Instead of building the complete test set for all parameters simultaneously, as done in AETG, the IPO method works iteratively. It constructs a 2-way test set for the first two parameters, then continues to grow the test set to be valid for the next parameter, and repeats this process until all parameters are covered. Algorithms for extending the test suite horizontally (e.g. when adding a new parameter), and vertically (e.g. when adding a new test to the suite), are presented. It was shown that the IPO algorithm produces results similar to the AETG strategy, however it is not clear whether it is able to do so more efficiently [31]

[18]. While also being a greedy algorithm, in opposite to AETG, IPO is a deterministic algorithm.

- IPOG: The *In-Parameter-Order-General* (IPOG) algorithm was published by the National Institute of Standards and Technology (NIST) in 2007, and builds on the previously published IPO algorithm [31]. It generalizes the 2-way generation technique shown in [30] to a general t-way generation strategy. As with IPO, it starts with constructing a t-way test suite for the first t parameters. It then iteratively builds a $(t+1)$-way test suite by adding the next parameter, until all parameters are covered. Over the years, a number of variants of this algorithm have been published. *IPOG-D* reduces the number of combinations that need to be enumerated using a recursive generation approach, and thus being more efficient in terms of space and time [32]. *IPOG-C* adds support for constraints between parameters [54]. Many of the algorithms are implemented and available via a Java tool *ACTS*, which is developed by the NIST [53].
- Hill Climbing: As described in [18], with the hill climbing approach the task of constructing a covering array is translated into an optimization problem. Each test suite has a cost associated that corresponds to the number of uncovered interactions. Starting from a random test suite, the current solution is transformed. If the cost associated with the new solution is lower, it is accepted as the current solution. When the cost reaches 0, a correct covering array has been found, though it may not be the global optimum.
- Simulated Annealing: Proposed in [46], it is a variant of hill climbing that attempts to reduce the risk of reaching a local, rather than a global optimum. It is doing so by probabilistically accepting a higher-cost, transformed solution as new current solution. This method has been found to be working relatively well, resulting in a number of new upper bounds for certain configurations when constructing covering arrays [49].

Isolating Failure-Inducing Input

After identifying a test configuration that results in an failure, it can be difficult to identify the root cause of the failure, if not working with a detecting array. As many popular tools and algorithms only support the more general covering arrays, this situation can be faced easily. Therefore, in an attempt to reduce the effort needed for debugging the program, several techniques have been developed, some of which are briefly summarized in this section.

Most notably is a technique called *delta debugging*, with well-known algorithms based on this idea called *ddmin* and its extension *dd* [56]. Published in 2002, the algorithms are based on work previously conducted on isolating erroneous code changes in compilers by [55], but extend this work to be usable for program input rather than program code. With delta debugging, the difference (delta) in program behaviour between two similar configurations is observed. If the program worked correctly with the previous configuration, and fails after changing the configuration as little as possible, one has identified the change that was applied as the failure-inducing input. While earlier work conducted by [44] uses a modified form of binary search to identify errors that are caused by one individual input, in the context of multiple interacting inputs a more sophisticated approach had to be developed [55]. Therefore, three problems are identified that are addressed by the *dd* algorithm. Firstly, in many cases not only one individual input causes the failure, but a combination of different inputs is responsible for the fault, which is called *Interference*. Secondly, under the name *Inconsistency*, it is observed that some input configurations are invalid, thus these configurations need to be filtered out. Lastly, it is stated that it is necessary to break down changes into smaller chunks, in order to reach an adequate *Granularity* of changes to observe the failure. Built with these concepts in mind, the *dd* algorithm recursively identifies the error, making use of the *divide-and-conquer*-technique. Even early experiments in the original paper showed, that this algorithm can be successfully applied to real life programs, e.g. the debugger tool *GDB*.

One algorithm for delta debugging published in 2006 that has gained attention is the *Hierarchical Delta Debugging* (HDD) algorithm [42]. Since the original *dd* algorithm ignored structure in input data, the test case reduction was believed to be non-optimal. Therefore, as the authors claim, if the input data is defined by a context-free grammar, e.g. a programming language, it may be beneficial to use the HDD algorithm for isolating the failure [42]. The algorithm tries to build an abstract syntax tree (AST) to gain knowledge about the input. From there on, the well-known *dd*-algorithm is applied to the different levels of the AST, in order to quickly narrow down the root cause of the failure. However, as [43] observed in 2016, the *HDD* algorithm has not been adopted to a great extend in the community. Based on the original algorithm, a new tool called *Picirency* was developed. The tool tries to improve practical aspects of the original algorithm, e.g. increasing the number input types available for reduction, or improving the reduction performance by 25-40% [43].

2.3.4 Random Testing

In opposite to other testing techniques, random testing does not consider any structure of the input or the program itself. While this type of testing was previously seen as inferior [20] [27], many researchers have since put up with this topic and found it to be applicable and effective in a variety of application fields [6]. If the input parameters for random testing are chosen carefully, it can provide valuable insights on the likeliness of a program succeeding. This property is unique to random testing, as other testing methods can only provide information about the presence, but not about the absence of failures [27]. For achieving reliable results, it is important to know how a program is executed in a real-world scenario. As [27] suggests, the viable input space needs to be partitioned, and testers need to estimate the probability of each partition being encountered in a real-life scenario. This so-called *operational profile* should be used to generate the input data. However, if no such information is available, it is still an option use to the uniform distribution. By using random testing it is then possible to derive statistical metrics about the reliability of the program that is being tested, e.g. the *Mean-Time-To-Failure*.

In recent years, with the concept of *Adaptive Random Testing* (ART) a different variant of random testing was introduced [13]. The paper found that the performance of testing could be improved, if the test generation takes so-called *failure patterns* into account. As an example, the inputs that cause a program to fail could be concentrated in a particular region, e.g. the lower third of possible inputs. The authors of the initial paper conducted an empirical study with small programs, consisting of 200 lines of code or less. They found that by incorporating the structural information about the failure patterns, they were able to improve the performance by up to 47%, compared to purely random testing with uniform distribution. However, more recent research has found ART to be inefficient for real-world programs, as the initial results assume unrealistically high failure rates [7]. An empirical analysis about the use of ART finds, that the results could not be reproduced in a real-world scenario [7].

2.4 Oracle Problem

While there are several effective techniques to generate test cases for software programs, all of them are faced with another issue, called the *Oracle Problem*. It describes the fact that it may be difficult to determine the correct output for a given input, especially in an automated manner. As specifications are often informal, the task of providing the output information remains in many cases with the testers [8]. In this section, some of the main techniques found by a comprehensive and relatively recent survey in [8] for reducing the burden on the testers by automating test oracles are summarized.

2.4.1 Model-based Testing Oracle

The first way of automatically obtaining test reference data is to use formal specifications. An obvious choice for doing so is to use formal models, e.g. a state transition system or model specifications in formal languages like the *Uniform Modeling Language* (UML). From this abstract, but formal specification of the systems, it is possible to generate test cases on the one hand (see Section 2.3.1), and the desired output of a test on the other hand. Using a model specified by a state transition diagram, a test case consists of a sequence of inputs that are applied to the system. The test case applies the inputs to the system and observes the corresponding outputs in the path until the test case ends. Therefore, after a concretion and execution of the test cases, by observing the output and states of the actual system under test, its correctness can be verified.

2.4.2 Code Contracts

Another variant of formal specifications are so-called *contracts*. Using contracts it is possible to specify the context of methods and functions in code. Contracts usually provide three different instructions [41]. Using a *pre-condition*, valid inputs to functions can be specified. An example of a pre-condition is the requirement of many algorithms, that an input array needs to be sorted. Secondly, using a *post-condition*, it can be specified in which state the data or object should be after the execution of the function. A sort algorithm could specify as post-condition, that the input array is in fact sorted at the end of the function. Thirdly, a programmer can use a *invariant* to specify properties of an object, that should hold true before the start of any function and at the end of any function. This means that while during the execution of a function the invariant may be violated, as long as it is again met after the execution has finished. Using these formal specifications it is possible to automatically generate tests including the expected results. However, there are some shortcomings of this method.

One problem of practical nature is that most languages, with the exception of Eiffel with the concept of *design-by-contract* [41], do not provide built-in support for contracts. Therefore, one must rely on third-party libraries and frameworks. While there are in fact libraries available for all of the five most popular programming languages [2], only C# with the Pex Framework [52] is also able to automatically generate test cases for the given contracts. Even with this framework, the tests that are generated can only provide probabilistic guarantees, since it is in general unfeasible to exhaustively cover all possible inputs. Therefore, while contracts are a good method of circumventing the oracle problem, in practice they are only of limited usefulness in terms of automated test case generation.

2.4.3 Metamorphic Testing

The concept of *metamorphic testing* (MT) is based on a different approach, and is associated with the class of *derived test oracles* [8]. As [14] describes,

[2]`https://www.tiobe.com/tiobe-index/`, accessed 18.02.2021

unlike most other testing techniques, it can be used both for test case generation and as a test oracle. MT uses existing test cases provided by different testing techniques, to generate so-called *follow-up* tests [57] using *metamorphic relations* (MR). Instead of specifying a specific output for a specific input to test the program behaviour, more abstract properties of the programs are utilized for generating the tests. For example, a function `min(a, b)` that returns the lower of the two parameters, could specify the metamorphic relation `min(a,b) = min(b,a)`. This property can now be checked for a multitude of automatically generated concrete inputs, and for all of the inputs the relation must hold true. A popular implementation of MT is *quickcheck* [15], a tool for testing Haskell programs using so-called *properties*, that are equivalent to metamorphic relations [39]. It provides a simple and lightweight framework for testing relations, since it generates a number of random test cases and the corresponding expected output values automatically.

