Software Construction

RWTH AACHEN UNIVERSITY

The present work was submitted to the Research Group Software Construction

of the Faculty of Mathematics, Computer Science, and Natural Sciences

Bachelor Thesis

Automated Fault Localization for Combinatorial Testing

Automatische Fehlereingrenzung beim kombinatorischen Testen

presented by

Joshua Bonn

Aachen, September 23, 2018

Examiner

Prof. Dr. rer. nat. Horst Lichter

Prof. Dr. rer. nat. Bernhard Rumpe

Supervisor

Konrad Fögen, M.Sc.

Statutory Declaration in Lieu of an Oath

Eidesstattliche Versicherung

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Bachelorarbeit mit dem Titel

Automated Fault Localization for Combinatorial Testing

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Aachen, September 23, 2018 (Joshua Bonn)

Belehrung

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicher ung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen.

Aachen, September 23, 2018 (Joshua Bonn)

Acknowledgment

Writing this thesis would not be possible without the assistance of some people. As such I use this paragraph to thank them for their help.

First, I would like to thank Konrad Fögen for introducing me to the topic of combinatorial testing, and for supervising this thesis. Through the seminar about combinatorial testing and our discussions I learned a lot about the topic. When I had questions about any part of the thesis he always had answers ready.

Additionally, I would like to thank Prof. Dr. rer. nat. Horst Lichter for agreeing to examine my thesis and allowing me to write it at the Research Group Software Construction. I am also grateful to Prof. Dr. rer. nat. Bernhard Rumpe for agreeing to be the second examiner for my thesis.

Last, but not least, I thank my friends and family who supported me during my studies, and my colleagues at work through whom I learned most of what I know about programming in Java.

Joshua Bonn

Abstract

Combinatorial testing is a model-based testing technique. It guarantees that at least one test case covers any combination between values of t or less parameters. Most combinatorial testing is done manually, but there are also attempts to integrate it into automated testing frameworks. In light of the increased movement towards continuous integration/delivery this is extremely important.

After the execution of a combinatorial test suite, the tester is left with the knowledge of which test cases failed. To correct the defect which caused the failure(s), fault localization for combinatorial testing attempts to find the sub-combinations responsible for failures. Numerous fault localization algorithms have already been developed in this active field of research. Some researchers already build a tool to support manual, but not automated, fault localization. The absence of such automation hinders widespread adaption and faster development of new algorithms.

This thesis presents an approach to automated fault localization. After a brief overview of the current state of combinatorial testing research, a general concept is presented. This includes requirements every fault localization automation framework must fulfill. From these requirements a general architecture is derived. As a proof of concept, this thesis presents a Java implementation of said framework. Finally, the implementation is evaluated and possible future areas of research are identified.

Contents

List of Tables

List of Figures

List of Source Codes

1 Introduction

Software testing can be considered one of the most important disciplines in software development; due to the increasing complexity of our programs, it becomes even more critical. As a result, modern software development can cover nearly every part of a program by tests. There are unit, component, integration, end-to-end, performance, and many other kinds of tests. To support developers and quality assurance workers in managing this testing flood, researchers developed numerous approaches; boundary values and equivalence class testing being two famous examples.

Another approach is *combinatorial testing* (CT). It originated in design of experiments and testers can use it to select relatively small test suites covering many input parameter interactions [KR02]. Combinatorial Testing is a model-based technique. Consequently, it does not create test cases based on a real piece of software, but rather through an abstraction of the system under test (SUT). Basically, testers model inputs and configurations as parameters, each having a number of different values (e.g. an operating system can be Windows, Linux, or MacOS) [UPL12]. Now, the general idea behind combinatorial testing is not to test the Cartesian product of the parameter space, but rather all value-combinations of any t or fewer parameters, thus reducing the number of necessary test cases, as each test case contains multiple t-value-combinations [KR02].

A large percentage of combinatorial testing research is focused on developing new algorithms to generate smaller test suites, or reducing the creation time. Other fields include the creation of models, real world applications, constraint handling, test case prioritization, and negative testing [NL11; FL18]. Additionally, test automation is an important topic. As Nie et al. noticed, manual combinatorial testing can quickly become impractical [NL11]. Therefore, multiple frameworks support completely automated CT while integrating it into a regular development environment, one example being JCUnit [Uka17].

In recent years, *fault localization* (FL) also became a prominent topic in CT research [SNX05; Wan+10; Gha+13]. In normal combinatorial testing, the tester receives a list of successful and failed test cases. Since no more information is given, developers have to manually search for detected defects with exemplary input combinations taken from these failed test cases. To help with locating the defect, fault localization attempts to provide the underlying, smaller combination which is responsible for the failure of one or multiple test cases. Instead of an n-value test case, developers can now, for example, work with a t-value sub-combination, and therefore better narrow down the possible defect in programmed code [Gha+15].

While researchers already developed many fault localization algorithms for manual use, the problem of automation remains largely unsolved. Ghandehari et al. developed a tool called BEN which automatically calculates additional test cases based on given test results, but it still requires manual execution of these additional test cases.

Contributions

The goal of this thesis is to develop a general architecture for a completely automated fault localization framework, and implement it as a proof of concept. The architecture should allow for extension through any test case generation, and fault localization algorithm. To reach this goal, the thesis makes the following contributions to combinatorial testing research:

1. It reviews the current state of automation in combinatorial testing, and approaches to fault localization.
2. An architecture for a general automated fault localization framework is presented. This architecture is derived from requirements which such a framework should fulfill. It could be used as a blueprint for developing similar frameworks in different programming languages.
3. A Java program named *CombTest* implements the architecture as a proof of concept. All of CombTest's extension points are described so that researchers could develop new algorithms for use in the framework. To give some examples, CombTest includes a few existing fault localization algorithms, and additionally, this thesis also presents a JUnit5 extension for using CombTest natively in JUnit.
4. Finally, it identifies future work in the context of CombTest.

Structure

This thesis consists of 7 different chapters, the first of which is this introduction. Next, chapter 2.1 introduces general concepts of combinatorial testing and fault localization, which are needed to understand later parts of the thesis. In chapter 3, work related to the concept of an automated fault localization framework is presented. Afterwards, chapter 4 lists objectives and requirements which the framework has to fulfill. These requirements are then transformed into a general architecture. Next, a Java implementation of the general architecture is presented in chapter 5, and possible extension points are shown. This includes instruction on how to integrate custom fault localization algorithms into the framework. Chapter 6 then evaluates CombTest according to the requirements, one quality model, and its ability to support fault localization through different algorithms. Finally, chapter 7 recapitulates the contributions presented in this thesis and proposes future areas of research.

2 Background

Contents

Fully understanding the following chapters requires some background information. Therefore, this chapter gives an overview on three subjects. First, section 2.1 presents basics of the combinatorial testing technique. Later chapters use them in requirements and descriptions of implementation details. Since chapter 4 will then transform these requirements into a general architecture, it is important to know some common architectural design principles. As an example, section 2.2 explains Robert C. Martin's *Clean Architecture*. Lastly, important technologies referenced throughout this thesis are explained in section 2.3.

2.1 Combinatorial Testing

Empirical studies have shown error correction to be significantly more costly in later development phases. Steckline et al. revealed an increase factor of three to eight in the design phase, 21-78 in test and up to 1615 in operation, all relative to correction costs in the requirements phase [Ste+04]. Due to rapid cost rise, it becomes imperative to detect as many errors as possible in early phases. This led to a widespread development and use of sophisticated software testing techniques.

In general, one can categorize these techniques into two approaches: black-box and white-box testing. While the latter derives test cases directly from source code, black-box testing focuses on inputs instead. One famous example would be *equivalence class testing* (ETC). Here, the tester splits all inputs of his/her test into classes for which, in the context of this test, it can be assumed that any contained values are interchangeable. Regardless of which representative value the tester eventually chooses from one equivalence class, in theory they should all lead to the same test result [ND12].

2.1.1 Interaction Faults

When testing a *system under test* (SUT) with multiple inputs, ETC offers two options: either create test cases in such a way that each individual representative appears in at least one test case, or use the Cartesian product between the representatives of all inputs. Employing the Cartesian product leads to a very large number of test cases, while testing every value at least once does not guarantee that most faults are found. For example consider the following piece of code:

```java
1  public void method(int waterLevel, double humidity, ...) {
2    if (waterLevel >= 100) {
3      // working code
4    } else {
5      if (humidity > 10) {
6        // non-working code
7      }
8    }
9  }
```

Source Code 2.1: InteractionFault.java

The fault (line 6) would only be detected by a test case having a water level lower than one hundred and a humidity higher than ten. While we may have a test case covering this particular fault by chance, it would be better to systematically test for all interaction faults of a certain size. Combinatorial testing provides just that. The idea is to make sure that all possible combinations between a fixed number of so called parameters are covered by at least one test case. To better understand this, the next sections will first introduce some terminology.

2.1.2 Parameters and Values

The most important input for combinatorial testing are the parameters and their values. They determine which concrete input values later appear in test cases. Generally speaking, parameters are the inputs for tests or configure the test's environment, depending on the current use case of combinatorial testing. Examples for parameters are the operating system, the web-browser on which sites are loaded, but also real input values to systems like the humidity and water level from Listing 6.

Only giving parameters to a combinatorial test does not do any good. No systematic tests can be designed when given only information like operating system, browser, humidity and water level. For each parameter the tester has to decide on real (representative)

Name	**Values**				
OS	Windows	Linux	MacOS	Android	iOS
Browser	Chrome	Edge	Firefox	Safari	
Ping	10 ms	100 ms	1000 ms		
Speed	1 KB/s	10 KB/s	100 KB/s	1000 KB/s	

Table 2.1: Parameters for the running example

values which can be tested. For example, values for operating system could be Windows, MacOS, Linux and Android. When each parameter now has its values, combinatorial testing can derive concrete test cases. Table 2.1 depicts a few parameters with their values for testing a browser game in different conditions. It will serve as a running example in later sections.

There are some conditions which parameters for combinatorial tests have to meet. Testers should always give at least one parameter, otherwise a combinatorial testing does not make any sense. Additionally, each parameter should have at least two values. If it would have none at all, the parameter could just be removed, and if there is just one value, the parameter is a constant, and only makes the development of combinatorial test suites unnecessarily harder.

An interesting sub-topic of combinatorial testing is how testers can find values for parameters. Generally, they can employ other testing techniques such as ETC or boundary-value analysis. Often, some parameters don't have a complex value space, like operating system, and the software requirements directly specify all possible values.

Throughout this thesis, n will denote the number of parameters.

2.1.3 Combinations

A test case in combinatorial testing is a combination of values, where there is exactly one value from each parameter. For example, (Windows, Edge, 1000 ms, 1 KB/s) is a test case for the parameters from Table 2.1. Additionally to full test cases, terminology in combinatorial testing also denotes incomplete ones as combinations. This means some parameters do not have to be set. In that case, they are denoted with a dash in the tuple. A combination of Windows and Firefox then looks like this: (Windows, Firefox, —, —).

That is also an example for a 2-value-combination. t-value-combinations in general are combinations in which t parameters have a value. The space of all possible combinations forms a containment hierarchy. A combination c_1 contains c_2 if all values which are set in c_2 have the same value in c_1. Consequently, (Windows, Firefox, —, 100 KB/s) contains (Windows, Firefox, —, —) and (Windows, —, —, —).

2.1.4 Testing Strength

Now that last section explained all basic terminology, this section describes the actual process of combinatorial testing, and the philosophy behind it. Besides parameters and values, the most important configuration option for combinatorial tests is the so called *testing strength*, often denoted with t. It describes to which extend combinations of parameter-values should be tested. A given testing strength of t means that all for possible t-value-combinations there should be least one test case containing it (also called t-way-testing). The advantage of testing only all t-value-combinations and not all n-value-combinations like strong ETC, is that it needs significantly fewer test cases [KWAMG04]. If we look at the test case (Windows, Edge, 1000 ms, 1 KB/s), we can see that it contains 6 different 2-value-combinations: (Windows, Edge, —, —), (Windows, —, 1000 ms, —), (Windows, —, —, 1 KB/s), and so forth. In fact, CombTest's IPOG algorithm implementation generates just 23 test cases for 2-way-testing, instead of $5 \cdot 4 \cdot 3 \cdot 4 = 240$ for exhaustive testing.

The difficult question which remains is: What t to choose? Should one only test all 2-value-combinations to have high confidence in the SUT? Or do you need $t = n - 1$ or $t = n$? Some empirical studies researched this question. For example, Richard Kuhn et al. examined the bugs found in different systems to check which t would suffice to detect them [KWAMG04]. Figure 2.1 depicts the results.

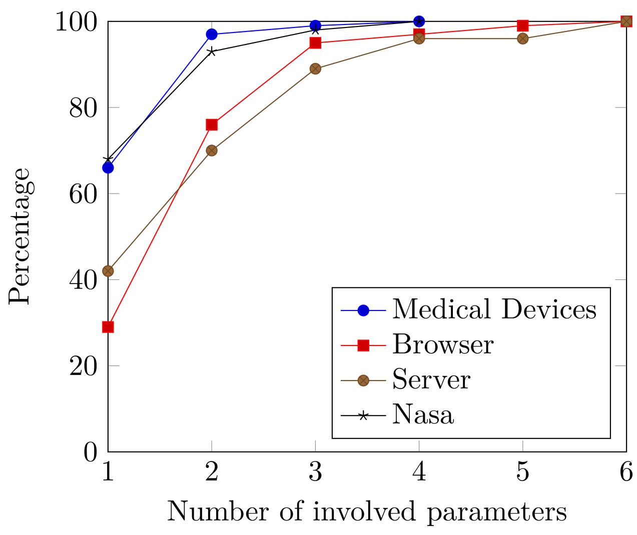

Figure 2.1: The percentage of errors over the number of involved parameters for four different software systems as reported by [KWAMG04]

One of the most noticeable results is that all defects would have been found by 6-way-testing. Considering the number of parameter and values typically found in such complex systems, this reduces the number of test cases significantly. One can also see the decline of new errors discovered by testing with an higher strength. It is generally assumed that for nearly all systems only a very small t is needed [KWAMG04]. And here the strengths of combinatorial testing really comes into play.

Finding a minimal test suite which contains all possible t-value-combinations is an NP complete problem [YZ09]. Therefore, most algorithms currently developed only provide a near optimal test suite. Usually combinatorial testing employs either greedy or heuristic algorithms. Examples include *IPOG* and *simulated annealing for combinatorial testing* [Jia+07; PN12].

2.1.5 Constraints

After an algorithms constructs a test suite, the tester has to execute all test cases. Otherwise, some t-way combinations may not be tested. This can be problematic, or impossible, because of constraints on parameters and values. For example, the IPOG algorithm could include (Windows, Safari, 100 ms, 1 KB/s) in a 2-way-testing suite for the parameters in Table 2.1. Currently, Apple does not provide a Safari installation for Windows computers. Hence, we cannot execute the test case. Deciding not to execute it could mean that perfectly valid 2-way-combinations like (Windows, —, 100 ms , —) could potentially go uncovered. This is where constraint handling comes into play.