2.4.4 Regression Testing

Finally, also regression testing can be used as a test oracle. As described earlier in Section 2.2.1, with regression testing a previous version of the SUT is used as a reference. Therefore, it is possible to use any test case generation technique, since all reference output can be obtained by simply running the test suite against the specified version of the SUT. However, this implies that the previous version of the software is in fact correct and does not contain any faults, which can pose problems in practice. Furthermore, since the old test oracle cannot provide reference output for newly added functionality, other testing strategies must be used for testing additional behaviour.

2.5 Digital Audio Workstations and Audio Plugins

As this thesis touches the testability of audio plugins, in this section an overview of the relevant technologies and tools is given. In the modern music industry so-called *digital audio workstations* (DAWs) play a big role [10]. DAWs are programs that allow different tracks to be created, mixed, modified, and exported. There are many different professional vendors of DAWs, prominent examples include Reaper [3], Pro Tools [4], FL Studio [5], or Logic Pro [6]. While the mixing of the tracks and the final exporting into a finished song is usually performed by built-in functionality of the DAW itself, many creative changes are performed with plugins for the DAWs. Some of the plugins even provide virtual instruments which can be controlled by the MIDI interface.

Audio plugins are essentially shared libraries that are developed against a number of different SDKs, which are then loaded at runtime by the DAW. The most prominent plugin formats are VST2 and VST3 published by Steinberg [7], AAX published by Avid [8] and Audio Unit published by Apple [9]. In an usual workflow, the DAW is the controlling component in the setup. When the audio processing starts, the DAW will utilize the plugins by calling dedicated functions provided by the SDKs. Most importantly, the DAW passes the raw audio data to the plugin, so that the plugin is able to add the desired effect by manipulating the contents of the audio buffer. As audio processing is a real-time domain, plugin developers must be aware of the fact that they need to complete their processing in time, otherwise so-called *glitches* occur, which are clearly audible disturbances in the audio. The plugins itself are either manipulated via a GUI, or by a common parameter interface also provided by the SDKs, which allows users to parameterize the plugins in the DAW directly. With this interface,

[3]`https://reaper.fm`, accessed 19.2.2021
[4]`https://www.avid.com/pro-tools`, accessed 19.2.2021
[5]`https://www.image-line.com/`, accessed 19.2.2021
[6]`https://www.apple.com/logic-pro/`, accessed 19.2.2021
[7]`https://www.steinberg.net/en/company/technologies/vst3.html`, accessed 19.2.2021
[8]`http://apps.avid.com/aax-portal/`, accessed 19.2.2021
[9]`https://developer.apple.com/documentation/audiounit`, accessed 19.2.2021

all exposed parameters are accessed by index and provide a value interval of $[0.0, 1.0]$, inclusive.

Most DAWs use multiple threads to communicate with plugins, usually with at least one dedicated thread for manipulating parameter values and one dedicated thread for audio processing itself. This means that audio plugins are real-time multi-threaded software, which requires extensive care by software developers.

3 Design and Implementation

In this section, the architecture and design choices of the regression testing tool are described. The source code of the tool is available on Github [1] Furthermore, some relevant implementation details and limitations are addressed.

3.1 Description of the Regression Testing Tool

The tool is written in C++, a widely used language for audio plugin development. There are multiple reasons for C++ being the standard language in the audio industry. Firstly, because it is a mature language and provides very good performance, it is very well suitable for real-time applications. Furthermore, as described earlier, audio plugins are essentially shared libraries developed against different SDKs that are implemented in C. Therefore, a language that is compilable to this format is necessary. Lastly, there exists a popular open-source cross-platform development framework called *JUCE*[2], which provides a large number of utility functionality specifically targeted at audio development. For example, it provides highly complex adapter functionality that allows the targeting of different plugin standards from the same codebase, without the need for additional configuration. In case of the regression tool, especially the functionality of loading plugins targeting different standards proved to be extremely useful.

[1]`https://github.com/vallant/reta`, accessed 25.3.2021
[2]`https://github.com/juce-framework/JUCE`, accessed 25.02.2021

3.1.1 Workflow

The functionality provided by the tool is separated into five different phases, as can be seen in Figure 3.1. These phases are intended to be run consecutively, with information between the phases being passed via JSON files. This design allows for decoupling as well as extensibility.

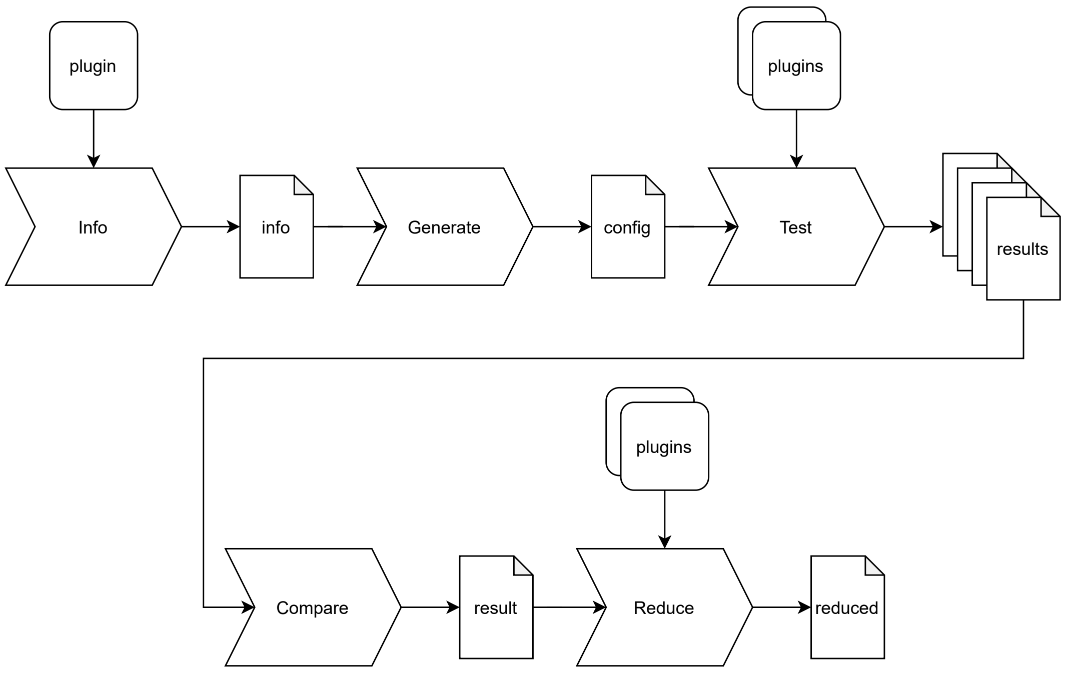

Figure 3.1: Workflow of the regression tool. The five phases communicate with JSON files. After finishing the last phase, a single JSON file containing the parameter-values and the test configuration that resulted in a difference in audio processing.

In the *Info-Phase*, the basic information about the plugin is collected, including its name, or the number and configuration of parameters. Based on this information, the *Generation-Phase* generates a test-suite, which is represented as a JSON file. This generation takes place either randomly or combinatorially, where each individual test includes information about the parameter configuration and the input signal to be used. In the following *Test-Phase*, the previously created suite is executed, resulting in a folder holding the test results. Apart from general information about the test suite and the plugin itself, for each individual test of the test suite the folder holds an audio-file and a companion file holding meta-information. These two files represent the resulting audio content after the test has been executed with the given configuration. The test phase is executed twice using the same configuration on two different versions of the same plugin, resulting in two dedicated folders holding the test results. In the *Comparison-Phase*, the signals and meta-data produced by the plugins are compared, again writing a JSON file holding the relevant pieces of information for further processing. For the comparison of the rendered audio files, the sample-by-sample measure of the *Root-Mean-Squared-Error-Deviation* (RMSE) is calculated and compared against a threshold:

$$RMSE = \sqrt{\frac{\sum_{t=1}^{T}(x_{ref}[t] - x_{actual}[t])^2}{T}} \tag{3.1}$$

If differences are discovered during comparison, it is not immediately clear which, parameters are responsible for the failure. Therefore, the final *Reduction-Phase* implements the *ddmin* algorithm (see Section 2.3.3) to highlight the parameter-values that caused the difference. As discussed previously, this is done by iteratively creating subsets of the failing configuration and running the tests against both versions of the plugin.

As described at earlier points in the thesis, two different test case generation modes are supported, namely random generation and combinatorial generation. To provide the combinatorial generation strategy, the well-known *ACTS* tool developed by the *NIST* is integrated. Because *ACTS* is a Java tool, the use of the *Java Native Interface* (JNI) is required. This means that developers need to use the Java Development Kit (JDK) and users of the software need to have the Java Runtime Environment (JRE) installed. As this requirements may be undesired by users of the tool, in addition to *ACTS*, also a different implementation of the *IPOG* algorithm written in C++ is integrated [3]. The default version of the tool therefore does not have any runtime dependencies.

[3] `https://github.com/jesg/dither-cxx`, accessed 2.3.2021

3.1.2 Client-Server Mode

Since professional-grade audio plugin development (at least) provides software for Windows and MacOS, this tool also provide means for performing the regression testing across operating system bounds. By providing this client-server functionality, it is also possible to use the tool in a continuous-integration environment. This is done via two special commands called *Serve* and *Update*. The *serve*-command starts a simple HTTP-server, which acts as a storage for the reference output of the plugins, alongside with the test suite that was used to generate the output. Therefore, instead of running through the *Info* and *Generate* phases, the server can be queried directly during the *Test*-Phase.

At the time a new reference version is created and accepted by the development team, it is necessary to update the reference regression output on the server, which is done via the *update* command. For doing so, it is necessary to go through the *Info*, *Generate*, and *Test* phase again. By invoking the *update* command, the folder containing the new reference output is uploaded to the server, overriding the old references.

3.2 Limitations

While the tool can help with identifying regression issues between different versions of the plugins, there are a number of situations in which it can not (yet) be used, which are summarized in this section.

First and foremost, because this tool is performing regression testing, previous versions of the software are used as ground-truth. While this reduces the resources required by developers to set up testing, this type of testing can only detect the presence of differences, not the presence of failures. This means that if the tool recognizes differences between two versions of the software, human knowledge is needed to classify this difference. If the reference version contained an error that is corrected in a new version, the tool may detect this, despite of the change being completely fine. On the other hand, if functionality is added in the new version of the software, due to the lack of this functionality in the reference version, testing of the new parts of the software is not possible. In sight of this situation it is clear that this tool cannot provide a complete testing solution, but rather support the process. It helps the developers and testers to focus on the changes that happened during the latest iteration, reducing the risk of breaking existing functionality.

Apart from only being able to highlight differences between versions, the multi-threaded behaviour of the plugins is not assessed. As described in Section 2.5, plugins operate in a multi-threaded and real-time environment, as audio processing and parameter handling are in many cases handled by dedicated threads. It is, for example, possible to change parameter-values of the plugin, while audio processing takes place. Practice has shown that this is a constant source of software failures. However, scheduling of threads is usually functionality provided by the operating system. Therefore, it cannot be guaranteed that a parameter change, that would influence the audio processing, arrives at exactly the same time during multiple runs of the same test suite.

The lack of support for testing the multi-threaded behaviour also means, that this tool cannot be used to assess the compatibility of different Digital Audio Workstations (DAW). Since the plugin standards only put limited restrictions on the order in which the exposed functions should be called, it has been found that DAWs utilize the plugins in very different ways. This behaviour cannot be simulated with this particular tool, however other tools like *pluginval* [4] are available, that claim to help increasing the robustness and compatibility of plugins.

Finally, the tool does not yet provide support for MIDI messages, therefore only effect plugins are supported at the moment.

[4] `https://github.com/Tracktion/pluginval`, accessed 25.02.2021

4 Evaluation

In this section, the evaluation setup and the subsequent results are presented, while a discussion of the results can be found in Section 5. The primary goal of the evaluation is to assess the effectiveness of random and combinatorial test case generation strategies, in conjunction with regression testing as test oracle strategy. This is done by introducing mutations into the audio-processing code of real-world audio plugins, in order to let them be tested by the regression tool. Furthermore, an analysis of the mutated files is conducted to put the results into perspective.

4.1 Evaluation Setup

As described in the previous paragraph, the evaluation setup consists of mutating plugins, in order to test them using the regression tool with various settings. For this purpose, three different plugins are evaluated, two of which are considered professional-grade. Two plugins are so-called equalizers, one being a compressor plugin. The professional-grade plugins are provided by sonible GmbH [1].

4.1.1 Mutation Generation

The generation of the mutations is performed using *universalmutator* [23], a language-agnostic mutation tool written in Python. In opposite to other tools available for C++ that operate on byte-code level, *universalmutator* performs the mutation on a source-code level. This property may be considered

[1]`https://sonible.com`, accessed 1.4.2021

hindering in other contexts, since it means that after having mutated the source code the project needs to be compiled, instead of only mutating already translated binary code. However, in this context this is actually a desired property, because the same set of mutated plugins should be tested against changing test suites. Therefore, compiling the mutants only once to re-use the resulting binaries results in a better overall performance. The mutation tool uses a regex-based strategy, with introduced mutations including, among others, off-by-one errors, argument swaps, literal replacements, and condition inversion. This make this tool suitable for the task, since mutations can be introduced into the control flow, as well as mathematical expressions. The mutations are introduced into sections of the source code that participate in modifying the final audio output of the plugin, therefore a wide range of different scenarios can be covered.

4.1.2 Mutation Selection

Because the design of the mutation tool is language-agnostic and purely based on text replacement, it does not have any understanding of the syntax or semantics of the source files it processes. While this allows the tool to be usable in a variety of situations and programming languages, it requires special care when generating the mutations. A moderation and selection process is necessary for the mutants to be usable, because the tool only performs text replacements that, in many cases, results in syntactically incorrect source files. Commonly used language features in C++, e.g. *templates*, have proven to be a frequent source of errors, since the tool would mistake the enclosing less-than and greater-than signs as condition in a branching statement as can be seen in Listing 4.1.

```
template <class T>
class Example(T t) {}
-----
template <=class T>
class Example(T t) {}
```

Listing 4.1: Example of an invalid mutation. The declaration of a template is interpreted as branching condition.

Another issue that is rooted in the design and scope of the tool also called for a manual selection process. Because the tool should only focus on the detection of regression errors in audio processing, by design the test suite is incomplete. Therefore, even when a mutant is syntactically correct, it may not be discoverable by the regression tool, if the mutation is introduced into code that is not related to audio processing. Finally, because the mutation tool generates a large number of mutations even for small source files, due to practical reasons only a subset of the generated mutants is used for evaluating the tool and the selected strategies.

To address these issues, a semi-manual mutant selection process is used, as depicted in Figure 4.1.2.

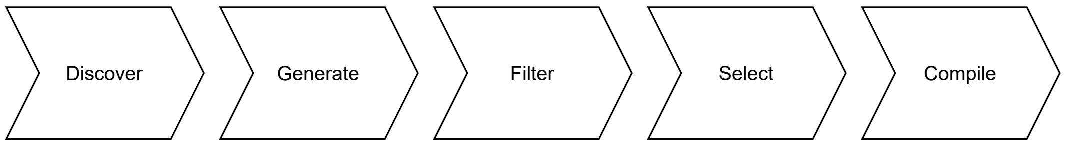

Figure 4.1: Mutation selection process. Because the test suite by design is incomplete, a semi-manual approach for filtering and selecting mutations was used, such that all mutations are potentially detectable by the tool.

The first step is to discover the source files contributing to the audio processing of the plugin, which is mainly done by obtaining coverage information using the *LLVM* compiler suite and its related tools. After having identified the relevant source files, the mutation tool *universalmutator* is used to generate mutated source files, regardless whether they are syntactically correct or the mutation is introduced in DSP code. Using the coverage information collected before, a first filtering of the mutated source files is then conducted. Sections of the code that do not contribute to the audio transformation can be identified, and source files that contain mutations in these regions are discarded. The same is true for mutations that obviously cannot result in any observable difference, e.g. string replacements in comments. After performing this first filtering, a fine-grade selection process is conducted. In order to minimize the bias that can be introduced by manually selecting mutations, a script is used to automatically and randomly propose mutated source files. These proposals are then reviewed manually to ensure that they are syntactically correct, and potentially detectable by the regression

Plugin	Number of files	Number of mutations
Frequalizer	4	748
smart:comp	13	611
smart:EQ2	9	865

Table 4.1: Number of mutated files and total number of mutations per plugin.

.

tool. For this purpose the notion of a *relevant mutation* is introduced, which states that a mutation is injected into a function or method that is used in the audio processing of the plugin. When the desired number of mutations per source file and per plugin is reached, as a final step the mutated source files are compiled into the binaries used for evaluation.

By performing the process described in this section, the following numbers of mutated plugins are obtained, as can also be seen in Table 4.1. For the *Frequalizer* 748 mutations in 4 files are constructed, for the *smart:comp* 611 mutations in 13 files are generated and the experiments for the *smart:EQ2* are conducted with 865 mutations in 9 files.

4.2 Test Scenarios

In the following the test scenarios are described and the results of the experiments are presented.

4.2.1 Experiments with Random Test Generation

This section describes the experiments that are conducted using a random test generation strategy. A total of five different scenarios are analyzed, which differ in the choice of the input domain for random selection. Furthermore, the experiments are conducted with different numbers of tests in the test suite. All experiments are repeated three times with different seeds provided to the random generator, in order reduce the effect of outliers. This low number of repetitions is due fact that the test runs take a long time to complete, in some instances more than 24 hours. Therefore, due to computational constraints it is unfeasible to include more data points.

Random Values for All Parameters

In the first scenario, random values from the continuous inclusive interval $[-0.1, 1.0]$ are taken per parameter per test. Information about the discreteness of parameters is not used. A negative value means that the parameter is not modified and the defaults as provided by the plugin itself are used in the test. This means that for each parameter there is a 10% chance of using a default value. The results of this experiment can be seen in Figure 4.2.

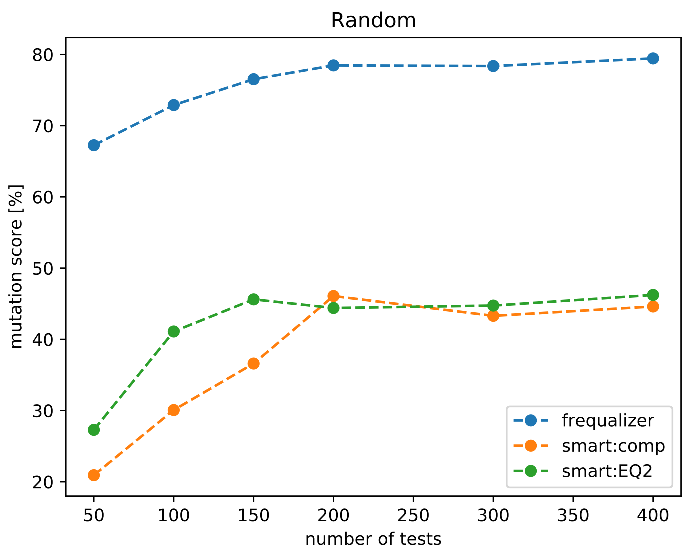

Figure 4.2: Mutation score for random test values. The values are randomly chosen from the interval $[-0.1, 1.0]$.