There are numerous ways to deal with constraints in combinatorial testing. They can be categorized into three different approaches. The *abstract parameter* and *sub-model* method are both applied to the parameters and values before a test suite is generated. They modify parameters or split values into different models to ensure that no invalid combinations can occur in the generated test suites. Another approach is the avoidance of invalid test cases during the generation itself. Researchers have modified IPOG and other algorithms to accept constraints in the form of explicit invalid combinations and mathematical expressions like $OS = Windows \Rightarrow Browser \neq Safari$. In a last method, multiple valid test cases preserving the t-value-combinations replace each final test case containing an invalid combination [GOM06].

In practice, most tools provide support for the avoidance method. It has a very good usability, because the user simply has to list all constraints and invoke the tool like s/he would in a constraint-free environment. For all other approaches, an additional invocation is necessary. Additionally, this method generates relatively small test suits. While all constraint handling techniques introduce a test case overhead, the number of additional test cases varies per method [GOM06].

Together, the parameters, testing strength, and constraints form the *input parameter model* (IMP), a general input for combinatorial tests.

2.1.6 Negative Testing

Usually, there is a distinction of testing with valid values to check for a correct result, and testing the SUT with unexpected input to assure correct error handling. The second variant is called negative (combinatorial) testing. Research presents two different approaches, which one can combine if necessary.

One way to test for correct error handling is to include some invalid values to each parameter. For example, testers can use robustness boundary value analysis to find

Test Case				
OS	**Browser**	**Ping**	**Speed**	**Result**
Windows	Chrome	10 ms	1 KB/s	pass
Linux	Chrome	100 ms	10 KB/s	pass
MacOS	Chrome	1000 ms	100 KB/s	pass
Android	Chrome	10 ms	1000 KB/s	fail
Linux	Chrome	10 ms	1000 KB/s	fail
Android	Firefox	100 ms	10 KB/s	pass
iOS	Edge	100 ms	1000 KB/s	pass
MacOS	Chrome	10 ms	1000 KB/s	fail

Table 2.2: Excerpt of a possible test results list for the example in Table 2.1

values which the SUT should not accept. The algorithm used to generate test suits then needs to add additional test cases testing only these invalid values. It needs to treat them separately, as otherwise the so called masking effect can occur, where valid t-value-combinations are not tested for a success scenario because they are only present in test cases which fail. PICT, a test cases generation tool, supports this approach [Cze06].

A rather new approach is creating invalid test cases using already specified constraints. At first, the tester splits all previous constraints into two categories: forbidden and error. The first one includes all constraints removing combinations from test case which really cannot be applied to the model because they are impossible to construct (e.g. $OS = Windows \Rightarrow Browser \neq Safari$). Error constraints describe those combinations which could be entered as input values, but invoke some sort of error. For example, telling the server that the connection has a negative ping should result in an error. For each of these error constraints, negative testing can construct test cases by negating the current error constraint, and therefore construct a new set of constraints under which a combinatorial test algorithm can now construct additional test cases. For example, when the first error constraint is negated, all test cases constraint-aware IPOG generates in this pass must only violate this one negated constraint and no other. Consequently, testers can later attribute failures in negative test cases to specific error constraints [FL18].

2.1.7 Fault Localization

After a tester executes a combinatorial test suite, s/he may receive a result list like the one in Table 2.2. Next, the software's developer has to locate the defect in the source code. While this could be relatively simple if the IPM only includes few parameters, the difficulty rises with the number of parameters. For example, in a combinatorial test with over 40 parameters, having one failed test case does not really help a developer to locate the defect and fix it. This is where *fault localization* for combinatorial testing comes into play.

Failure-Inducing Combinations

The main problem with the test results from Table 2.2 is that there could be multiple root causes for each failed test case regarding the parameters involved. Combinatorial testing research calls such a root cause a *failure-inducing combination*. These combinations do not have to be complete, so some values do not need to be set. If a combination is failure-inducing, each test-case which contains it will fail. If a developer were to know that the failure-inducing combination (—, Chrome, 10 ms, —) causes all failing test cases, s/he could narrow down the possible defect locations in the code. This is why fault localization is needed in combinatorial testing.

Approaches

Over time, researchers developed several algorithms which assist the tester during the location of failure-inducing combinations. These algorithms can be split into two categories: non-adaptive and adaptive.

Non-Adaptive Nearly all fault localization algorithms introduce some additional test cases to further narrow down which combinations can possibly be failure-inducing. This is needed since the original test suites do not hold enough information as section 2.1.7 described. Non-adaptive algorithms do not generate these test case based on test results; instead, they introduce additional test cases in the generation step [Zhe+16].

One example for a non-adaptive method is *locating and detecting arrays* (LDA) [CM08]. It uses information about parameters and values to create an initial test suite which allows for a detection of failure-inducing combinations via failure of specific test cases.

While non-adaptive methods have the advantage of generating only additional test cases in the normal generation step, they usually have other limitation which prevent use in practice. Examples are a maximum number of detectable failure-inducing combinations, or a low testing strength [Zhe+16]. Additionally, the test suite is always larger than normal, even if all test cases pass. This results in a higher overhead.

Adaptive This thesis focuses on adaptive fault localization algorithms and their complete automation. In contrast to their non-adaptive counterparts, these algorithms dynamically generate more test cases based on results from the initial test suite. To further understand adaptive fault localization, section 3.2 explains one algorithm.

Adaptive algorithms have some advantages over non-adaptive variants. Since they base additional test cases on initial test results, none are generated if there were no failures. This reduces the time spent for testing if all test cases pass. Additionally, one needs fewer test cases since the algorithms know which combinations to focus on during the generation of further test cases [Zhe+16]. On the other side, adaptive algorithms introduce an additional testing step which developers have to implement in combinatorial testing tools and automate as much as possible. That is the goal of this thesis.

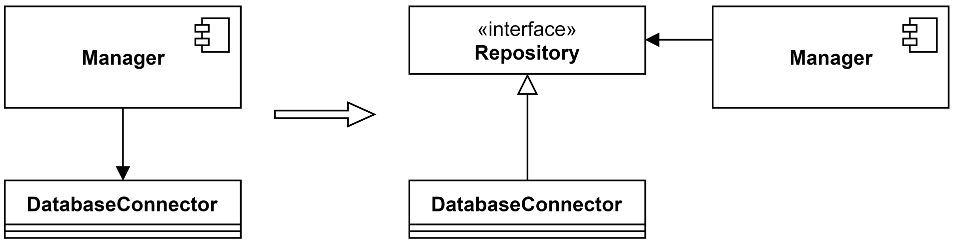

Figure 2.2: Class diagram for the dependency inversion principle

2.2 Clean Architecture

Many developers will say that one of the most important parts of a software product is to have a good *software architecture*. There is, however, the problem of defining what a (good) software architecture actually is. In an effort to document this, the Software Engineering Institute of the Carnegie Mellon University compiled a list of about 30 definitions [SEI10]. Nearly all of them agree that it is some form of abstraction from actual implementation details and data structures, dealing with general system components. On the other hand, Taylor et al. take a more process-oriented approach and define software architecture as a set of design decisions [TMD09]. Additionally, Robert C. Martin defines the goal of software architecture as “to minimize the human resources required to build and maintain the required system” [Mar17].

Even though a multitude of definitions exists, nearly all papers or books agree that you should follow a set of rules to reach a good architecture. While this does not mean an architecture is necessarily good if one follows all rules, most of them are at least a good indicator. In his book Clean Architecture, Robert C. Martin defines such a set of concepts and rules which all accumulate to his version of a good architecture. The next sections will explain the most important ones, which the architecture presented in section 4.2 uses.

2.2.1 SOLID- and Component Principles

Martin first grouped the *SOLID* principles together in 2000 [Mar00], and generally applies them just above the programming level. They define how one should design a good software module with concern to boundaries, and how multiple modules should interact with each other. While architecture normally works on component levels, the SOLID principles are still very important to good architecture, as one can also apply them to whole components. This can be seen with Martin’s component principles, which generalize the SOLID principles. The following list shortly explains each SOLID principle:

- *Single Responsibility Principle*: A module should only be responsible for one part of the functionality. This means, that it only has one reason to change; that is when the functionality it is responsible for changes.

- *Open-Closed Principle*: At the same time developers should be able to add new functionality to a module (open), but other programmers need to depend on existing code, which therefore should not be modified (closed).

- *Linkov Substitution Principle*: This principle is probably best explained by interfaces in any programming language (or abstract classes). It states that every part of a program should be interchangeable if the new part adheres to the old contract. In Java, you could imagine an interface to be such a contract, and you should be able to switch the concrete class implementing an interface without the system's code needing to change.
- *Interface Segregation Principle*: A module should not be forced to depend on any functionality it does not use. Consequently, programmers should split big interfaces into smaller interfaces if they combine separate parts. If one client is then only interested in one part, it does not need to depend on the big interface. As a result, the client and interface module become more decoupled.
- *Dependency Inversion Principle*: A module should not rely on low-level modules, but on abstractions. Figure 2.2 visualizes this concept. The `Manager` no longer depends on the low-level `DatabaseConnector`, but on the high-level `Repository` abstraction. As a result, developers can easily switch out concrete implementations (Linkov Substitution Principle). This pattern is used throughout the entire architecture and realization and is therefore very important.

Derived from SOLID, Martin presents three principles describing the cohesion classes should have inside individual software components. It is not possible to achieve all of these principle at the same time, so a system's architect has to decide which ones s/he will use.

- *The Reuse/Release Equivalence Principle*: This is a very weak principle stating that the grouping of classes into components should make sense, both to user and developer, from a release perspective. To be exact, this means that modules which can be independently released should not be inside the same component.
- *The Common Closure Principle*: All modules in one component should just have one reason to change. This is equivalent to the Single Responsibility Principle, just for components instead of modules.
- *The Common Reuse Principle*: Components should only consist of modules which are reusable together. Like with the Interface Segregation Principle, clients should not have to depend on components (code) they do not really need.

2.2.2 Details

One of the topics Martin argues passionately about are frameworks and databases. In his opinion, looking at a software product's architecture should not show you which framework the developer used, but rather what kind of system it is. He compares this to building's architecture where you can normally see what type of building a blueprint depicts and not how it was constructed.

As such, no application code should actually depend on any specific framework, database, or front-end presentation mode. Instead these parts are just small details in a software product, and developers can "easily" change them. One can directly derive these rules from the SOLID principles, such as the Linkov Substitution- and Dependency Inversion Principle.

With all details at the outermost layer, Martin then defines a Clean Architecture to be made out of concentric circles, each depending only one the next inner circle, with the dependencies never going outward. As a result, developers can easily change all details, and business rules only depend on entities, which depend on no other components at all [Mar17].

2.3 Technologies

As a proof of concept, chapter 5 will later explain the implementation of a fault localization framework. To develop such a framework, one needs multiple technologies. This chapter will present the most important ones, namely *Java*, a popular programming language, the build- and dependency management tool *Maven*, and *JUnit*. Each section will only give a brief overview of the most important features this thesis uses. For more information, they provide links to common resources on the Internet. If you are already familiar with these technologies, feel free to skip to chapter 3.

2.3.1 The Java Programming Language

Sun Microsystems first introduced the Java programming language in 1995. It is an object-oriented language with many build in useful libraries for different disciplines, like database connections (JDBC), networking, multi-threading and much more [HC02]. Since its introduction continuously rose in prominence, and is now leading the TIOBE index as the most popular programming language [TSB18]. Today, it is mostly used in the backend of popular services, for example, by Netflix and Twitter [OC18]. While the standard Java library alone is very good, where the language really shines is its ecosystem. There are thousands of libraries for common use cases ranging from machine learning to web API development [Sky18; PS17].

2.3.2 Maven

Nearly every language has a tool for dependency management and complex build processes. For a long time, developers used Apache Ant, an XML-based build tool. During the development of the Jakarta Turbine project, Apache programmed *Maven* as a successor to reduce XML-duplication and ease general complex management [TASF04]. In a non-representative survey in 2017, Maven had a market share of 76% [BS17].

Maven, like Ant, is an XML-based tool and offers sophisticated build, dependency management, and plugin support. Developers store all information necessary to start Maven in a *pom.xml* file. For a small project, it may look like the following example:

```xml
<project>
  <modelVersion>4.0.0</modelVersion>

  <groupId>de.rwth.swc.sample</groupId>
  <artifactId>sample</artifactId>
  <version>1</version>

  <dependencies>
    <dependency>
      <groupId>de.rwth.swc.combtest</groupId>
      <artifactId>combtest-junit-jupiter</artifactId>
      <version>1.0-SNAPSHOT</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Source Code 2.2: pom.xml

This file declares a project called `sample` in its first version. It uses just one dependency: CombTest's JUnit extension. The `<scope>test</scope>` part specifies that only test classes may use CombTest, and that maven will not package it with code deployed to production.

Maven also allows the definition of sub-modules. This essentially means that developers can split a project into multiple sub-projects, and keep common information (e.g. dependency versions) in a parent *pom.xml*.

Maven splits it build process into seven phases: `validate`, `compile`, `test`, `package`, `verify`, `install`, and `deploy`. The `test` phase runs all unit tests, `install` makes the project available for local use. This means other projects can import it using the dependency management system. Further information about the Maven life cycle is available at [TASF18].

2.3.3 JUnit(5)

In earlier chapters this thesis established that not testing software can be very dangerous. Since testing is such an important task, and many elements are reusable across projects, Kent Beck established the xUnit family. Fist, there was only a framework for Smalltalk testing, but later on, frameworks for all famous programming languages appeared [Bec97]. Often, the name of these testing frameworks begins with the first character of a programming language's name; hence, there is JUnit for Java, RUnit for R, and SUnit for Smalltalk.

Currently, *JUnit5* is the latest version after JUnit4 became too complex to adapt it to new features such as Java 8's lamdas [Phi15]. It introduces many interesting new concepts over JUnit4. One of the most important points is a clear differentiation between JUnit Jupiter and JUnit Platform. Jupiter is only one implementation of a `TestEngine`, and developers can add other test engines dynamically. The platform does not know anything about the execution of tests. Instead, it can only tell test engines to discover tests or execute specific ones. Due to the separation of concerns, developers can still use old JUnit4 tests, since there is a JUnit4 `TestEngine`.

Writing one test is very simple and requires only the use of one annotation:

```java
@Test
void test() {
  assertEquals(9, square(3));
}
```

Source Code 2.3: SimpleJUnitTest.java

Where JUnit5 truly shines is in its extension system. For example, dynamic test generation is possible via `TestTemplateInvocationContextProvider`, and for passing parameters to test cases developers can use a `ParameterResolver`. Combined, this functionality allows for multiple invocations of the same test method with different values [Bec+18]. Parameterized tests are a good example:

```java
@ParameterizedTest
@ValueSource(ints = {2, 4, 6})
void even(int number) {
  assertEquals(0, number \% 2);
}
```

Source Code 2.4: ParameterizedJUnitTest.java

As this example demonstrates, parameterized tests are very near to what combinatorial tests could look like in JUnit5. One test method is reused multiple times for different input parameters. Testers define these parameters by some external source (`@ValueSource` or an input parameter model).

3 Related Work

Contents

This chapter provides an overview of the current state of research connected to automated fault localization for combinatorial testing. Until now, there has been no work connecting both the automation of combinatorial test suite generation and fault localization. However, both approaches are actively researched separately. The next sections will present the current state of research.

3.1 Automated Combinatorial Testing

One of the most researched topic in the field of combinatorial testing is the actual generation of t-way test suites [NL11]. Often, researchers only implement these algorithms as a proof of concept. Nearly always, those implementation communicate with the user over a CLI or GUI [Cze06]. This means the user has to treat the generation step separated from actual testing. When these tests are not automatable, this is acceptable. With an increase in automatic tests however, automated combinatorial testing becomes more attractive. Instead of having to copy the whole generated test suite into the employed test automation software, it performs the execution by itself, and therefore tester can quickly change parameters, constraints and testing strength [UQ17].

One of the biggest example for an automated combinatorial testing tool is *JCUnit* [Uka17; UQ17]. It is based on the popular, but outdated, Java testing framework JUnit4. Users can specify multiple parameters as methods which return factories. These factories then provide concrete values to JCUnit at runtime [Gam+95]. Due to the general definition of the factory interface used to provide values, one can use any valid Java object or primitive as a value. This means even less work for the developer, as s/he does not have to convert textual representations to actual objects, as most other combinatorial testing framework require [Ret14; Cze06; Uka17]. Additionally to factories for parameters, testers can also use the familiar concept of functions to model constraints. With the correct annotations and a boolean return value, the framework can evaluate whether a combinations is valid solely based on a method. Since Java is a Turing complete language, developers can write every possible constraint.

JCUnit does not support fault localization in any form. The results it returns are the failed and passed test cases shown by JUnit. Additionally, one can not provide a custom

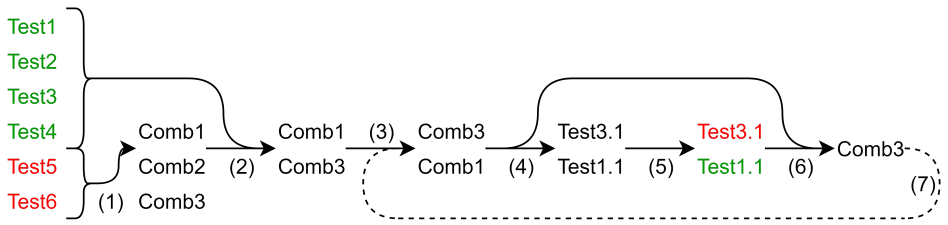

Figure 3.1: BEN's basic algorithm

generation algorithm [Uka17].

3.2 Fault Localization Tools

Researchers already developed many algorithms for fault localization. Examples include *BEN*, *AIFL*, *IterAIFL*, *Improved Delta Debugging* (IDD) and *comFIL* [Gha+13; SNX05; Wan+10; LNL12; Zhe+16]. While most of them were implemented as part of the respective paper, nearly none of those implementations are available publicly. Until now, only Ghandehari released the tool BEN for public use. While BEN also tries to find the defects' location in the provided source code, it first performs a localization of failure-inducing combinations.

Figure 3.1 show the basic algorithm. First, BEN takes all failed test cases from the initial generation and extracts possible failure-inducing combinations (suspicious combinations) (1). Next, it uses the successful test cases to reduce the list of suspicious combinations (2), as no failure-inducing combination can appear in a successful test case. In (3), BEN uses an internal ranking schema to order the combinations according to their probability of being failure-inducing. To achieve this, it assigns and internal probability of being in a failure-inducing combination to each parameter-value combination (e.g. OS=Windows) based on occurrences in failed test cases and additional criteria. Averages over parameter-value combinations in suspicious combinations then give each combination an accumulated probability of being failure-inducing. For the first few combinations in the ranking, BEN generates new test cases which contain these combination (4). Results from the test cases' execution is then used to further reduce the number of suspicious combinations (6). The algorithm can then start over at step (3) until it meets a stopping condition (7) [Gha+13].

To use BEN, the user has to jump between the actual SUT and BEN's GUI. S/he must enter each iteration's test result into BEN via files, and receives further manually executable test cases. The user then needs to execute all additional test cases generated into a text file. This cycle can repeat a few times.

4 Concept

Contents

This chapter introduces the general concept of a framework executing automated fault localization for combinatorial testing. Section 4.1 explains general requirements which the framework needs to fulfill to guarantee good fault localization in an automated setting. Next, section 4.2 describes an architecture which satisfies all requirements and the important concepts of Clean Architecture.

4.1 Requirements

Before one can actually begin the development of an architecture for a program, one needs to know the fundamental requirements placed upon it. They define what functionality the system offers (functional requirements) and how it does so (non-functional requirements). To define them, it is best to first go over the goals and objectives.

4.1.1 Goals and Objectives

As we saw in chapters 2.1 and 3, many algorithms for combinatorial test suite generation and fault localization exist. Researchers have also automated the general combinatorial test execution. For fault localization, no such automation attempt has yet been made. This thesis provides that missing link. Therefore, some very general objectives can be set up for the concept.

Firstly, the framework should reuse existing algorithms. Reuse is one of the core concepts of software development, and prevents that the same work is done multiple times. For example, the framework should not attempt to implement new algorithms for fault localization, but should provide interfaces to use existing ones. Secondly, the framework should be forward facing with regards to future extensibility. As combinatorial testing is a field of very active research, new ideas are bound to emerge, and this framework would be useless if it did not support them. One example for these new ideas is negative testing with the help of error constraints as presented in [FL18].

Despite being very actively researched, combinatorial testing is not known by many people outside the field of software testing. As such, the framework should be easy to use to provide a good introduction to the field. If there are many hurdles to starting with combinatorial testing, mainstream developers will not use it.

Additionally to normal test developers, the framework should also make life easier for combinatorial test researchers. At the moment, there is no tool which allows researchers to test their new algorithm extensively. The presented framework should remove much of the boilerplate code needed to get a test generation or fault localization algorithm to run with arbitrary input data.

All these objectives are very important for the requirements' definition and can therefore be found in many of them.

4.1.2 Users

To find all requirements it is often helpful to look at the possible users of the program. In case of the framework this thesis presents, there are just two important user groups which the objective already mentioned: testers and algorithm developers. The listing of all requirements is structured into these two groups. For every group, all functional and non-functional requirements are listed, each of which is explained in depth. Some requirements are relevant for both user groups. In that case, the list only contains them once for easier identification, but the requirement then includes all important aspects for both user groups.

4.1.3 Tester Requirements

This section presents all requirements which are important to people who either write tests or have to execute them.

F1 Configuration of the Input Parameter Model The user can specify the input parameter model for a combinatorial test. This includes the following parts:

a) All parameters and values. Per IPM, the user has to give at least one parameter. Each parameter must have at least two values.

b) The testing strength as described in section 2.1.4. The user can specify it as any number in the range of 1 to the number of parameters (inclusively).

c) Constraints on parameters and values.

If a user provides an input parameter model which does not meet all criteria described above, the framework must notify the user. For every test, the user can specify a new IPM, but s/he may also reuse an existing one.

F2 Configuration of Test Case Generators The user can specify algorithms which generate initial combinatorial test cases. Since there are multiple, independent ways to generate test cases (like normal IPOG tests and negative test cases), the system must allow the user to specify an arbitrary number of generators. The user can reuse every generator across multiple combinatorial tests.

F3 Configuration of the Fault Localization Algorithm Additionally to test case generators, the user can also specify an algorithm which the framework must use for fault localization. If the user does not configure any fault localization algorithm, the fault localization feature is disabled.

F4 Initial Generation When the user starts the combinatorial test, first, the framework calls all generators to get their initial test cases. For this, the framework passes the IPM specified by the user to every generator to make sure that all test cases are meant for the same model. If the user specified no generators, the system returns no test cases. It is important that it does not crash in such a case.

F5 Test Results The user must have an option to pass test results to the framework. Without test results, no fault localization would be possible. The most important part of this framework is that there is a way to pass test results to the framework automatically. Otherwise, no automated fault localization is possible. There are two general approaches the framework can take (non-exclusively):

a) The user gets all generated test cases from the framework and has a way to give all test results back to the framework. This means that some program evaluates the test cases' result outside of the framework.

b) The user provides an evaluation method as a callback. The framework itself can then call this method whenever it requires the result to a specific test case. In this case, the user only needs to interact with the framework to start the combinatorial test. With the other variant, s/he may needs to provide test results multiple times if s/he enables fault localization.

F6 Conditions for Fault Localization The system must only use fault localization if all of the following conditions are fulfilled:

a) The user provides a fault localization algorithm via requirement F3.

b) At least one test in the initially generated test suite failed. If no test case failed, fault localization would never find any failure-inducing combinations.

F7 Fault Localization Iterations After the user provides all needed test results, the framework propagates them to the specified fault localization algorithm, which can then generate further test cases. The test results needed for propagation are defined as follows:

a) In the first iteration, the framework needs results for all test cases initially generated by the generators from requirement F4.

b) In every following iteration it needs the results for all test cases generate by the fault localization algorithm in the previous iteration.

F8 Fault Localization Stopping After the user provided fault localization algorithm generates no more additional test cases, the framework must not require execution of further test cases.

F9 Test Result Caching The framework must cache all test case results per combinatorial test. Depending on the SUT, some test cases have a long execution time. That is why the number of test cases which have to be executed should be as low as possible. During fault localization, and due to multiple test case generators, duplicate test cases can appear. In this case, the framework must not require the user to execute the same test case again, but rather reuse the previously known result. The underlying assumption of this requirement is that the same test cases always produce the same result. Otherwise fault localization would not work reliably.

F10 Life Cycle Reporting The framework must notify the user of life cycle events for a combinatorial test. Otherwise, the user would not know in what state the test is (initial testing or fault localization). The framework must report the following events:

a) Generation of initial test cases via a generator the user provided (requirement F2).

b) Complete finished execution. This means that either the initially generated test cases have all been executed if no fault localization is enabled (requirement F6), or fault localization has finished (requirement F8).

c) The start of the fault localization process.

d) The generation of new test cases by the user provided fault localization algorithm.

e) The fault localization process has completely finished. This report must include the failure-inducing combinations which the fault localization algorithm found.

f) The execution start for one specific test case.

g) The execution finish for one specific test case. This report must include the test result.

All fault localization specific life cycle reports (c, d, e) must only occur when fault localization is enabled (requirement F6). Additionally, all reports concerning the execution of one test case (f, g) are only done once per test case regardless of how many times it was generated, since the framework uses caching (requirement F9).

F11 Event Reporting Additionally to life cycle reporting, the framework must provide a general reporting interface which the generators and fault localization algorithm can use. Through this interface users can collect additional information for execution statistics.

N1 Use of native Objects The user must be able to express the values for each parameters with types from the underlying programming language. In a fully automated environment, combinatorial tests could need any type of value as an input. If the user only has the option to choose between a small, arbitrary, and framework-made selection of types, s/he has to program the conversion logic her/himself. This would result in more error-prone programming and duplicate code. For simplicity reasons, the framework must be solely responsible for supplying all parameters in the right type.

N2 Performance-Independence from Object Sizes The general execution time and performance of the whole framework can not depend on the size of the objects used as values. For example, in Java the performance of `equals` and `hashCode` determine the performance of a comparison between objects. Users of the framework must not have to concern themselves with performance optimization in those methods. In a worst case scenario this could lead to developers changing these methods in classes used in production.

N3 Reuse Configuration While in general the user does all configuration only for one specific combinatorial test, s/he should be able to reuse fault localization algorithm, generator, and reporting settings across multiple combinatorial tests.

N4 Test Framework Agnostic A fault localization framework must not depend on a specific testing framework. A violation of this requirement would be a direct violation of Clean Architecture, where the framework should just be a detail.

4.1.4 Algorithm Developer Requirements

Now that last section explained all requirements for testers, there are still a few requirements left a fault localization framework must fulfill for use by algorithm developers. These requirements focus on extendability.

N5 Developing Fault Localization Algorithms The framework must provide an interface such that combinatorial test researchers can develop and use their own fault localization algorithms. One must be able to achieve this goal by implementing just one interface. The framework needs to provide all necessary information to the algorithm through this interface. As it is not possible to predict what information future algorithms need, the framework should at least support all current ones. This includes BEN, AIFL, IterAIFL and IDD, as mentioned in section 3.2.

N6 Developing Test Case Generators Additionally to fault localization algorithms, developers must also have the option to write custom test case generators. For example, generators for an IPOG generation and negative test cases as described in [FL18] must be possible. Each generator only needs to implement one interface which provides the generator with all necessary information. This includes the IPM, and a report mechanism as requirement F11 described.

N7 Developing Reporters With the multitude of logging and reporting frameworks which are available, some developers may need to write custom report adapters to integrate the framework into their development environment. This means, that life cycle (requirement F10) and event reporting (requirement F11) must include the capability to write custom reporters. For example, in the Java programming language many developers use SLF4J, so they could need a `FrameworkReportingToSlf4JAdapter`.

4.2 Architecture

Now that all requirements for a general fault localization automation framework have been stated, this chapter describes the derived architecture. The architecture is generally platform and programming language independent, so one could also implement it in another language than Java. This section is structured as follows: First, an overview of the very general architecture is given. The following sections all describe one part of the architecture in greater detail.

4.2.1 Overview

This section describes the general architecture of the fault localization framework. This includes the component based structure and general process information. Each architectural decision is documented and explained. The following subsections will cover parts of the general architecture and mostly refer to Figure 4.1. This diagram depicts all important framework components. Model/API is responsible for the communication with the framework's user; This user can also be another testing framework. The GenerationManager is responsible for storing all test results and deciding when to call the fault localization and generation algorithms.