Random Values with 10% Chance to Choose a Boundary Value

In some cases it may be the case that errors do not occur within the value interval, but rather at the boundary values. Because it is very unlikely that using a random selection one of the boundary values is actually chosen, this experiment uses a different value selection process. While in most cases still a random value is chosen from the interval $[-0.1, 1.0]$, with a probability of 10%, the tool chooses one of the three boundary values $\{0.0, 0.5, 1.0\}$. Again, negative values represent the plugins default value for the particular parameter. The results of this experiment can be seen in Figure 4.3.

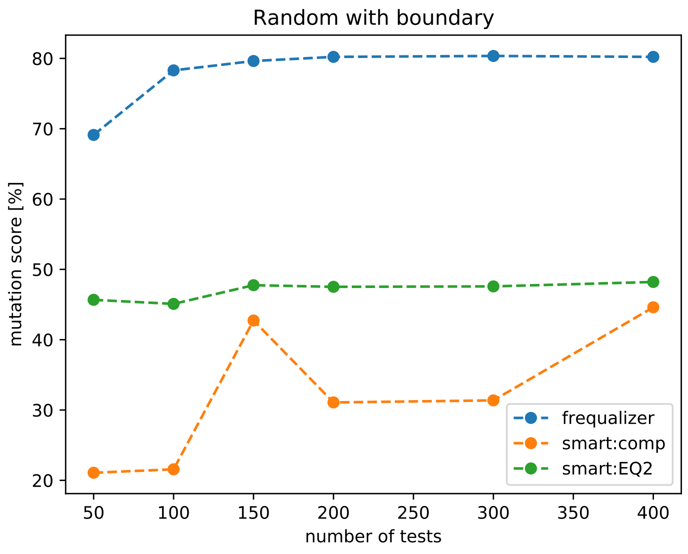

Figure 4.3: Mutation score for random test values. The values are randomly chosen from the interval $[-0.1, 1.0]$. With a chance of 10% one of the boundary values $\{0.0, 0.5, 1.0\}$ was chosen.

Random Selection from Values of Interval Sampled in 0.25-Steps

In this experiment, the possible input values are obtained by sampling the interval $[-0.25, 1.0]$ in 0.25-steps. Information about discreteness of parameters is not considered, in each test case a parameters value is chosen randomly from the sampled values, where negative values represent the parameter default values.

Table 4.2 shows the respective input models for the three plugins. Figure 4.4 displays the result of this experiment.

Plugin	Input Model
Frequalizer	6^{32}
smart:EQ2	6^{68}
smart:comp	6^{27}

Table 4.2: Input models for random selection from sparsely sampled interval.

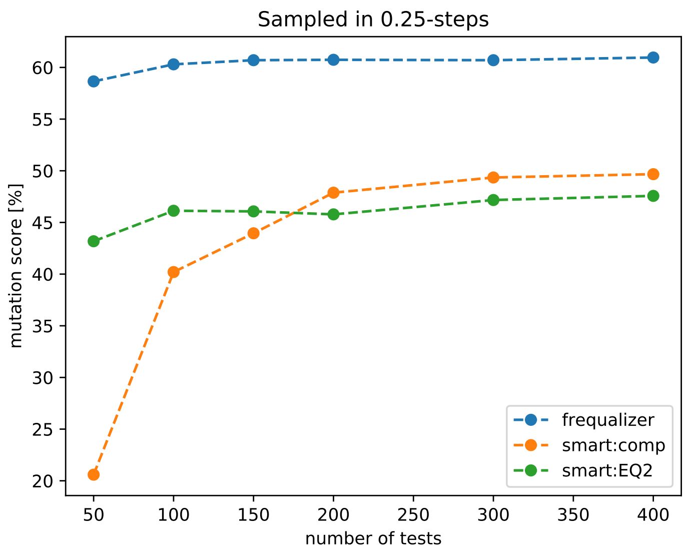

Figure 4.4: Mutation score for randomly chosen value from interval $[-0.25, 1.0]$ sampled in 0.25-steps.

Plugin	Input Model
Frequalizer	12^{32}
smart:EQ2	12^{68}
smart:comp	12^{27}

Table 4.3: Input models for random selection from finely sampled interval.

Random Selection from Values of Interval Sampled in 0.1-Steps

As in the previous experiment in Section 4.2.1, the input domain is sampled. However, in this experiment the sampling takes place in 0.1-steps, resulting in more possible values that can be chosen from, where negative values again represent the parameter default values. Results of this experiment can be seen in Figure 4.5, Table 4.3 shows the input models that were used in the experiment.

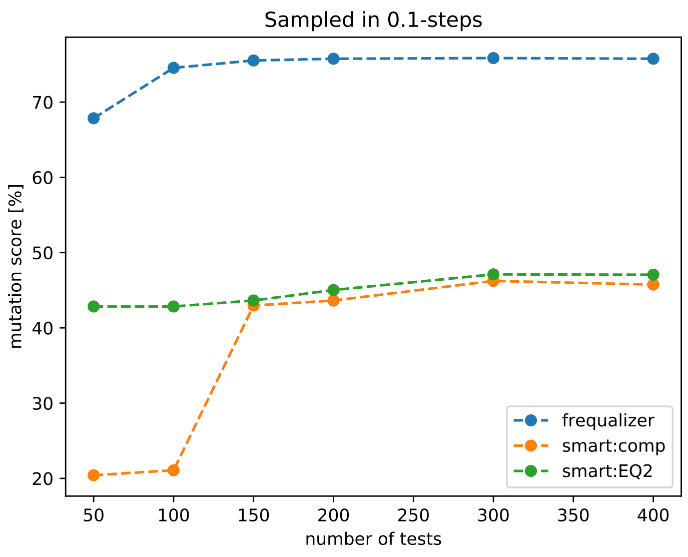

Figure 4.5: Mutation score for randomly chosen value from interval $[-0.1, 1.0]$ sampled in 0.1-steps.

Plugin	Input Model
Frequalizer	$3^7 12^6$
smart:EQ2	$3^{34} 7^8$
smart:comp	3^9

Table 4.4: Input model for discrete parameters. The value for continuous parameters is still chosen randomly from the interval $[-0.1, 1.0]$ with a 10% chance of using a boundary value.

Utilization of Information About Parameter Discreteness

In some cases audio plugins provide more information about the parameters. Therefore, it is be possible to identify parameters that may only take discrete values. An example for this scenario is a boolean parameter, that can only hold values 0.0 or 1.0. This experiment takes advantage of this information, resulting in more complex input models as shown in Table 4.4. This table only applies to parameters that are announced to be discrete by the plugin, while for continuous parameters a different approach is taken. As already described in Section 4.2.1, the value for continuous parameters is randomly chosen from the interval $[-0.1, 1.0]$ in 90% of cases, while in the remaining 10% of cases one of the three boundary values $\{0.0, 0.5, 1.0\}$ is chosen. Results of this setup are displayed in Figure 4.6.

Comparison of Random Generation Strategies

This section provides diagrams on how the different experiments performed in comparison, on a per-plugin basis.

4.2.2 Experiments with Combinatorial Test Generation

While the previous section described the results that are obtained by generating the test cases randomly, this section is presents the results of generating the test cases in a combinatorial manner. Because the test generation using *IPOG* is deterministic, only one round of each experiment is conducted. There are four distinct scenarios in terms of the input domain that are

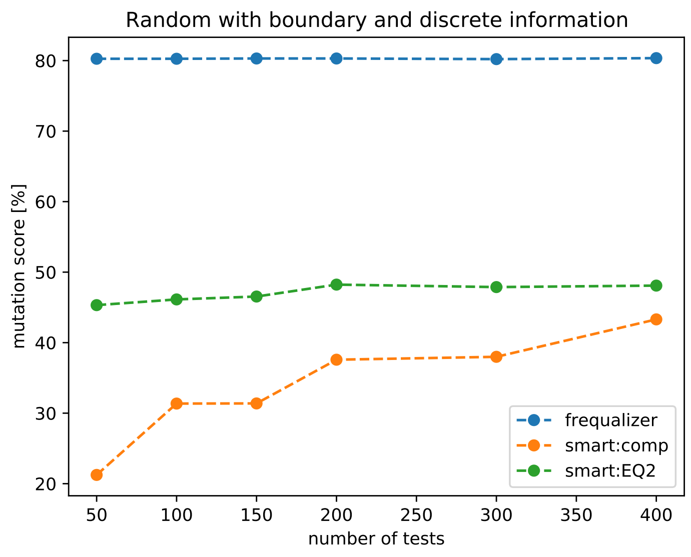

Figure 4.6: Mutation score for test suites utilizing information of discrete parameters. Continuous parameters values are chosen from interval $[0.0, 1.0]$, with a chance of using a boundary value from $\{0.0, 0.5, 1.0\}$. Discrete parameters values are chosen randomly from the discrete values they can take.

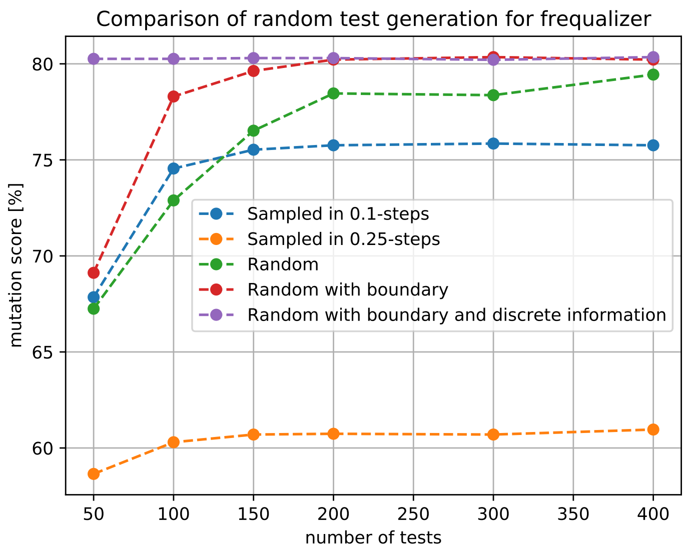

Figure 4.7: Comparison of random test case selection for Frequalizer.

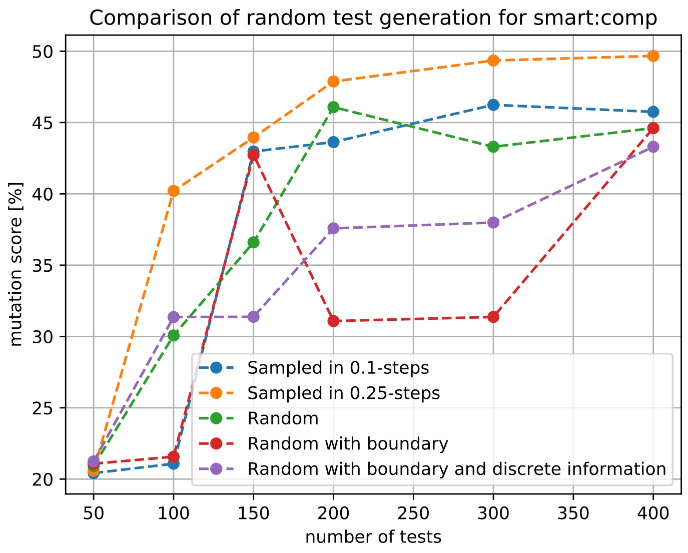

Figure 4.8: Comparison of random test case selection for smart:comp.

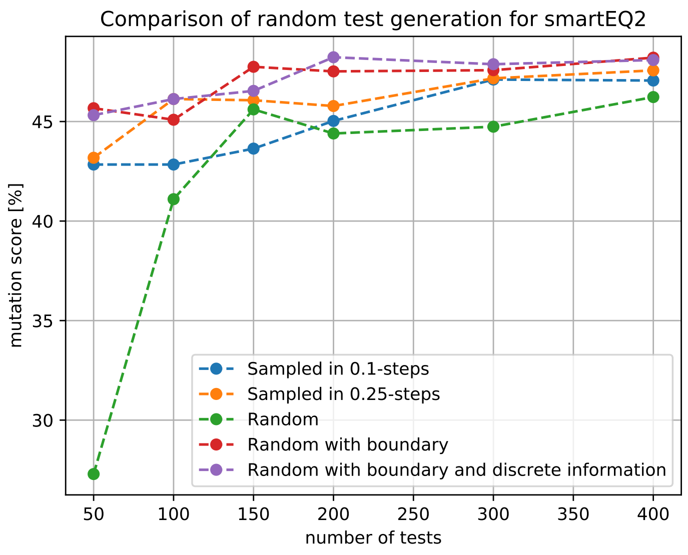

Figure 4.9: Comparison of random test case selection for smart:EQ2.

conducted. The input domains of the first and second experiment consist of values obtained by sampling the continuous interval $[-0.1, 1.0]$ in 0.1-steps, and $[-0.25, 1.0]$ in 0.25-steps. This is consistent with the input domain of the experiments conducted with the random generation strategies in Section 4.2.1.

The third and fourth experiment take a more advanced approach and also include the information provided by the plugin about the parameters. Specifically, the plugin may announce that certain parameters are discrete and therefore can only take certain values. For example, a boolean parameter would announce that it can only hold values $\{0.0, 1.0\}$, therefore instead of sampling the complete interval $[0.0, 1.0]$, a more targeted input domain can be obtained.

The experiments are only conducted with a combinatorial strength of 2 and 3. As the following tests show, no further improvement is to be expected by higher strengths, while the number of test cases increases considerably. For the *smart:EQ2* it is not possible to collect values for all scenarios, due to computational constraints.

Plugin	Input Model - Sparse	Input Model - Fine
Frequalizer	$3^{7}12^{6}6^{17}$	$3^{7}12^{6}12^{17}$
smart:EQ2	$3^{34}7^{8}6^{26}$	$3^{34}7^{8}12^{26}$
smart:comp	$3^{9}6^{18}$	$3^{9}12^{18}$

Table 4.5: Input models for combinatorial testing.

The data points labeled with *Fine - All* represent the input models as shown in Table 4.3, *Sparse - All* is performed with the input model shown in Table 4.2. In case of the data points labeled with *Fine - Discrete* and *Sparse - Discrete* a different approach is taken. Here the information provided by the plugin regarding the discreteness of parameters is in fact used for generating the tests. This means, that for discrete parameters the input model is the same as shown in Table 4.4. While in the previous section the value for continuous parameters is chosen randomly, for these tests the input interval is sampled. Therefore, for continuous parameters the interval is sampled in 0.1-steps in case of the *Fine - Discrete* data points, and sampled in 0.25-steps for the *Sparse - Discrete* data points. This results in input models as shown in Table 4.2.2.

Experiments with Combinatorial Strength 2

In the following section, the mutation scores of test suites generated with a combinatorial strength of 2 for the described input domains are shown. In Figure 4.11, the mutation score for the plugin *frequalizer* is presented, Figure 4.11 depicts the mutation score for the plugin *smart:comp*.

Experiments with Combinatorial Strength 3

This section presents the mutation score of test suites generated with a combinatorial strength of 3. The results of the experiments for *frequalizer* can be seen in Figure 4.13, while Figure 4.14 shows the results for the *smart:comp*. Note that due to computational constraints it is not possible to conduct all experiments when the input space is sampled in 0.1-steps.

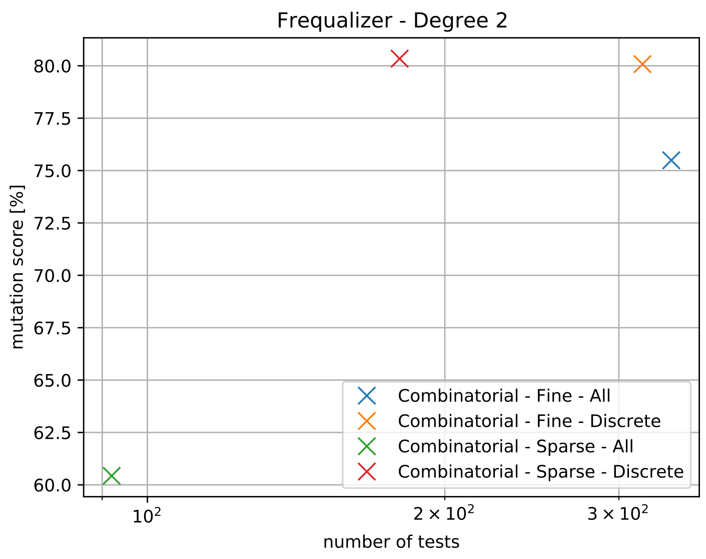

Figure 4.10: Combinatorial tests with combinatorial strength of 2 for Frequalizer.

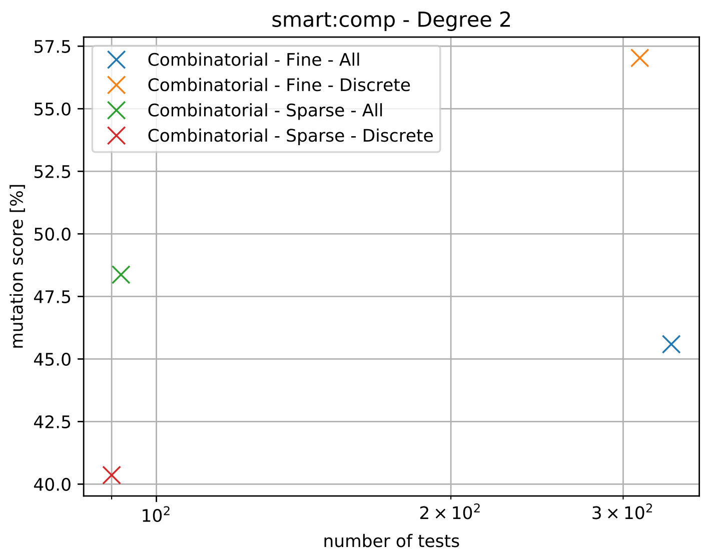

Figure 4.11: Combinatorial tests with combinatorial strength of 2 for smart:comp.

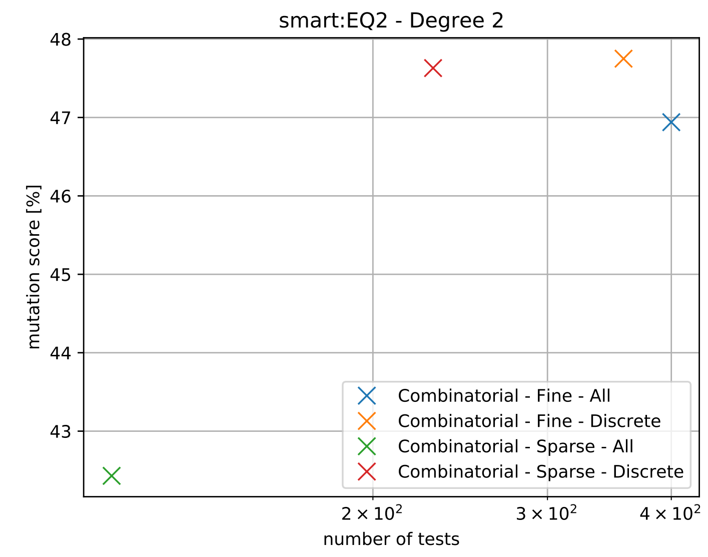

Figure 4.12: Combinatorial tests with combinatorial strength of 2 for smart:EQ2.