Engine and Model Split

Maybe the most important design decision is the split between model and engine. Figure 4.1 marks each of the framework's big parts. The engine includes basic reporting capability and the business logic within the GenerationManager. On the other side we find Model/API, some reporting based on said model, and a Conversion-component, last of which is the main reason for this split into to parts.

While splitting an architecture into multiple parts is nothing out of the ordinary, the place of this split is determined by the requirements. Specifically, non-functional requirements N1 and N2. To recap: Developers must be able to define values in native

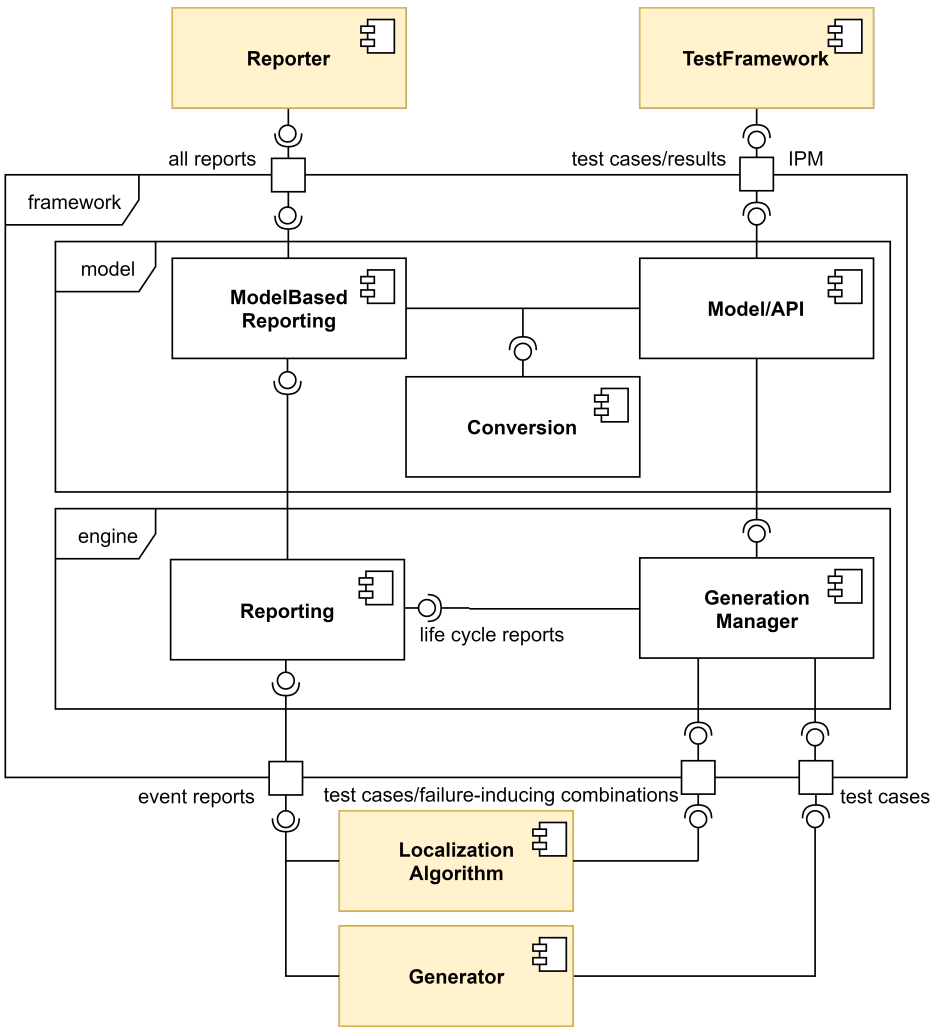

Figure 4.1: Component diagram for the general framework architecture

objects, but they must not negatively impact the execution speed. This results in the need to convert all values into some other representation for which the system can manage the performance independent of any actual values. It makes sense to have a real clean cut along this line. The actual API which the user or test framework interacts with needs to be able to accept objects as values and work with them. Generation management does not need to know what actual objects are used. In the end, all test cases which the user gives back to the framework need to be in object-form again. This means a continuous conversion between objects and internally used types is necessary, so it makes sense to outsource these task to a custom component — the Conversion-component.

As a result, the framework calls all fault localization algorithms and initial generators with the internal representation types. This releases algorithm developers of the responsibility to either convert all values themselves, or go with a decision to make no conversion at all, which would hurt performance quite substantially.

Another advantage of the split is that, theoretically, one could develop different models, each using the same engine. This means that while the standard implementation of the framework could be via classes of types `InputParameterModel`, `TestCase`, etc., there could also be another model which loads information from Xtext files.

On the other side, having a conversion mechanism for all test case introduces a slightly increased difficulty in reporting. Since some reports contain actual domain model objects like failure-inducing combinations (requirement F10), the conversion has to take place here to. This is why a ModelBasedReporting-component is necessary. Its sole purpose is to convert reports from the engine's Reporting-component into formats which the user can interpret. To achieve this, it uses the same Conversion-component as the normal API.

Outside Extension/Interfaces

The general architecture has five points which connect to externally programmed components. This thesis denotes them with the term *extension point*. Two of these offer functionality to outside of the framework while the rest uses functionality provided by external components.

Test Framework This interface in the top right corner of Figure 4.1 provides the complete usage API. Users or other testing frameworks can use it to start the framework for a specific combinatorial test, and provide configuration information like the InputParameterModel and fault localization algorithm. Therefore, it fulfills requirements F1, F2, and F3. Additionally, the framework uses it to accept test results for all test cases generated in the GenerationManager. Since the test framework becomes a detail outside of our own framework, this therefore also satisfies requirement N4.

Generator To use any fault localization, first, some combinatorial test cases need to be generated (requirement F4). This is done via the Generator-component. The framework can use an arbitrary number of generators, which all need to have the same interface. What can not be seen in Figure 4.1 is that this interface needs to be in the GenerationManager-component. This is done to adhere to Clean Architecture defined by Robert C. Martin [Mar17]. Therefore, implementations should use the Dependency Inversion Principle (section 2.2.1). Instead of the framework depending on a concrete generator, it should instead depend on a generalization or interface [Fow]. As a result, both the concrete generator and the framework depend on the interface. When it is now located inside the framework, this means that the framework does not directly depend on any externally programmed component. The concrete generator becomes a detail [Mar17].

Since all generator now implement the same interface, they are interchangeable. Through the right configuration, developers can now program new generation algorithms and use them in the framework (requirement N6). This demonstrates the Linkov Substitution Principle.

Localization Algorithm One of the most important parts of the framework is the connection to a fault localization algorithm. This connection is modeled at the bottom in the middle in Figure 4.1. Through this interface, the framework passes all information needed for fault localization to an concrete algorithm (requirements F7, F8, and N7). Since the framework should not depend on an actual implementation (requirements F3 and N5), implementations should again use the Dependency Inversion Principle as with the Generator.

Event Reports Requirement F11 introduced the need for event reporting. As Figure 4.1 shows, this is done using an interface which components outside of the framework can call. Only the fault localization and generation algorithm need to call this interface. It should support simple reports in the internal engine representation format.

Reporter If a developer wants to implement a custom reporter, s/he can do so through the reporter interface. Figure 4.1 contains it at the top left corner. The framework calls the registered reporters through an interface, as implementations should use the Dependency Inversion- and Linkov Substitution Principle. The ModelBasedReporting-component converts all life cycle- and event reports and then propagates them to the registered reporters.

General Process

All requirements together define a very clear process which the framework has to follow in order to achieve working fault localization. Figure 4.2 depicts it as an EPC diagram. The gray boxes represent life cycle reports, while the blue ones represent a test result cache. This section will put the process in context of the general architecture and it's components (Figure 4.1). As a consequence, the responsibilities of all components will also become clearer. The process can be divided into three phases.

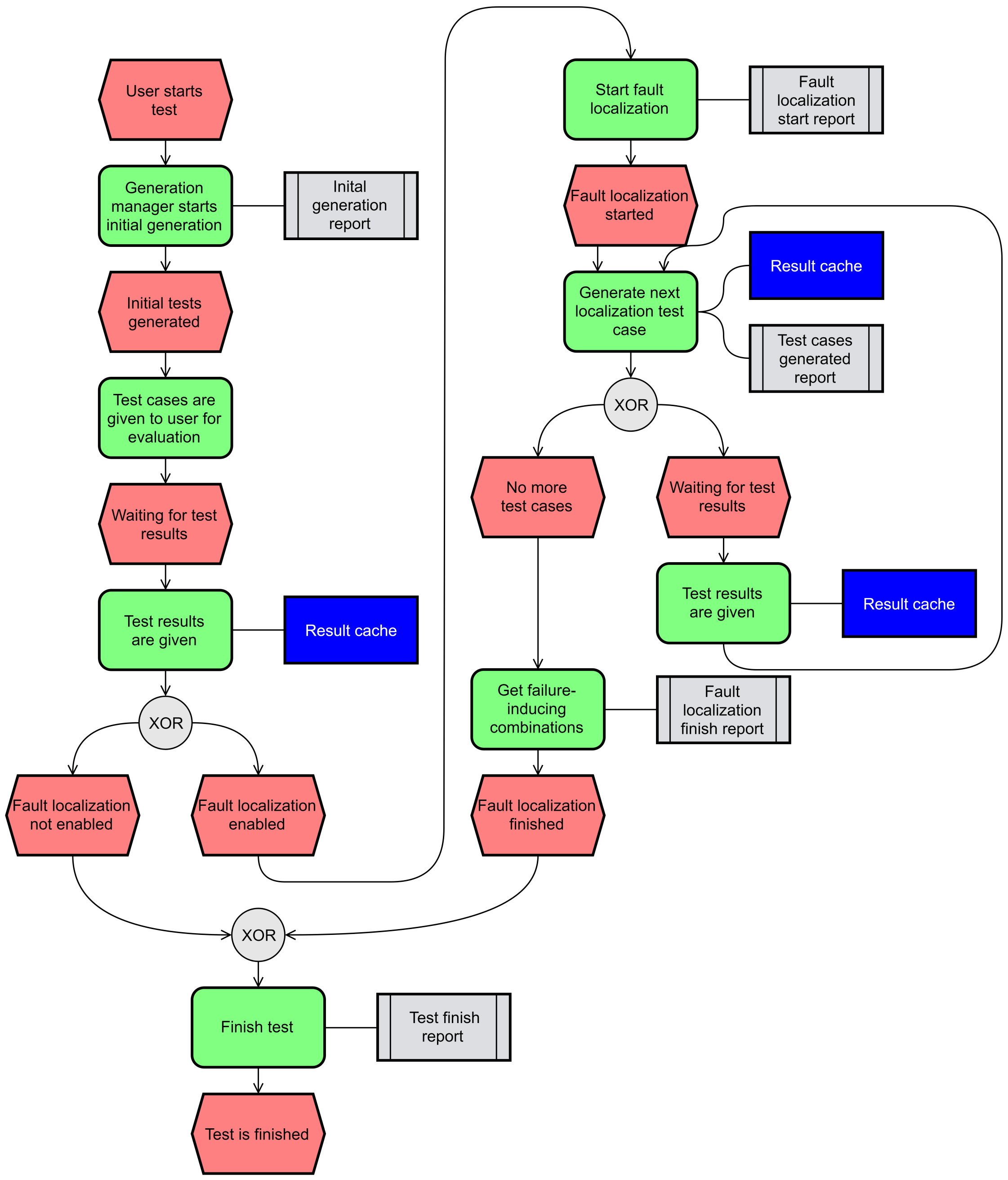
User starts test
Generation manager starts initial generation
Inital generation report
Initial tests generated
Test cases are given to user for evaluation
Waiting for test results
Test results are given
Result cache
XOR
Fault localization not enabled
Fault localization enabled
Start fault localization
Fault localization start report
Fault localization started
Generate next localization test case
Result cache
Test cases generated report
XOR
No more test cases
Waiting for test results
Test results are given
Result cache
Get failure-inducing combinations
Fault localization finish report
Fault localization finished
XOR
Finish test
Test finish report
Test is finished

Figure 4.2: EPC diagram for the framework

1. First, the framework executes traditional automated combinatorial testing. This means, that a number of generators provide test cases. For normal combinatorial testing, an IPOG algorithm can generate a t-way combinatorial test suite for the user provided IPM. On the other hand, test cases for more complex testing procedures, such as negative CT, are also possible. This phase needs all components except the LocalizationAlgorithm. First, a testing framework or any other user starts the whole process and provides an IPM. The Conversion-component then converts this IPM to an engine-processable format. After the API passes it to the GenerationManager, all configured generators provide their created test cases in the internal representation format. Since the user cannot know this format, the Conversion-component converts all test cases again, and the API passes them to the user/testing framework. Additionally, the manager publishes a report describing the initial generation through the Reporting- and ModelBasedReporting-component to any registered reporter. Finally, the Conversion-component converts all test results — provided by the user — and passes them down to the GenerationManager, which now caches them and evaluates all criteria defined in requirement F6.

2. If those criteria are fulfilled, the fault localization phase begins. Figure 4.2 shows this phase on the right side. The framework adopts an iterative approach, and in each iteration it executes the same steps. First, the GenerationManager passes all test results and the IPM to the LocalizationAlgorithm. This algorithm now decides, through some internal mechanism, whether to generate further test cases for evaluation or no test cases. In the latter case, fault localization stops and the manager reports all failure inducing combinations. Otherwise, the framework repeats the process from step 1, and the user gives all relevant test results. This is repeated until the stopping condition is met.

3. If not all conditions for fault localization are fulfilled or fault localization is finished, the final phase creates a report to let all reporters know that the framework will evaluate no more test cases. This ends the test.

4.2.2 Domain Model

Requirement F1 already defines a big part of the domain model, which Figure 4.3 shows as an UML class diagram. Those classes each describe the external representation which the user, or other testing frameworks, can access. The internal representation cannot be derived from any requirement, and developers wishing to implement an automated fault localization framework should choose one on their own.

The central entity of the domain model is the `InputParameterModel` as section 2.1.5 described. It contains all information necessary to start the generation of a combinatorial test. This includes a name or other identification by which the user can recognize different IPMs, the strength with which generators should generate test cases, parameters describing the SUT, and constraints on the values of these parameters. An IPM needs to contain at least one parameter, otherwise test case generation does not make any sense.

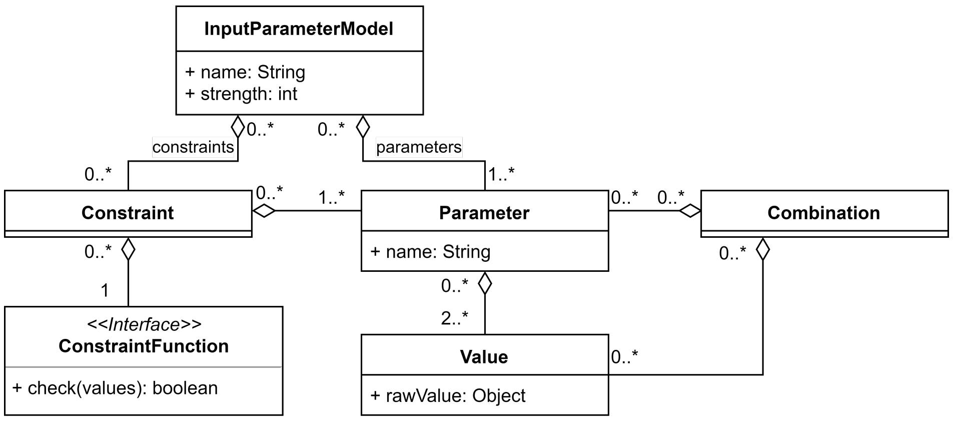

Figure 4.3: Class diagram of the domain model

Constraints, however, are not required. While it may be rare in practice, some systems could also be totally unconstrained on their input space.

A constraint is modeled in such a way that it takes values for a number of defined parameters and then evaluates whether the given value-combination satisfies the constraint. This general interface means that developers can employ any type of constraint representation literature proposes. A function returning false when a test case contains one specific sub-combination, and true otherwise, represents an invalid combination constraint. At the same time, the `ConstraintFunction` can hide more complex functions. This one can easily represent logical expressions and evaluate them lazily just when the framework needs them for constraint solving. The concrete interface of a constraint function can be different in the actual implementation.

Parameters, like the IPM, also have a name to help identification by the user. This is especially important during reporting, as the user may not always know which parameter is meant when presented with some value, and often, multiple parameters have the same values (for example boolean parameters). Thus the user needs a unique identification mechanism. Consequently no two parameters can have the same name in one IPM. The values themselves do not need to store any other information than the actual object (as of requirement N1). This could result in the misconception that the framework does not need a `Value` class, but especially in programming languages which allow **`null`**, encapsulation provides a differentiation between a missing values, and an explicit **`null`** value. Additionally, there are some combinatorial testing features which assign attributes to values. Future developers could easily add them to the `Value` class. For example, the framework could give users the option to define value weights to control how often values appear in test cases.

A test case itself is only a combination of values where each parameter has one assigned value. This means, that the framework does not need to differentiate between those two types. As a consequence, the domain model only contains the `Combination` type, but an actual implementation of the framework could always add an additional class for representing “complete” combinations (test cases).

4.2.3 Reporting

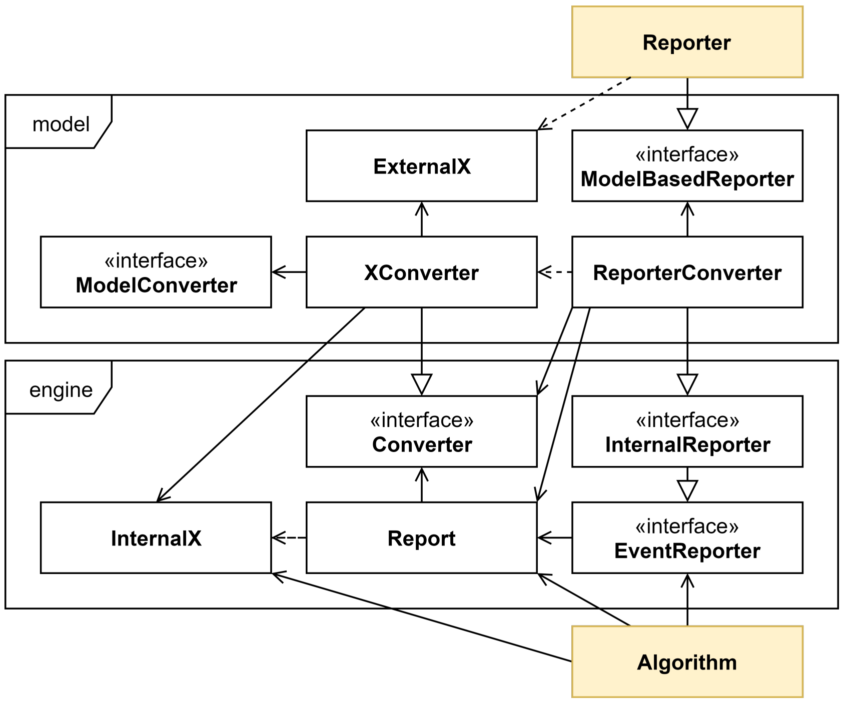

Figure 4.4: Component diagram of report conversion

One very interesting area of the architecture is reporting. Since the framework needs a split between engine and model as described in section 4.2.1, reporters have to convert between internal and external representation types. This introduces an added complexity into an otherwise straightforward part of the application.

This conversion for life cycle reporting, as requirement F10 defined, is not difficult. A component handling the management of all registered external reporters just uses the `Conversion`-component to create a representation of all propagated information which the external reporter can understand/parse. Where this gets much more difficult is in event reporting (requirement F11). This type of reporting passes some notable events, like debug information, or any other messages which could be interesting for users, but is not considered in life cycle reporting. In particular, life cycle reporting can not pass on any information which is algorithm specific and thus the framework needs to establish other means of communication.

Some of this information only consists of text. In this case the algorithm just needs to pass a simple string like `"debug information"` to the user. This is not complex, as a the framework can easily provide such an API. Other information like `"(OS=Windows, ping=10)is a failure-inducing combination with 45% probability"` is more difficult to pass in a readable way. Since the algorithm only knows the internal representation, one can not be sure, that it knows how to translate it into a readable format.

And even if it did know this, what if a reporter needs the report's actual objects for some operation? All of this leads to the conclusion that the framework needs some mechanism for converting the internal representation to an external representation. This reporting mechanism should allow reports to pass all actual objects from the algorithm, but still employ conversion to external representation types.

Figure 4.4 shows the architecture's solution. Since there are many domain objects which the framework needs to convert (see section 4.2.2), this figure contains only the general case for some entity called `X`. It has an internal and external representation. The `Algorithm` now wants to pass a `Report`, which contains an `InternalX` to the user provided `Reporter`. How the report is structured internally is not important for this to work, it only has to provide some support for passing objects and converting them.

Now, the general process is that the algorithm provides a report which contains some `InteralX` to its `EventReporter`. The implementation behind this interface is actually the `ReporterConverter`. This converter now receives a `Report`, and begins the conversion process. To this end, it has several registered `Converter` implementations, one for each entity which needs conversion. These converters can use any part of the `ModelConverter` to change an internal to its external representation. For each object passed along inside a `Report`, the `ReporterConverter` checks all converters for whether they can convert the type. If that is the case for one of them, in replaces the object. After it replaced all objects, it passes the `Report` on to the user provided `Reporter`, now in an converted format.

This solution has several advantages. Since users can dynamically register converters, algorithms can directly use custom internal representations in their reports and just need to provide a `Converter` which creates an external representation. Additionally, the architecture is also robust against changes in the domain model. Developers only need to adjust the converters of changed entities; all others can be used as normal (Single Responsibility Principal).

Furthermore, this solution preserves Clean Architecture principles. All arrows in the simplified class diagram of Figure 4.4 point inwards, to classes within engine. As a result, the `Reporter` and `Algorithm` become details, and programmers can quickly change them without effecting any class in model and engine. Additionally, developers can change external representations without adjusting any part of the engine. Even if `Report` needs to know about `Converter`, it does not need to know concrete implementations.

All of this is again achieved through the Dependency Inversion Principle. At any point where arrows for the process would normally point away from a class in engine or model, the architecture introduces an interface, effectively hiding the real implementation (see `ModelBasedReporter`, `EventReporter`, `Converter`, and `ModelConverter`).

5 Realization

Contents

Now that the last chapters explained the general concept and architecture of a fault localization framework, this chapter focuses on *CombTest*, the actual Java implementation of said framework. One can consider this to be a proof of concept. First, all general information about how the architecture was transformed into code is presented in section 5.1. Next, section 5.2 describes important extension points and how users can write their own fault localization or generators. Finally, section 5.3 demonstrates how testers can use CombTest's API to write their own combinatorial tests.

5.1 Overview

To understand how CombTest realizes all extension points and how developers can use them, this chapter first explains CombTest's general structure. Section 5.1.1 will describe the general structure of the code and how Maven is used to manage it. Earlier in the thesis, the idea of a split between an engine and model component was introduced. Section 5.1.2 explains its realization and challenges faced during development. Lastly, Section 5.1.3 describes the general concept of a *test case group*.