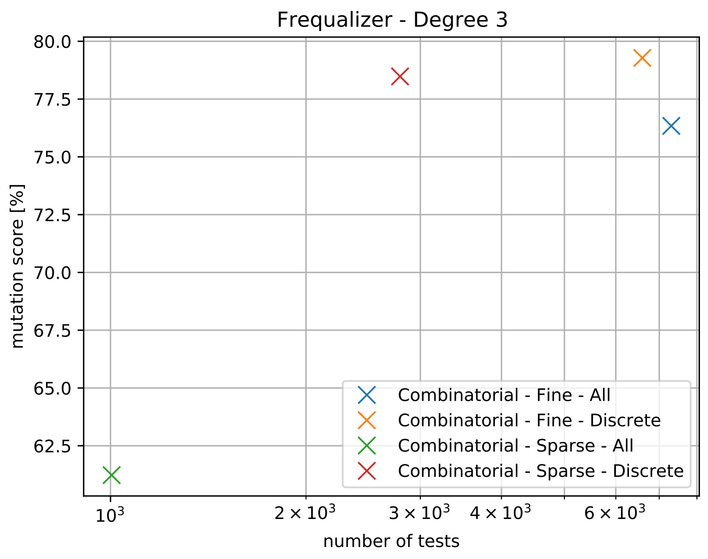

Figure 4.13: Combinatorial tests with combinatorial strength of 3 for Frequalizer.

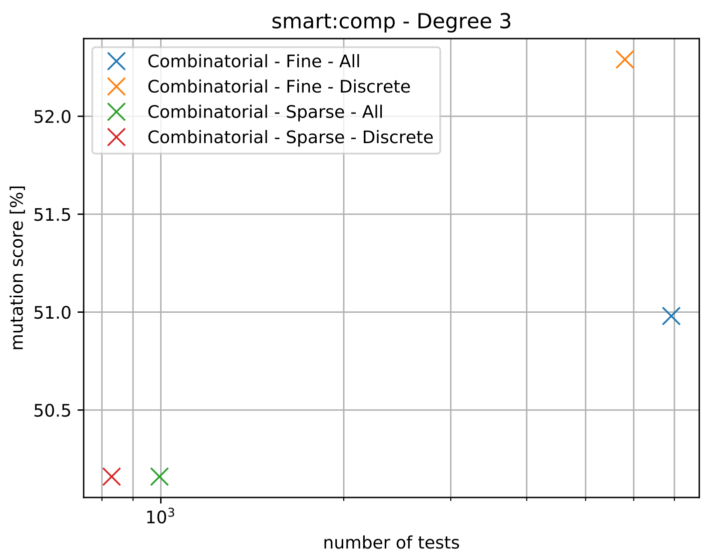

Figure 4.14: Combinatorial tests with combinatorial strength of 3 for smart:comp.

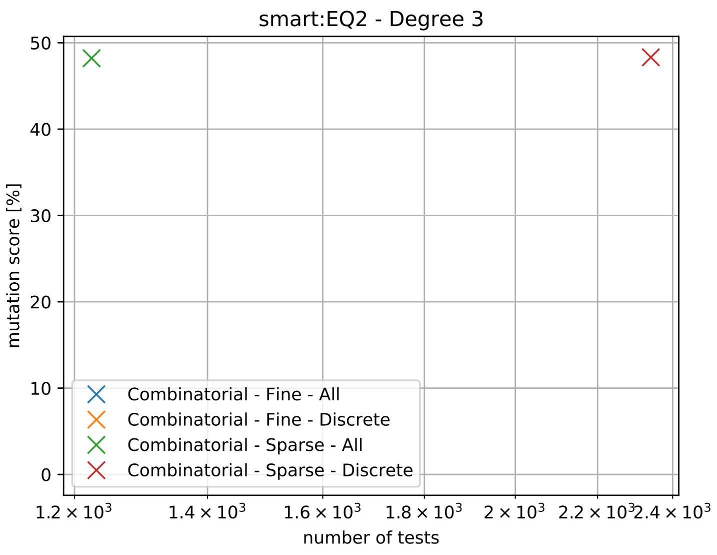

Figure 4.15: Combinatorial tests with combinatorial strength of 3 for Frequalizer. Due to computational constraints it was not possible to conduct the experiments when the input space was sampled in 0.1-steps.

4.3 Complexity Analysis of Signal Processing Code

To put the data presented in the previous sections into perspective, the source files, that are chosen to be included in the mutation process, are analyzed. The results of this analysis are presented in this section. For performing this task, the tool *metrics++* [2] is used. With its help, the number of significant lines of code, the maximal nesting depth, and maximal cyclomatic complexity of the source code associated with audio processing are collected. The data is averaged on a per-plugin basis and depicted in Figure 4.16.

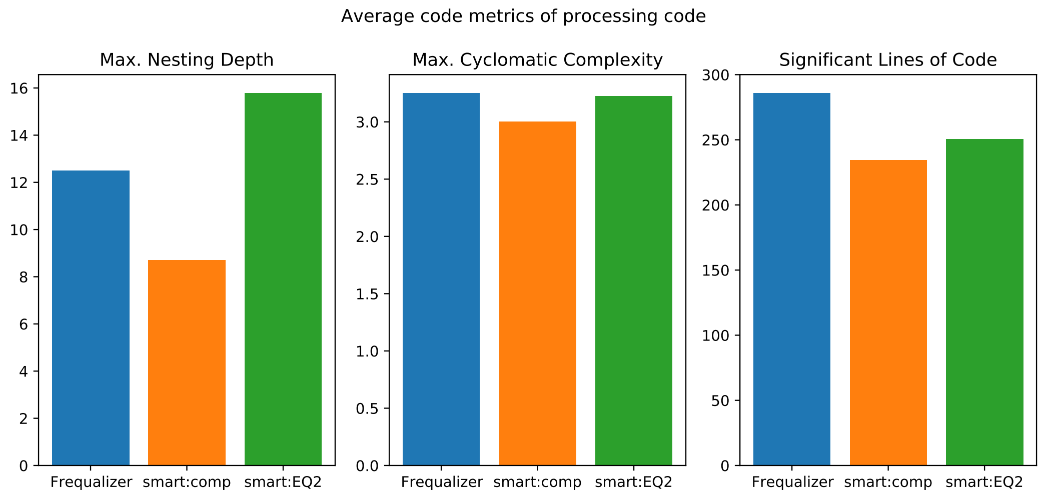

Figure 4.16: Averaged code metrics. The numbers represent the per-plugin average of files used in mutation.

[2]https://metrixplusplus.github.io, accessed 3.3.2021

5 Discussion

In this part of the thesis, the results presented in Section 4 are discussed and put into context. A critical reflection on the reasons for the relatively low mutation score is given. Furthermore, the results of the combinatorial and random test generation strategy are compared in more detail, to reach a conclusion on which one is better suited in which situations. For doing so, the data on the complexity of the signal-processing source code is interpreted.

5.1 Comparison of Test Generation Strategies

The data presented in the previous chapter reveals an interesting fact. In the context of the experiments that are conducted by this thesis, it can be seen that test suites generated randomly reach approximately the same mutation score as test suites that are generated using combinatorial strategies. While this means that both methods are potentially suitable for testing audio plugins, the highest mutation score is reached by a different number of tests in the test suites. The data shows, that considerably less randomly generated tests are necessary to reach the highest mutation seen in the experiments, than by using the combinatorial methods.

The benefits of using random testing instead of combinatorial testing are especially visible when comparing the results of Section 4.2.1 to Section 4.2.2. This is displayed in Figure 5.1. This diagram shows the mutation score for an input domain sampled in 0.1-steps from the interval $[-0.1, 1.0]$. It can be seen, that the mutation score of test suite generated by the random strategy quickly converges. On the other hand, it shows that the mutation score of test suites constructed by the combinatorial method, both for combinatorial strength 2 and 3, is not decisively higher than for the random method, even though much more tests are needed.

When comparing the test suites generated with a combinatorial strength of 2 and 3, no improvement can be observed using the higher number of interactions. However, as expected, a much higher number of tests is needed to satisfy the testing requirements. Therefore, the results indicate that in context of the plugins that were used in the experiments, no significant sections of signal processing code are controlled by complex logical expressions. This is also in line with the fact that randomly generated test suites are able to reach a similar mutation score at even lower numbers of tests.

It has to be mentioned that while the mutation score for the open-source plugin *frequalizer* is quite high at about 80%, the mutation score for the professional-grade plugins only lies in the neighborhood of 50%. It can further be observed that in any case, the mutation score for combinatorial and random test suites is consistent. A spot check of the mutations that have not been killed indicates, that these mutations are located in sections of the code that simply have not been executed by the test suite, or that do not effect the signal processing. A reason for this could be that a number of mutated files is not located in code belonging to the product itself, but rather in library code that provides utility classes and functionality. This utility code may not be used directly in the plugin, and mutations in these sections cannot be found by the tool. In opposite to this situation, while the *frequalizer* plugin also uses functionality provided by an utility library, it utilizes nearly all of the code available for its audio processing in certain configurations, therefore the mutation score is considerably higher.