5.1.1 Maven Structure

The CombTest-framework is structured as a Maven multi-module project. This basically means that there is one parent project containing multiple child modules, each only dealing with one part of the application. The parent project defines information important

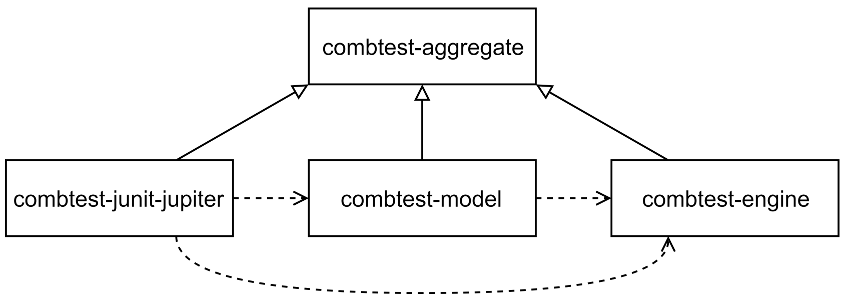

Figure 5.1: Component diagram of the Maven structure

to all child modules. For example, In CombTest's case this means it manages all versions for dependencies like JUnit and fastutil.

Figure 5.1 shows the general structure of CombTest. *combtest-aggregate* is the parent-project from which all modules inherit common information. In total, there are three chile modules. *combtest-engine* and *combtest-model* are direct mappings from the big architecture parts model and engine. The JUnit extension resides in a third component. As a result, developers only need *combtest-junit-jupiter* if they want to program combinatorial JUnit tests. The advantage of different modules is a stricter enforcement of separation of concerns. When a developer wants to add a part to the framework, s/he has to ask him/herself in which module it fits best, or if a change requires adjustment of multiple modules.

Dotted arrows in the figure all show direct dependencies through the maven dependency management functionality. At all points where classes in engine would normally depend on classes in model, CombTest uses the Dependency Inversion Principle to flip the dependency around. As a result, there is a clear hierarchical dependency structure, conforming to the standards of Clean Architecture [Mar17]. Since Maven does not allow cyclic dependencies between modules, it enforces correct implementation of said standards.

5.1.2 Engine and Model Split

Section 4.2.1 raised the need for a split of representation types between the framework's engine and model parts. For a brief recapitulation: The requirements state that users can enter any native object as a value, but the performance should not depend on object sizes. In Java this means the performance of `equals` and `hashCode` should not impact the framework's performance. Consequently, an internal representation type is needed for all modeling aspects.

In CombTest, all internal representation types revolve around primitive types like **`int`** and **`double`**. A values is therefore simply denoted by its index in the corresponding parameter. For example, in the OS parameter from Table 2.1, "0" represents Windows and "3" is Android. To pass a parameter to another method, it is therefore sufficient just to pass one integer showing the parameter's size. Now the method has all information needed to perform combinatorial calculations. When not considering constraints, the IPM is only the testing strength and definition of all parameters. As a result, CombTest's engine module can represent it as an `int` for the strength, and an `int[]` for parameter sizes. The convention is that the parameters are just indexed from zero to $n - 1$. `int[] parameters = 5, 4, 3, 4` therefore represents Table 2.1. Test cases are just the assignment of values to parameters in an `int[]`. For example, `int[] testCase = [-1, 2, 0, -1]` is equivalent to (—, Firefox, 10 ms, —). The number minus one denotes that the respective parameters is not set to any value.

Having primitive internal representation types has many advantages. The implementation does not depend on actual value object sizes, and creation of values, and especially comparison, is computationally very cheap. In combinatorial test generation one of the main bottlenecks for performance seems to be the speed with which values are created and how fast we can compare test cases or check if a values has already been set. Using primitive types solves both problems. In an earlier version of CombTest, engine directly used the types given in the model. Even when using just boxed integers, the performance was noticeable better if CombTest converted the model. Of course, this manual checking was not a representative study, but with growing object sizes and complexity there has to be some point where the fixed cost of conversion will be outgrown by multiple calls to slow `equals` implementations during computationally expensive test case generations.

Of course, there are also some disadvantages. Normal objects are nearly always better when writing code, or trying to understand it. Since they have descriptive names and one can write functions in classes, code just looks much more fluent. With primitive types, the writer and reader have to know all conventions as to not make any mistakes. Additionally, some design decisions of the Java programming language itself lead to difficulty. Since generics only allow complex types, developers have to use auto-boxing for primitives. Therefore CombTest uses *fastutil*, a library consisting of primitive type implementations for the Java Collections API [Vig14]. Another problem is the implementation of arrays in Java. Currently they do not support any sensible `equals` and `hashCode` implementation. Instead, they employ the default implementation which uses the object's storage location. Therefore, comparison between two exact copies is not possible. As a result, it is very difficult to use arrays in collections. To mitigate this problem, CombTest uses a class called `IntArrayWrapper`, which is just a wrapper class and uses `Arrays.equals(int[] first, int[] second)` and `Arrays.hashCode(int[] array)` for a correct comparison.

Some of these problems will be avoidable in the future. Since Java is currently working to support primitive types in collections and the creation of data classes, there is hope that one day CombTest will not have to use primitive types at all, and still have a high performance [Goe14].

5.1.3 Test Case Groups

One specific focus of the implementation (not the architecture in general!) was the support for negative testing as [FL18] describes it. To recapitulate: This type of negative testing works by negating error constraints one by one and generating a new suite of test

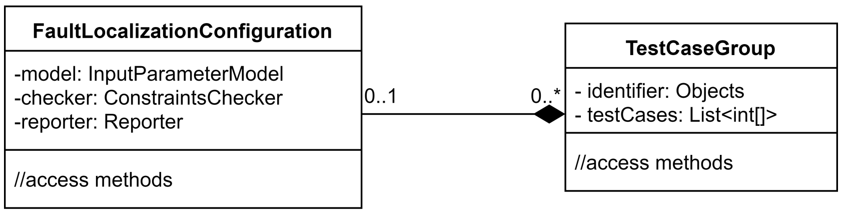

Figure 5.2: Class diagram for `TestCaseGroup`

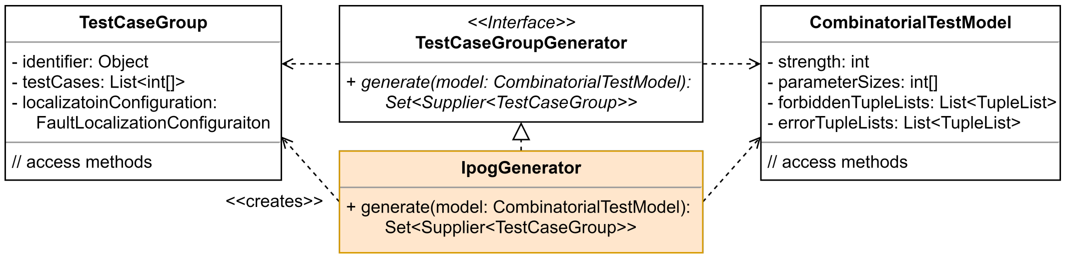
TestCaseGroup
- identifier: Object
- testCases: List<int[]>
- localizatoinConfiguration: FaultLocalizationConfiguraiton
// access methods
<<Interface>>
TestCaseGroupGenerator
+ generate(model: CombinatorialTestModel): Set<Supplier<TestCaseGroup>>
IpogGenerator
+ generate(model: CombinatorialTestModel): Set<Supplier<TestCaseGroup>>
<<creates>>
CombinatorialTestModel
- strength: int
- parameterSizes: int[]
- forbiddenTupleLists: List<TupleList>
- errorTupleLists: List<TupleList>
// access methods

Figure 5.3: Class diagram for `TestCaseGroupGenerator`

cases for each negated constraint. For constraint aware fault localization, this leads to the problem of which constraints to use. Additionally, masking effects can occur if fault localization does not differentiate between positive and negative testing.

All of these arguments lead to the design decision of a layer between combinatorial tests and individual test cases. The main component of this layer is called a `TestCaseGroup`. One combinatorial test can have multiple `TestCaseGroups`, and each group consists of an arbitrary number of test cases. Fault Localization is done per group. This means CombTest fulfills requirements F6, F7, F8, and F10 per group, and not for all test cases together. Theoretically, one could emulate a framework without `TestCaseGroups` by just adding a collective generator which puts all groups from multiple generators into one test case group.

Having `TestCaseGroups` leads to the problem that a group specific fault localization algorithm needs to know which constraints to consider and which parameters to use. Therefore, CombTest introduces a `FaultLocalizationConfiguration`. It is part of a `TestCaseGroup` and specifies all important information for fault localization algorithms. Consequently, a it includes a constraints checker, IPM, and a reporter (see Figure 5.2).

5.2 Extension Points

After last section explained all basics about how the fault localization framework is realized, this section will go more into detail about four important extension points. For all other extension points, please consult the documentation of CombTest.

5.2.1 TestCaseGroupGenerator

`TestCaseGroupGenerator` is the realization of the generator in form of an interface. As Figure 5.3 shows, it only needs one method. The sole responsibility of a generator is to provide an arbitrary number of `TestCaseGroup`s. The possibility to create multiple `TestCaseGroup`s per generator also rooted from the desire to support negative testing. Since the number of groups depends on how many error constraints the user specified, it would be very difficult to give exactly one generator per constraint. Basic combinatorial testing algorithms like IPOG can still use the interface and just return one `TestCaseGroup`.

The use of Java `Supplier`s is an important design decision, too. Generators now do not need to invoke group generation themselves, but rather pass this responsibility to the framework. Consequently, optimizations are possible in CombTest. For example, it would be relatively easy to introduce parallel creation of `TestCaseGroup`s. Therefore, generator developers do not have to deal with parallelization themselves and can focus on real generation aspects.

Generators cannot generate groups without knowing some basic information about the combinatorial test. Therefore, CombTest provides the internal representation of an IPM, called `CombinatorialTestModel`. It includes all information necessary to generate groups for test cases.

As a proof of concept, CombTest provides multiple sample generators. The most important one is certainly a full implementation of the IPOG algorithm. It generates just one `TestCaseGroup` which contains test cases covering each possible value-combination of size t. The algorithm was implemented according to a several performance improvements and implementation advice by Kleine et al. [Sim18]. CombTest makes some modifications to the original IPOG to allow constraint handling in IPOG. Additionally, Konrad Fögen provided an implementation of his negative testing algorithms [FL18].

5.2.2 FaultLocalizationAlgorithm

As Figure 5.4 shows in the top left corner, CombTest's interface for fault localization is very near to the requirement's definition. To allow for an algorithm to decide whether it needs further test results for finding failure-inducing combinations, `computeNextTestCases` takes all requested test results and returns a list of new test cases (requirement F7). When an algorithm requires no further test results, it can calculate all failure inducing combinations in `computeFailureInducingCombinations`. Most algorithms like AIFL and Improved Delta-Debugging already accomplish this step during the generation of further test cases, so for them it is only an access method for an internal list of failure-inducing combinations. Probability-based algorithms such as BEN can benefit from the return type of `computeFailureInducingCombinations`. Since they have in internal ranking of which combinations are most likely to be failure-inducing, they can convey such information to the user via an ordered list.

During the implementation of `Ben`, `Aifl` and `IterAifl` a shared general structure became apparent. Each of these algorithms keeps an internal list of suspicious com-

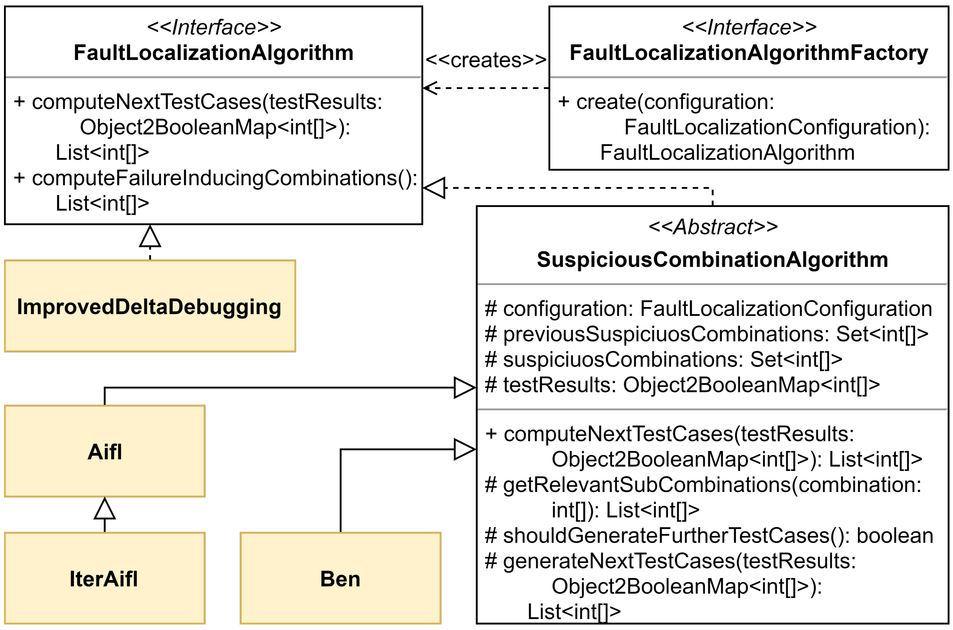
<<Interface>>
FaultLocalizationAlgorithm
+ computeNextTestCases(testResults: Object2BooleanMap<int[]>): List<int[]>
+ computeFailureInducingCombinations(): List<int[]>
<<creates>>
<<Interface>>
FaultLocalizationAlgorithmFactory
+ create(configuration: FaultLocalizationConfiguration): FaultLocalizationAlgorithm
<<Abstract>>
SuspiciousCombinationAlgorithm
# configuration: FaultLocalizationConfiguration
# previousSuspiciuosCombinations: Set<int[]>
# suspiciuosCombinations: Set<int[]>
# testResults: Object2BooleanMap<int[]>
+ computeNextTestCases(testResults: Object2BooleanMap<int[]>): List<int[]>
# getRelevantSubCombinations(combination: int[]): List<int[]>
# shouldGenerateFurtherTestCases(): boolean
# generateNextTestCases(testResults: Object2BooleanMap<int[]>): List<int[]>
ImprovedDeltaDebugging
Aifl
IterAifl
Ben

Figure 5.4: Class diagram for `FaultLocalizationAlgorithm`

binations and computes further test cases only if a specific criterion is fulfilled. They build a list of suspicious combinations by only using some sub-combinations in each failed test case and removing those sub-combinations appearing in successful test-cases. `SuspiciuosCombinationAlgorithm` handles the management of suspicious combinations according to some details a concrete algorithm provides through abstract methods. Only `ImprovedDeltaDebugging` did not share this structure as it checks failed test cases one at a time to find failure-inducing combinations.

Since section 5.1.3 defined that CombTest needs to execute fault localization for each `TestCaseGroup` individually per combinatorial test, it needs multiple instances of one `FaultLocalizationAlgorithm`. This is why the user has to specify the algorithm with a `FaultLocalizationAlgorithmFactory`, which provides a new algorithm per group.

5.2.3 Reporter

CombTest brings requirements F10 and F11 together into one single interface as shown in Figure 5.5. Each life cycle event has a corresponding method which conveys needed information. To distinguish between groups, CombTest passes a `TestCaseGroupContext` to all methods which it executes per group. Events about the execution of one single test case do not contain a context, as CombTest employs caching and may therefore use a test execution result for other groups.

Additionally, two methods deal with event reporting. A reporter may provide a severity level with `getReportLevel()`. As a result, CombTest only passes reports with higher, or equal severity to the `report` method. `report` itself uses the report with resolved

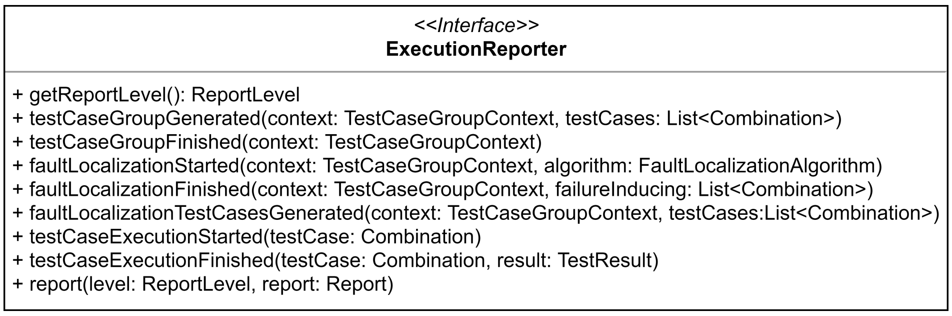
<<Interface>>
ExecutionReporter
+ getReportLevel(): ReportLevel
+ testCaseGroupGenerated(context: TestCaseGroupContext, testCases: List<Combination>)
+ testCaseGroupFinished(context: TestCaseGroupContext)
+ faultLocalizationStarted(context: TestCaseGroupContext, algorithm: FaultLocalizationAlgorithm)
+ faultLocalizationFinished(context: TestCaseGroupContext, failureInducing: List<Combination>)
+ faultLocalizationTestCasesGenerated(context: TestCaseGroupContext, testCases:List<Combination>)
+ testCaseExecutionStarted(testCase: Combination)
+ testCaseExecutionFinished(testCase: Combination, result: TestResult)
+ report(level: ReportLevel, report: Report)

Figure 5.5: Class diagram for `ExecutionReporter`

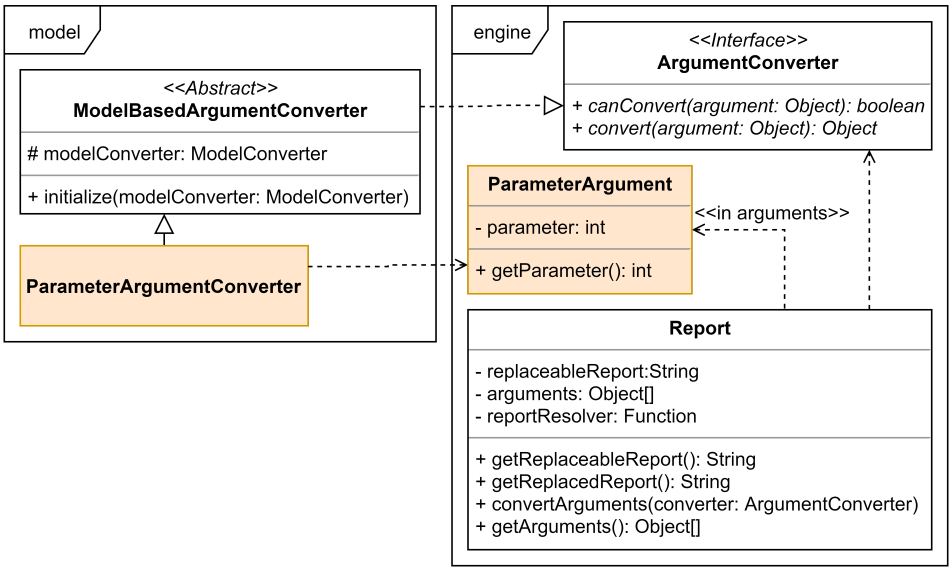
model
<<Abstract>>
ModelBasedArgumentConverter
# modelConverter: ModelConverter
+ initialize(modelConverter: ModelConverter)
ParameterArgumentConverter
engine
<<Interface>>
ArgumentConverter
+ canConvert(argument: Object): boolean
+ convert(argument: Object): Object
ParameterArgument
- parameter: int
+ getParameter(): int
<<in arguments>>
Report
- replaceableReport:String
- arguments: Object[]
- reportResolver: Function
+ getReplaceableReport(): String
+ getReplacedReport(): String
+ convertArguments(converter: ArgumentConverter)
+ getArguments(): Object[]

Figure 5.6: Class diagram for `ArgumentConverter`

arguments (section 4.2.3).

`PrintStreamExecutionReporter` is a sample implementation printing out all events to console in a simple format.

5.2.4 Argument Converter

Section 4.2.3 introduced the idea of argument converters to transform internal to external representations. CombTest not only uses this mechanism for reporting, but also for identifiers in `TestCaseGroups`. The idea is that for negative testing, the negated constraint should be the identifier of its corresponding group. Since `TestCaseGroupContexts` use identifiers for reporting, one would need a conversion to an external constraint representation. Since argument converters are already implemented for reporting, the framework can just reuse them.

The general interface of an `ArgumentConverter` is not very complex. There are just two methods, one specifying whether it can convert a given object, and another one for actually converting said object. The interface's contract specifies that CombTest only calls `convert(Object argument)` if `canConvert(Object argument)` returns **`true`**. As a result, the former method does not have to deal with checking the argument's type, and developers can avoid much duplicate code.

To successfully convert internal representations, each `ArgumentConverter` needs to know about the conversion method used. Therefore, it needs a `ModelConverter`, which can convert parameters, values, constraints, and complete IPMs. Since most converters need this, CombTest provides `ModelBasedArgumentConverter`, an abstract class which takes care of initializing and handling the `ModelConverter`. CombTest provides an implementation of `ModelBasedArgumentConverter` for values, parameters, combinations, and constraints.

5.3 Usage

More important than how to write extensions for CombTest is how developers can use it to write combinatorial tests. There are two ways to do that. Option one is to directly use the framework's API for configuration and execution. This method has some slight disadvantages. CombTest is no full testing framework in the sense that it does not incorporate a test discovery mechanism. Consequently, developers would have to write a `Main.java` class which configures all test CombTest should execute. The second option is using the framework's JUnit5 extension. It fully incorporates CombTest into the popular testing framework, and makes all familiar features of parameterized tests available. This is mostly due to the fact that the extension was just written as a proof of concept, and therefore copies large code and design portions of `junit-jupiter-params`. The next two sections present usage examples of both APIs. First, the lower-level CombTest API is explained in section 5.3.1. Next, section 5.3.2 shows how to natively integrate combinatorial tests into JUnit5.

5.3.1 Framework

Using CombTest's API for test execution requires three arguments: A configuration, a test method, and an IPM. Since all of these arguments contain some optional settings, the builder pattern is used extensively [Gam+95]. The following lines show an example test, which first configures all required parameters, and then executes it with a function containing a build-in flaw at a specific sub-combination.

```java
public class FrameworkUsage {
  public static void main(String[] args) {
    new CombinatorialTestExecutionManager(
        consumerManagerConfiguration()
            .executionReporter(new PrintStreamExecutionReporter())
            .generator(new IpogTestCaseGroupGenerator())
            .localizationAlgorithmFactory(ben())
            .build(),
        FrameworkUsage::testFunction,
        inputParameterModel("exampleTest")
            .strength(2)
            .parameters(
                parameter("param1").values(0, 1, 2),
                parameter("param2").values("0", "1", "2"),
                parameter("param3").values(0.0f, 1.1f, 2.2f),
                parameter("param4").values(true, false))
            .forbiddenConstraint(constrain("param1", "param3")
                .by((Integer firstValue, Float thirdValue) ->
                    !(firstValue == 0 && thirdValue == 1.1f)))
            .build())
        .execute();
  }

  private static void testFunction(Combination testCase) {
    final int firstValue =
        (Integer) testCase.getValue("param1").get();
    final String secondValue =
        (String) testCase.getValue("param2").get();
    assertFalse(firstValue == 1 && "1".equals(secondValue));
  }
}
```

Source Code 5.1: FrameworkUsage.java

The example will first run a normal IPOG algorithm to generate an initial set of test cases for the given input parameter model `"exampleTest"`. None of these test cases will contain (0, —, 1.1f, —). Next, the execution manager runs all these test cases by passing the respective combination to `testFunction`, which throws exceptions on (0, "1", —, —). Since the test configures a fault localization algorithm factory, CombTest passes all test result to the BEN algorithm, and further tests narrow in on the failure-inducing combination. The `PrintStreamExecutionReporter` will print out all steps to the console.

`FrameworkUsage.java` does not show all possible configuration options. It is generally possible to directly specify a low level executor. The default one uses caching and the process described in sections 4.2.1 and 5.1.3. Additionally, users can customize the mapping between internal and external representation types by providing a custom `ModelConverterFactory`. Finally, they can configure an arbitrary number of `TestCaseGroupGenerator`, `ExecutionReporter`, and `ArgumentConverter`.

5.3.2 JUnit5 Extension

CombTest's JUnit5 extension has some more sensible default values. As a results, the barriers of entry are slightly lower, and a minimal example works with just configuring an IPM. Most configurations in `combtest-junit-jupiter` are done through annotations like in `junit-jupiter-params`. For example, configuring a generator means adding the `@Generator(IpogTestCaseGroupGenerator.class)` annotation to a test. The example from section 5.3.1 now looks like this:

```java
class JunitUsage {
  @CombinatorialTest
  @LocalizationAlgorithm(Ben.class)
  @Reporter(PrintStreamExecutionReporter.class)
  @ModelFromMethod("model")
  void combinatorialTest(int param1, String param2, float param3,
      boolean param4) {
    assertFalse(param1 == 1 && "1".equals(param2));
  }

  private static InputParameterModel.Builder model() {
    return inputParameterModel("exampleTest")
        .strength(2)
        .parameters(
            parameter("param1").values(0, 1, 2),
            parameter("param2").values("0", "1", "2"),
            parameter("param3").values(0.0f, 1.1f, 2.2f),
            parameter("param4").values(true, false))
        .forbiddenConstraint(constrain("param1", "param3")
            .by((Integer firstValue, Float thirdValue) ->
                !(firstValue == 0 && thirdValue == 1.1f)));
  }
}
```

Source Code 5.2: JunitUsage.java

As we can see, this example has nine fewer lines of code. Additionally it is easy to execute, as most modern Java IDEs will pick up JUnit tests and allow direct execution through a dedicated GUI. One advantage of the extension is the use of JUnit's `ParameterResolver`. Now, the user does not have to extract all values from a `Combination` object, but can instead expect them as parameters for a test method. They will also work with the `ArgumentsAggregator` and `ArgumentConverter` features of `junit-jupiter-params`.

6 Evaluation

Contents

Having an implementation of the general architecture is a very important step. What is still missing though, is an evaluation of CombTest. It would be of no use if CombTest did not satisfy all requirements presented in section 4.1, or if the source code was so untested, that nobody would want to use it. This chapter deals with such questions.

6.1 Requirements

Perhaps the most important question which one can ask about any software is: Does it fulfill all requirements? If a product is very nice to use and has a good code structure, the development process still failed if it is completely different from the original request. Therefore, there is another module in CombTest called *combtest-evaluation* which has a package called `requirements`. This package contains so called acceptance tests for each requirement, to verify all functionality works as intended [HH08]. Additionally, these tests represent a way of learning how to use CombTest's API.

JUnit tests cannot easily cover all non-functional requirements. Therefore, the next few paragraphs verify these arguments with a more argumentative approach.

N2 Performance-Independence from Object Sizes Requirement N1 specified that test developers can use native objects as values in IPMs. To shield users from having to develop faster `equals` and `hashCode` versions, requirement N2 demanded performance-independence from these methods. To test whether CombTest achieves this, a small experiment has been setup.

In class `RequirementN2Test` there are two separate test cases. One of them contains a model consisting only of **`int`** values, the other instead has instances of `SlowClass`. This class was specifically programmed for the performance test, and has deliberately slow implementations of `equals` and `hashCode` due to explicit waits. When each test method now measures the time for its execution, the one using `SlowClass` would be much slower if the framework ever uses any of its methods. To guarantee a fair comparison, both IPMs

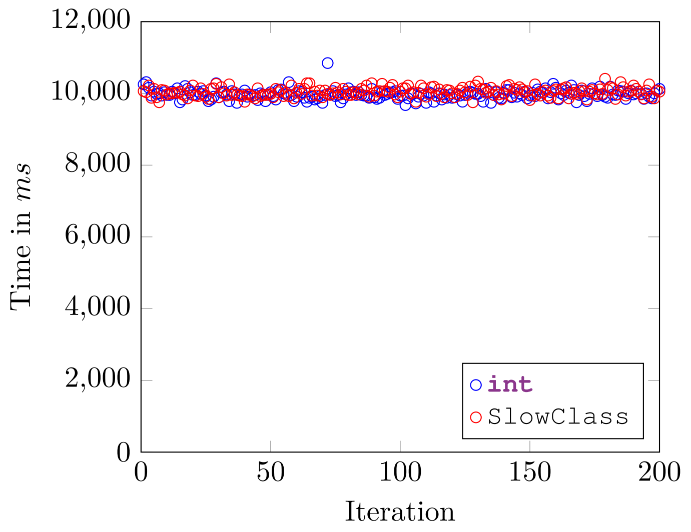

Figure 6.1: Comparison of CombTest execution times between Java primitive types (**`int`**) and a custom slow type over 200 iterations. For **`int`**, median, average, and standard deviation were 9970 *ms*, 9974.48 *ms*, and 131.493507 *ms* (in that order). For `SlowClass` they were 10033 *ms*, 10037.065 *ms*, and 121.669504 *ms*.

are equal except in the type of their values. Both have two constraints targeting the exact same combinations. Additionally, they test the parameters with the same strength.

The general setup is an initial generation of a test suite covering all 3-value-combinations using IPOG. An introduced fault in the tests' execution methods forces CombTest to use BEN for fault localization. This guarantees that the tests use all parts of the framework.

If the execution of a program on any computer were completely deterministic, this would be enough to check execution times. However, concurrently running background processes and garbage collection by the Java environment destroy this illusion. Every run, even on the exact same computer, will always be slightly different. To mitigate these factors to the tests' validity, each combinatorial test is executed multiple times (in our case 200). Calculating an median value will then show any differences in execution times. Figure 6.1 shows the duration of every iteration and the text below it states median, average, and standard deviation. Since the `SlowClass` variant is not more than five seconds slower, we can safely assume that `equals` and `hashcode` were never called; CombTest fulfills requirement N2.

All tests were executed on a Windows 10 computer running with JDK 1.8.0 update 181 using IntelliJ IDEA ULTIMATE 2018.2.2. These programs ran on an AMD FX-6100 running at 3.30 GHz supported by 16 GB of DDR3 RAM.

N4 Test Framework Agnostic There is no test case which can show CombTest's test framework agnosticism, so instead an argumentative approach is needed. Since there are no JUnit imports in engine or model, there is at least no direct dependency. Nevertheless, CombTest could indirectly depend on a test framework; for example, if its architecture was build in such a way that only one specific testing framework could actually use it. However, while JUnit5 was the intended target framework since the beginning, CombTest construction happened before the JUnit extension and as such JUnit5 did not influence any architectural decision.

N5-N7 Developing Custom Implementations of Algorithms and Reporters Requirements N5 through N7 deal with the development of fault localization algorithms, generators, and reporters by third parties. Section 5.2 presented multiple example implementations. As such it is clear that other developers can add to CombTest through these extension points. Additionally, section 4.2.3 introduced `ArgumentConverters`. While the requirements did not include them, they are also extendable through custom implementations.

Requirement N5 specifically stated that CombTest should support all current fault localization algorithms. Proving this requirement by actually implementing all algorithms would take too much time, so instead CombTest contains four representative algorithms (AIFL, IterAIFL, Improved Delta Debugging and BEN). Each of them employs a slightly different method of finding failure-inducing combinations, but all of them fit inside the interface defined by section 5.2.2. Additionally, while non-adaptive fault localizations were not a focus of this thesis, developers could also implement such algorithms by writing a different generator working in conjunction with a specialized fault localization algorithm.

Since requirement N7 was the last one, we can therefore say that CombTest fulfills all requirements according to their specification.

6.2 Quality

There are many ways to measure a software's overall quality. Miguel et al. mention over 18 software quality models between 1977 and 2013 alone [MMR14]. Section 6.2.1 will analyze CombTest according to the famous ISO/IEC 25010:2011 standard [Sta11]. During development itself, programmers often do not continuously evaluate their code with such standards, but instead use static code analysis tools like SonarQube. Therefore, section 6.2.2 discusses SonarQube's evaluation of CombTest.

6.2.1 ISO/IEC 25010

Like most software quality models, ISO/IEC 25010:2011 defines several characteristics according to which one should evaluate a software product [MMR14]. In this case there are eight characteristics as Figure 6.2 depicts. Each characteristic defines a number of sub-characteristics whose assessment accumulates to an evaluation of the respective top-level characteristic. For example, *Reliability* includes *Modularity*, *Reusability*, and several other sub-characteristics. This section will directly evaluate CombTest according

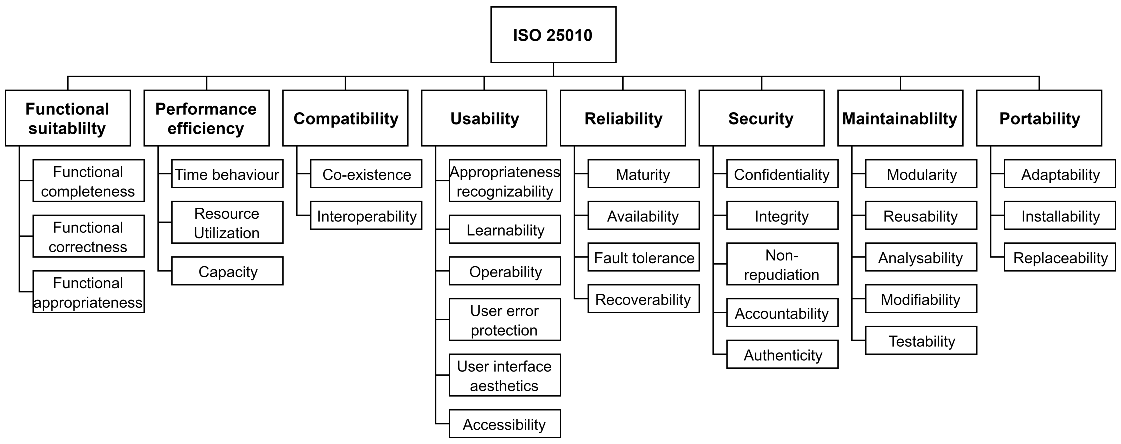

Figure 6.2: ISO/IEC 25010:2011 characteristics

to all top-level characteristics, as evaluating sub-characteristics would simply not fit into the limit of this thesis.

Functional Suitability Section 6.1 already showed that all CombTest fulfills all requirementsd. Additionally, it also fulfills implicitly stated needs, such as the support for negative combinatorial testing. In section 6.3 one will also see that automated fault localization works in general.

On the other side however, there are also some features which fault localization could need in the future which are not possible with the current version of CombTest. One could image, that an even more adaptive form of fault localization is possible, where on algorithm covers all t-value-combinations and localizes faults at the same time. It could then directly construct test cases based on previous results. Such a process is currently not possible with the framework, as only the `FaultLocalizationAlgorithm` interface offers dynamic test generation functionality. At the same time, CombTest can only call classes implementing this interface when there was at least one fault in an initial test suite. Consequently, it is not possible to generate additional dynamic test cases after one successful initial test. However, since CombTest's core generation manager is very adaptable and hidden behind an interface, it would be possible to simply switch it out for one which could handle the described scenarios.

Performance Efficiency Since generating even a near-minimal combinatorial test suite is a very hard and time-consuming problem, CombTest should not introduce an additional significant performance decline. This inefficiency in test suite generation was one of the main arguments for introducing a split between model and engine. Converting every generated test case one at a time to present the external format to the user is of linear complexity for a constant number n of parameters. However, test suite generation is never linear, so it makes sense to speed up this important part of the framework through the use of only primitive types. What all of this means is, that it would not have been hard to make CombTest a little bit faster by not implementing a model engine split, but this performance hit was deliberately taken to increase generator execution speed.

To evaluate CombTest's performance, a small experiment was conducted similarly to the one in paragraph 6.1. However, instead of measuring the complete execution time, only the time inside CombTest counts. Therefore,the experiment uses wrapper classes around generation and localization algorithms which stopp the time measuring process. The input parameter model consists of eight parameters (eight values each), no constraints, and a testing strength of three. The test executor is configured to throw exceptions at exactly one specific three-value-combination, thus simulating a failure-inducing combination. All in all, IPOG generates 1050 test cases, with BEN using an additional 20 test cases to fully locate the fault.

`de.rwth.swc.combtest.experiements.PerformanceTest` contains all performance tests. When executing, it prints out the execution time to console in nanoseconds. 200 measurements result in a median of 10.377 *ms*, average of 10.498 *ms*, and standard deviation of 0.641 *ms*. Since normal execution time for the IPOG algorithm is measured in seconds, not milliseconds, these times are plenty small enough for a framework like CombTest.

Compatibility Since CombTest uses a Clean Architecture, and therefore does not directly depend on any testing framework, developers could write extensions for every framework they want to use it in. The only requirement is that test cases can be dynamically generated based on previous results. For example, CombTest is compatible with JUnit4 as one could write a `Runner` which executes CombTest.

There is one small incompatibility which needs to be taken into consideration. In version 5.3 JUnit5 introduced the concept of parallel test execution. Due to the way concrete code, fault localization with CombTest's JUnit5 extension is not compatible with parallel execution. However, later versions of the JUnit5 extension could fix this.

Usability Evaluating the usability for a given software product nearly always involves users testing features or just using the product in production. Since this is just not possible due to time constraints placed on a bachelor thesis, there was no way to conduct any empirical study. Consequently any usability evaluation is a completely subjective, and most certainly influenced by the fact that I wrote CombTest.

Since CombTest uses the popular Maven framework, there is an easy and established way to use it in production. As Maven has a very high market share, and gradle uses the m2 dependency model, too, nearly every Java developer will have a way to install CombTest. Additionally, the JUnit5 extension will help for an easy adoption, since it is very near to parameterized test. While JUnit5 is relatively new, and some developers may not have switched from JUnit4, writing basic tests is relatively similar to JUnit4 tests, so writing combinatorial tests should not introduce to much of a hurdle.

Once a developer learns all basics of CombTest, writing combinatorial tests is not particularly more time consuming than writing regular test cases. The main complexity lies in developing an input parameter model and a functioning test oracle. Both of these tasks reside outside of the CombTest framework and thus do not influence the usability.

Due to the extensive reporting possible with CombTest, users should also be able to locate any errors they make early on. All in all, this means using CombTest should be no more difficult than using any other Java testing framework.

Reliability Since CombTest could not be tested in production, not much can be said about its reliability. Additionally, large parts of the complexity reside in specific algorithm implementations. Those are supplied by third party developers, and thus cannot be evaluated for reliability.

To mitigate the risk of having a framework which always fails, there are many unit tests in addition to the acceptance tests introduced in section 6.1. Tests are of course never a proof of working software, but they increase the confidence that CombTest is reliable.

One part which could be improved is error handling around extension points. For example, if an external `TestCaseGroupGenerator` was not written very well, one would not expect the whole framework to fail.

Security Security was not a concern during the development of CombTest, since the framework should only be used in isolated testing environments, and never in production systems.

Maintainability An important concern in a software product's longevity is its maintainability. If no one can fix bugs or introduce new functionality, developers will move away from the framework. CombTest's maintainability is held high mainly through a good modularity concept. Since engine, model, and the JUnit extension are in different Maven modules which have a clear dependency structure Figure 5.1 shows, changing details like the extension is easily possible without affecting core functionality in engine. Additionally, nearly all classes form a DAG in their dependency structure. This also eases maintainability.