In general, the evaluation shows that for the three plugins that are used in the experiments the generation of test suites, using a random strategy does not perform inferior compared to combinatorial methods in many cases. However, random testing does so with considerably less tests.

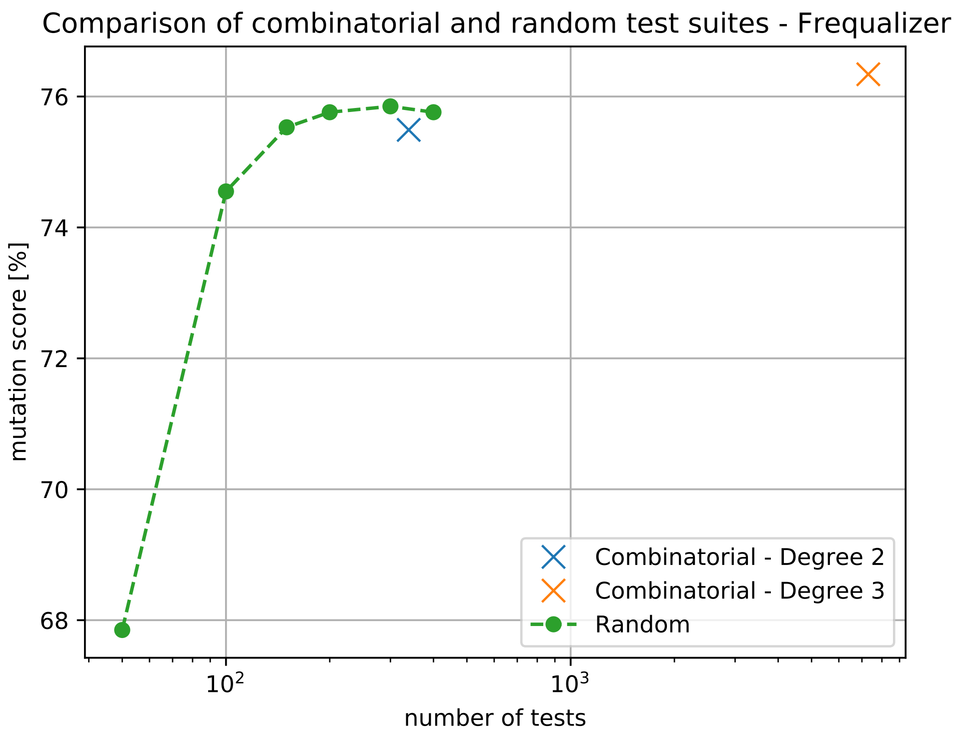

Figure 5.1: Comparison of random and combinatorial generation strategies. Both experiments are conducted for *frequalizer*, and use an input domain that is obtained by sampling the interval $[-0.1, 1.0]$ in 0.1-steps. It can be seen that a similar mutation score can be obtained by considerably less tests when using a random generation strategy. Using a higher combinatorial strength for combinatorial generation does not improve the mutation score decisively.

5.2 Code Complexity of Mutated Source Files

While the last section describes the results of the experiments and compares the two test suite generation techniques, in this section the complexity metrics collected in Section 4.3 are interpreted. The metrics of nesting depth, cyclomatic complexity and significant lines of code quite similar, as Figure 4.16 shows. It is noteworthy, that while the average nesting depth of the classes is relatively high, the cyclomatic complexity within these functions is quite low. This indicates that while the software is in fact complex and deeply nested, its logical complexity is low. Therefore, it can be argued that the probability of a fault being introduced by an interaction of two or more different parameters is low, limiting the benefits of using combinatorial test generation methods. The results provided by the experiments in Section 4 support this assumption, since the rigorous coverage of interaction faults only results in a limited improvement of the mutation score, if any.

However, it is important to state that no relevant empirical data on nesting-depth and cyclomatic complexity in professional software has been found in literature. Therefore, more research is necessary to put these findings into context.

6 Conclusion & Future Work

6.1 Conclusion

In this thesis, research is conducted on the use of automatic regression testing for digital signal processing applications, specifically targeted at plugins for digital audio processing. For doing so, an automatic cross-platform regression testing tool is presented, that supports combinatorial and random test case generation. It applies the same configuration to two different versions of the plugin and renders test output. The result is compared sample-by-sample, in order to discover differences in audio output. The focus of the thesis is to assess, whether combinatorial test case generation can outperform simple random test case generation strategies. This is done by introducing mutations into the sections of source code associated with audio processing in three different real-world audio plugins. Using test suites generated by the two strategies, it is evaluated whether the regression tool is able to detect the mutations. The results of the evaluation show, that the use of combinatorial test case generation strategies does not provide relevant improvements compared to random test generation. Moreover, it is found that a similar mutation score can be achieved by a lower number of random tests.

In order to provide more insights into the reasons for these findings, source code related to signal processing of the plugins used in evaluation is analyzed. It is discovered, that although the nesting depth of the analyzed classes is high, its cyclomatic complexity is low. This means that there are only a limited number of cases in which an error is only observable by using a certain combination of input parameters. Therefore, it can be explained why the rigorous coverage of parameter-combinations, as provided by the combinatorial test generation strategy, does not improve the coverage considerably.

However, as the next section describes, it may be necessary to analyze the source code structure of a greater number of DSP applications, in order to generalize the findings for this application domain.

6.2 Future Work

There are multiple directions in which further research could be conducted.

First and foremost, in order to gather more reliable results it may be beneficial to perform the mutation analysis described in Section 4 with additional audio plugins. For doing so, it would be advisable to include different types of plugins, namely synthesizers and plugins controlled via MIDI-messages, since as described in Section 3.2, the tool does not yet provide support for these kinds of plugins. The use of mutation testing for evaluating the effectiveness of the testing techniques in general should also be compared to real-world data. A possible research direction could be, to empirically observe software projects during a longer period of development, and observe whether the techniques laid out in this work are able to identify regression errors. Doing so would also provide valuable information about the ability of mutation testing to simulate errors in real-world software projects.

Another potential research topic could be a broader empirical study, collecting complexity metrics (e.g. nesting depth) for modern professional-grade software in general and for DSP code specifically. This would make it possible to put the results presented in this thesis into context, and provide a better foundation for the observation that signal-processing code is not prone to be subject to faults caused by an interaction of a multitude of parameters.

A third possible direction is to conduct experiments with different distance metrics used in comparing audio signals. In this thesis, as described in Section 3, the *root-of-mean-squared-error* (RMSE) is used as a metric for deciding on the similarity of two signals. This sample-by-sample comparison method may not be suitable in all situations. For example, if a DSP application produces nondeterministic output for any reason, obviously such a strict distance measure does not produce the desired results. Furthermore, it may be beneficial to employ metrics that provide more information about the structure of the observed differences. One possibility could be to use *cross-correlation*, in order to observe differences that stem from an offset in the audio signals.

The field of audio plugin development may also be a good candidate for implementing combinatorial testing techniques in manual quality assurance. As described earlier, the plugins can be used in a multitude of DAWs and operating systems with many different configurations. Therefore, it may be interesting to generate the tests that are used in the manual quality assurance using combinatorial methods, even if no automated test oracle is available.

A topic that is not inherently connected to audio plugins, describes a rather general problem, is to deal with multi-threaded environments and the nondeterminism that is caused by it. As described, audio plugins usually use at least two threads, a processing thread that transforms the input signal, and a message thread that handles parameter-changes. In real-world scenarios it is very common that users of audio plugins modify parameters during audio processing. Because it cannot be guaranteed that the changes are handled exactly at the same time, the behaviour of the plugins in this multi-threaded environment cannot be tested yet. This could be overcome by replacing the thread-handling implemented by the operating system with deterministic scheduling techniques.

Appendix

Bibliography

Aichernig, Bernhard K et al. (2014). "Model-based mutation testing of an industrial measurement device." In: *International Conference on Tests and Proofs*. Springer, pp. 1–19 (cit. on p. 13).

Alaqail, Hesham and Shakeel Ahmed (2018). "Overview of software testing standard ISO/IEC/IEEE 29119." In: *International Journal of Computer Science and Network Security (IJCSNS)* 18.2, pp. 112–116 (cit. on p. 15).

Anand, Saswat et al. (2013). "An orchestrated survey of methodologies for automated software test case generation." In: *Journal of Systems and Software* 86.8, pp. 1978–2001 (cit. on p. 12).

Apfelbaum, Larry and John Doyle (1997a). "Model based testing." In: *Software quality week conference*, pp. 296–300 (cit. on pp. 2, 13).

Apfelbaum, Larry and John Doyle (1997b). "Model based testing." In: *Software quality week conference*, pp. 296–300 (cit. on p. 2).

Arcuri, A., M. Z. Iqbal, and L. Briand (2012). "Random Testing: Theoretical Results and Practical Implications." In: *IEEE Transactions on Software Engineering* 38.2, pp. 258–277. DOI: `10.1109/TSE.2011.121` (cit. on p. 21).

Arcuri, Andrea and Lionel Briand (2011). "Adaptive random testing: An illusion of effectiveness?" In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pp. 265–275 (cit. on p. 22).

Barr, Earl T et al. (2014). "The oracle problem in software testing: A survey." In: *IEEE transactions on software engineering* 41.5, pp. 507–525 (cit. on pp. 2, 22, 23).

Bassil, Youssef (2012). "A simulation model for the waterfall software development life cycle." In: *arXiv preprint arXiv:1205.6904* (cit. on p. 6).

Bell, Adam Patrick (2015). "Can we afford these affordances? GarageBand and the double-edged sword of the digital audio workstation." In: *Action, Criticism & Theory for Music Education* 14.1, pp. 43–65 (cit. on p. 25).

Bertolino, Antonia (2007). “Software testing research: Achievements, challenges, dreams.” In: *Future of Software Engineering (FOSE’07)*. IEEE, pp. 85–103 (cit. on p. 1).

Bhuvaneswari, T and S Prabaharan (2013). “A survey on software development life cycle models.” In: *International Journal of Computer Science and Mobile Computing* 2.5, pp. 262–267 (cit. on p. 7).

Chen, Tsong Yueh, Hing Leung, and IK Mak (2004). “Adaptive random testing.” In: *Annual Asian Computing Science Conference*. Springer, pp. 320–329 (cit. on p. 21).

Chen, Tsong Yueh et al. (2018). “Metamorphic testing: A review of challenges and opportunities.” In: *ACM Computing Surveys (CSUR)* 51.1, pp. 1–27 (cit. on p. 23).

Claessen, Koen and John Hughes (2011). “QuickCheck: a lightweight tool for random testing of Haskell programs.” In: *Acm sigplan notices* 46.4, pp. 53–64 (cit. on p. 24).

Clarke, Paul, Rory V O’Connor, and Murat Yilmaz (2018). “In search of the origins and enduring impact of agile software development.” In: *Proceedings of the 2018 International Conference on Software and System Process*, pp. 142–146 (cit. on p. 6).

Cohen, David M et al. (1994). “The automatic efficient test generator (AETG) system.” In: *Proceedings of 1994 IEEE International Symposium on Software Reliability Engineering*. IEEE, pp. 303–309 (cit. on p. 18).

Cohen, Myra B et al. (2003). “Constructing test suites for interaction testing.” In: *25th International Conference on Software Engineering, 2003. Proceedings.* IEEE, pp. 38–48 (cit. on pp. 17–19).

Colbourn, Charles J and Daniel W McClary (2008). “Locating and detecting arrays for interaction faults.” In: *Journal of combinatorial optimization* 15.1, pp. 17–48 (cit. on pp. 17, 18).

Duran, J. W. and S. C. Ntafos (1984). “An Evaluation of Random Testing.” In: *IEEE Transactions on Software Engineering* SE-10.4, pp. 438–444. DOI: 10.1109/TSE.1984.5010257 (cit. on p. 21).

Fowler, Martin, Jim Highsmith, et al. (2001). “The agile manifesto.” In: *Software Development* 9.8, pp. 28–35 (cit. on p. 8).

Gelperin, David and Bill Hetzel (1988). “The growth of software testing.” In: *Communications of the ACM* 31.6, pp. 687–695 (cit. on pp. 1, 10).

Groce, Alex et al. (2018). “An extensible, regular-expression-based tool for multi-language mutant generation.” In: *2018 IEEE/ACM 40th International*

Conference on Software Engineering: Companion (ICSE-Companion). IEEE, pp. 25–28 (cit. on p. 32).

Gustavsson, Tomas (2016). "Benefits of agile project management in a non-software development context: A literature review." In: *Fifth International Scientific Conference on Project Management in the Baltic Countries, April 14-15, 2016, Riga, University of Latvia*. Latvijas Universitate, pp. 114–124 (cit. on pp. 1, 5).

Hagar, Jon D et al. (2015). "Introducing combinatorial testing in a large organization." In: *Computer* 48.4, pp. 64–72 (cit. on p. 15).

Hajjdiab, Hassan and Al Shaima Taleb (2011). "Adopting agile software development: issues and challenges." In: *International Journal of Managing Value and Supply Chains (IJMVSC)* 2.3, pp. 1–10 (cit. on p. 6).

Hamlet, Richard (2002). "Random testing." In: *Encyclopedia of software Engineering* (cit. on p. 21).

Hussain, Azham, Emmanuel OC Mkpojiogu, and Fazillah Mohmad Kamal (2016). "The role of requirements in the success or failure of software projects." In: *International Review of Management and Marketing* 6.S7, pp. 306–311 (cit. on pp. 1, 5).

Kuhn, D Richard, Raghu N Kacker, and Yu Lei (2010). "Practical combinatorial testing." In: *NIST special Publication* 800.142, p. 142 (cit. on p. 15).

Lei, Yu and Kuo-Chung Tai (1998). "In-parameter-order: A test generation strategy for pairwise testing." In: *Proceedings Third IEEE International High-Assurance Systems Engineering Symposium (Cat. No. 98EX231)*. IEEE, pp. 254–261 (cit. on pp. 16, 18, 19).

Lei, Yu et al. (2007). "IPOG: A general strategy for t-way software testing." In: *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*. IEEE, pp. 549–556 (cit. on pp. 18, 19).

Lei, Yu et al. (2008). "IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing." In: *Software Testing, Verification and Reliability* 18.3, pp. 125–148 (cit. on p. 19).

Lindstrom, Lowell and Ron Jeffries (2004). "Extreme programming and agile software development methodologies." In: *Information systems management* 21.3, pp. 41–52 (cit. on p. 9).

Luo, Lu (2001). "Software testing techniques." In: *Institute for software research international Carnegie mellon university Pittsburgh, PA* 15232.1-19, p. 19 (cit. on p. 10).

Mackinnon, Tim, Steve Freeman, and Philip Craig (2000). “Endo-testing: unit testing with mock objects.” In: *Extreme programming examined*, pp. 287–301 (cit. on p. 11).

Mahmoud, Thair and Bestoun S Ahmed (2015). “An efficient strategy for covering array construction with fuzzy logic-based adaptive swarm optimization for software testing use.” In: *Expert Systems with Applications* 42.22, pp. 8753–8765 (cit. on pp. 16, 17).

Mathew, Tom V (2012). “Genetic algorithm.” In: *Report submitted at IIT Bombay* (cit. on p. 14).

Mathur, Sonali and Shaily Malik (2010). “Advancements in the V-Model.” In: *International Journal of Computer Applications* 1.12, pp. 29–34 (cit. on p. 7).

Mayer, Johannes and Ralph Guderlei (2006). “An empirical study on the selection of good metamorphic relations.” In: *30th Annual International Computer Software and Applications Conference (COMPSAC’06)*. Vol. 1. IEEE, pp. 475–484 (cit. on p. 24).

Melnik, Grigori and Frank Maurer (2006). “Comparative analysis of job satisfaction in agile and non-agile software development teams.” In: *International conference on extreme programming and agile processes in software engineering*. Springer, pp. 32–42 (cit. on p. 6).

Meyer, Bertrand (1992). “Applying’design by contract’.” In: *Computer* 25.10, pp. 40–51 (cit. on p. 23).

Misherghi, Ghassan and Zhendong Su (2006a). “HDD: hierarchical delta debugging.” In: *Proceedings of the 28th international conference on Software engineering*, pp. 142–151 (cit. on p. 20).

Misherghi, Ghavarietyssan and Zhendong Su (2006b). “HDD: hierarchical delta debugging.” In: *Proceedings of the 28th international conference on Software engineering*, pp. 142–151 (cit. on p. 21).

Ness, Brian and Viet Ngo (1997). “Regression containment through source change isolation.” In: *Proceedings Twenty-First Annual International Computer Software and Applications Conference (COMPSAC’97)*. IEEE, pp. 616–621 (cit. on p. 20).

Nie, Changhai and Hareton Leung (2011). “A survey of combinatorial testing.” In: *ACM Computing Surveys (CSUR)* 43.2, pp. 1–29 (cit. on p. 14).

Nurmela, Kari J and Patric RJ Östergård (1993). *Constructing covering designs by simulated annealing*. Citeseer (cit. on p. 19).

Sharif, Bushra, Shoab A Khan, and Muhammad Wasim Bhatti (2012). "Measuring the impact of changing requirements on software project cost: an empirical investigation." In: *International Journal of Computer Science Issues (IJCSI)* 9.3, p. 170 (cit. on p. 6).

Shi, Ce, Yu Tang, Jianxing Yin, et al. (2014). "Optimum mixed level detecting arrays." In: *Annals of Statistics* 42.4, pp. 1546–1563 (cit. on pp. 17, 18).

Stardom, John (2001). *Metaheuristics and the search for covering and packing arrays*. Simon Fraser University Burnaby (cit. on p. 19).

Stolberg, Sean (2009). "Enabling agile testing through continuous integration." In: *2009 agile conference*. IEEE, pp. 369–374 (cit. on p. 9).

Tang, Kit-Sang et al. (1996). "Genetic algorithms and their applications." In: *IEEE signal processing magazine* 13.6, pp. 22–37 (cit. on p. 14).

Tillmann, Nikolai and Jonathan De Halleux (2008). "Pex–white box test generation for. net." In: *International conference on tests and proofs*. Springer, pp. 134–153 (cit. on p. 23).

Yu, Linbin et al. (2013a). "Acts: A combinatorial test generation tool." In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, pp. 370–375 (cit. on p. 19).

Yu, Linbin et al. (2013b). "An efficient algorithm for constraint handling in combinatorial test generation." In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, pp. 242–251 (cit. on p. 19).

Zeller, Andreas (1999). "Yesterday, my program worked. Today, it does not. Why?" In: *ACM SIGSOFT Software engineering notes* 24.6, pp. 253–267 (cit. on p. 20).

Zeller, Andreas and Ralf Hildebrandt (2002). "Simplifying and isolating failure-inducing input." In: *IEEE Transactions on Software Engineering* 28.2, pp. 183–200 (cit. on p. 20).

Zhou, Zhi Quan et al. (2004). "Metamorphic testing and its applications." In: *Proceedings of the 8th International Symposium on Future Software Technology (ISFST 2004)*. Software Engineers Association Xian, China, pp. 346–351 (cit. on p. 24).