Countless JUnit5 test cases proof the testability of CombTest, but there is room for improvement. Particularly the managing classes like `BasicCombinatorialTestManager` have too many dependencies which introduce a significant overhead in testing.

Portability Lastly, ISO/IEC 25010:2011 considers portability. Paragraph 6.2.1 already evaluated some sub-characteristics of this category. For example, Maven automatically introduces an easy installation and replacement process. Additionally, Java is a very portable programming language. There are Java versions for nearly all major operating systems. Since CombTest uses Java 8, which is one of the most used Java version to date, nearly everyone can use CombTest on their computers.

6.2.2 SonarQube

To get a few metrics about its code quality, CombTest was examined using SonarQube [SS18]. The analysis used commit fd8cc50e97cf36ed860fa00891e810b79c92c707 and an off-the-shelf version of SonarQube 7.2.1. Consequently it is easily reproducible.

The analysis uncovered some interesting statistics. CombTest consists of exactly 165 classes stored in 150 files. They collectively contain 756 functions spread across 6,887 lines of code. In addition to these code lines, there are 22.4% comments and some empty lines, amounting to a total of 12,118 lines. By far the biggest module is engine, containing 3851 lines of code, followed by model and then the JUnit extension.

While SonarQube does report some issues for each module, all of those are false positives. For example, one issue criticizes catching `Throwable` instead of `Exception`. However, catching `Throwable` is intended to notice test failures. Otherwise, CombTest could fail every time a test case uses an JUnit `assert` method. Hence, these kind of issues can, and should, not be fixed.

SonarQube reports test coverage at 78.9% for all modules, with engine and model having 89.3% and 85.5% respectively. For these two modules the numbers are quite good and indicate enough tests, but the JUnit extension is not tested at all since it was just implemented as a proof of concept on top of the framework. However, extensive manual testing showed no failures.

6.3 Experiments

Knowing that CombTest can, in some way, localize faults and adheres to certain quality standards is a very important step. However, for practical use knowing that it can localize these faults in a way which works with real combinatorial testing is even more important. Therefore, this section deals with some experiments which were conducted to show that CombTest reliably finds failure-inducing combinations for IPMs near to practical use (that is, if the used localization algorithm is able to find the combinations).

To examine CombTest for this characteristic, several experiments were conducted. For each experiment, a model definition was taken from the CITLAB's unconstrained benchmark IPMs, and CombTest's JUnit extension introduced several failure-inducing combinations [VG12; VG13]. Since writing test oracles or generating valid inputs for programs based on abstract IPMs is another big and complex topic of combinatorial testing, all faults were mocked instead by throwing an exception via JUnit's `assertFalse` for specific input combinations.

The experiments used two model from CITLAB: *Banking2* and *Healthcare2*. They each have several constraints, since almost no realistic IPM is totally unconstrained. Banking2 has a 4×2^{14} configuration. This means that there is one parameter with four values, and fourteen parameters with two values each. Meanwhile, Healthcare2 has a $4 \times 3^5 \times 2^5$ configuration. Testing strength was set to 3 in each test. Every experiment used CombTest's Reporter extension point to gather interesting information, like the number of fault localization iterations, total number of test cases, and many more. Table

Fault	Model	Algo.	#Init.	#Local.	#Iter.	#Found
0=0, 1=0, 2=0	Healthcare2	AIFL	73	189	1	6528
		IDD	73	10	10	1
		IterAIFL	73	1235	2	6528
		BEN	73	58	7	1
	Banking2	AIFL	47	81	1	18344
		IDD	47	11	11	1
		IterAIFL	47	564	2	18344
		BEN	47	27	4	1
0=0, 1=0	Healthcare2	AIFL	73	249	1	17501
		IDD	73	6	6	1
		IterAIFL	73	1678	2	17501
		BEN	73	58	7	1
	Banking2	AIFL	47	124	1	58251
		IDD	47	7	7	1
		IterAIFL	47	899	2	58251
		BEN	47	15	2	175

Table 6.1: All experiment results. The rows contain (from left to right): The failure-inducing combinations, CITLAB model, fault localization algorithm, number of initial test cases, number of localization test cases, number of fault localization iterations, number of “failure-inducing” combinations the algorithm reported.

6.1 shows some selected values. Each fault is represented as a collection of value and parameter indexes. For example, 0=1,3=2 means that the failure-inducing combination is mapping the first parameter to its second value, and the fourth one to its third value (indexes start at zero). The indexes are taken from the `InputParameterModels` inside the test classes in the experiments package inside the evaluation Maven module.

As we can see in Table 6.1, the algorithms all have very different usage patterns. While AIFL and IterAIFL use only one and two iterations respectively and generate many fault localization test cases, Improved Delta Debugging takes the opposite approach and generates only few test cases with one test case per iteration. BEN is in between those extremes, generating multiple test cases per iteration with an relatively arbitrary number of test cases. The number of calculated possible failure-inducing combinations also differs greatly due to algorithm design. In each case, the returned combinations included the failure-inducing combination. All in all, the results show that CombTest can support a wide variety of fundamentally different fault localization algorithms for combinatorial testing.

7 Conclusion

The first two chapters of this thesis introduced the general problem of automated fault localization for combinatorial testing. While there were some available programs for fault localization, none of them allowed for a fully automated setting. To that end, this thesis has made several contributions.

First, chapter 2.1 gave a general overview of combinatorial testing. In addition to basics like parameter, values, and constraints, it introduced more complex topics such as negative testing and fault localization. Chapter 3 then analyzed existing work done in the realm of combinatorial testing. While there are some approaches to automated combinatorial testing and to manual fault localization, there is no work connecting both topics for automated fault localization. Thus, the need for a corresponding framework was established.

Before one could write any such framework, on needs basic requirements. Consequently, chapter 4 examined objectives any fault localization framework should fulfill. This included the need for an extensible platform to which third party developers could contribute their own algorithms to advance the field through collaboration. To reach these goals, a framework needs to satisfy many requirements, both specific to general fault localization, and to an extensible platform. Those requirements were specified, and a following section extracted a general architecture out of them.

As a proof on concept for this general architecture, chapter 5 introduced CombTest, a Java framework. Besides a general usage API, this chapter also presented some implementations for common fault localization algorithms.

Chapter 6 then evaluated CombTest. While we saw that CombTest fulfills all requirements and the experiments showed that the general idea of automated fault localization worked as expected, there were some points which the evaluation by ISO/ICE 25010:2011 found deserving further improvements. The following list of future improvements includes those points. In addition to general framework enhancements, it also includes general research which could advance the field of (automated) fault localization:

Evaluate CombTest Support in other Testing Frameworks

Currently, only a JUnit extension exists for CombTest. Since it is designed to not depend on any specific testing framework, it should be possible to introduce extensions for other testing frameworks. For example, TestNG could also support combinatorial tests via CombTest.

Parallel Test Execution

JUnit 5.3 introduced support for native parallel test execution. This even works when using `TestTemplateInvocationContextProviders` like CombTest's JUnit extension. However, JUnit currently stops the execution if one parallel test executor does not find any more test cases to evaluate. Since other executing threads could still be evaluating the result for test cases from the initial test suite, fault localization algorithms may need to generate additional test cases afterwards. These test cases will no longer be executed. A possible solution would be to make the `TestCaseSpliterator` thread safe. Since the actual execution happens in the `tryAdvance` method, this would not be as easy as adding a **`synchronized`** keyword.

Test Case Caching

CombTest recalculates all test cases for every run. Consequently, complex models take a very long time to evaluate. This is especially obstructive in the development of new test cases, when a developer already finished his/her model, and just wants to adjust the actual test method. Therefore, it would make sense to introduce functionality which caches calculated test cases based on equivalence of IPMs and generators. As a result, only one run would take a very long time, and any subsequent runs would be much faster. This feature could be realized as an additional extension point so that developers can use any external data storage they want. Therefore, sharing test cases across development and testing machines would be possible with a central database.

Increase Testability of Managers

As stated in paragraph 6.2.1, most managers like the `BasicCombinatorialTestManager` are not very well testable. This class could be refactored into multiple classes to support a better testability. Next, one could write more unit tests to guarantee sufficient regression testing.

Dynamic Fault Localization and Combinatorial Testing

Paragraph 6.2.1 introduced the concept of algorithms which combine fault localization and general combinatorial testing. In these algorithms, passing tests cases generated for fault localization could count towards the t-value-combination coverage criterion. To date, no such algorithms are known, and as of now, CombTest does not have the ability to support it. Consequently, future work could develop such an algorithm and implement support for it in CombTest.

Bibliography

[Bec+18] S. Bechtold, S. Brannen, J. Link, M. Merdes, M. Phillip, and C. Stein. *JUnit 5 User Guide*. https://junit.org/junit5/docs/current/user-guide/. Last retrieved 2018-09-18. 2018 (cited on page 14).

[Bec97] K. Beck. “SIMPLE SMALLTALK TESTING”. In: *Kent Beck’s Guide to Better Smalltalk: A Sorted Collection*. SIGS Reference Library. Cambridge University Press, 1997, 277–288. DOI: 10.1017/CBO9780511574979.033 (cited on page 13).

[BS17] Baeldung SRL. *Java in 2017 Survey Result*. https://www.baeldung.com/java-in-2017. Last retrieved 2018-09-18. 2017 (cited on page 12).

[CM08] C. J. Colbourn and D. W. McClary. “Locating and detecting arrays for interaction faults”. In: 15 (Aug. 2008), pp. 17–48 (cited on page 9).

[Cze06] J. Czerwonka. “Pairwise Testing in Real World. Practical Extensions to Test Case Generators”. In: 2006 (cited on pages 8, 15).

[FL18] K. Fögen and H. Lichter. “Combinatorial Testing with Constraints for Negative Test Cases”. In: *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Apr. 2018, pp. 328–331. DOI: 10.1109/ICSTW.2018.00068 (cited on pages 1, 8, 18, 22, 33, 35).

[Fow] M. Fowler. *InversionOfControl*. https://martinfowler.com/bliki/InversionOfControl.html. Accessed: 13.08.2018 (cited on page 25).

[Gam+95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2 (cited on pages 15, 38).

[Gha+13] L. S. Ghandehari, Y. Lei, D. Kung, R. Kacker, and R. Kuhn. “Fault localization based on failure-inducing combinations”. In: *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. Nov. 2013, pp. 168–177. DOI: 10.1109/ISSRE.2013.6698916 (cited on pages 1, 16).

[Gha+15] L. S. Ghandehari, J. Chandrasekaran, Y. Lei, R. Kacker, and D. R. Kuhn. “BEN: A combinatorial testing-based fault localization tool”. In: *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Apr. 2015, pp. 1–4. DOI: 10.1109/ICSTW.2015.7107446 (cited on page 1).

[Goe14] B. Goetz. *Project Valhalla.* `http://openjdk.java.net/projects/valhalla/`. Last retrieved 2018-09-18. 2014 (cited on page 33).

[GOM06] M. Grindal, J. Offutt, and J. Mellin. *Handling Constraints in the Input Space when Using Combination Strategies for Software Testing.* Tech. rep. HS- IKI -TR-06-001. University of Skövde, School of Humanities and Informatics, 2006 (cited on page 7).

[HC02] C. S. Horstmann and G. Cornell. *Core Java 2: Volume I, Fundamentals, Sixth Edition.* 6th. Pearson Education, 2002. ISBN: 0130471771 (cited on page 12).

[HH08] B. Haugset and G. K. Hanssen. "Automated Acceptance Testing: A Literature Review and an Industrial Case Study". In: *Agile 2008 Conference.* Aug. 2008, pp. 27–38. DOI: `10.1109/Agile.2008.82` (cited on page 41).

[Jia+07] H. Jia, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence. "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)* (2007), pp. 549–556 (cited on page 7).

[KR02] D. R. Kuhn and M. J. Reilly. "An investigation of the applicability of design of experiments to software testing". In: *27th Annual NASA Goddard/IEEE Software Engineering Workshop, 2002. Proceedings.* Dec. 2002, pp. 91–95. DOI: `10.1109/SEW.2002.1199454` (cited on page 1).

[KWAMG04] D. R. Kuhn, D. R. Wallace, and J. A. M. Gallo. "Software Fault Interactions and Implications for Software Testing". In: *IEEE Trans. Softw. Eng.* 30.6 (June 2004), pp. 418–421. ISSN: 0098-5589. DOI: `10.1109/TSE.2004.24`. URL: `http://dx.doi.org/10.1109/TSE.2004.24` (cited on page 6).

[LNL12] J. Li, C. Nie, and Y. Lei. "Improved Delta Debugging Based on Combinatorial Testing". In: *2012 12th International Conference on Quality Software.* Aug. 2012, pp. 102–105. DOI: `10.1109/QSIC.2012.28` (cited on page 16).

[Mar00] R. C. Martin. "Design principles and design patterns". In: *Object Mentor* 1.34 (2000), p. 597 (cited on page 10).

[Mar17] R. C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design.* 1st. Upper Saddle River, NJ, USA: Prentice Hall Press, 2017. ISBN: 0134494164, 9780134494166 (cited on pages 10, 12, 25, 32).

[MMR14] J. P. Miguel, D. Mauricio, and G. Rodriguez. "A Review of Software Quality Models for the Evaluation of Software Products". In: *CoRR* abs/1412.2977 (2014). arXiv: `1412.2977`. URL: `http://arxiv.org/abs/1412.2977` (cited on page 43).

[ND12] S. Nidhra and J. Dondeti. "Black Box and White Box Testing Techniques - A Literature Review". In: *International Journal of Embedded Systems and Applications* 2.2 (June 2012) (cited on page 4).

[NL11] C. Nie and H. Leung. "A Survey of Combinatorial Testing". In: *ACM Comput. Surv.* 43.2 (Feb. 2011), 11:1–11:29. ISSN: 0360-0300. DOI: 10.1145/1883612.1883618. URL: http://doi.acm.org/10.1145/1883612.1883618 (cited on pages 1, 15).

[OC18] Oracle Corporation. *Go Java*. https://go.java/index.html. Last retrieved 2018-09-18. 2018 (cited on page 12).

[Phi15] M. Philipp. *JUnit Lambda*. https://www.indiegogo.com/projects/junit-lambda/. Last retrieved 2018-09-18. 2015 (cited on page 13).

[PN12] M. Patil and P. Nikumbh. "Pair-wise Testing Using Simulated Annealing". In: *Procedia Technology* 4 (2012). 2nd International Conference on Computer, Communication, Control and Information Technology(C3IT-2012) on February 25 - 26, 2012, pp. 778 –782. ISSN: 2212-0173. DOI: https://doi.org/10.1016/j.protcy.2012.05.127. URL: http://www.sciencedirect.com/science/article/pii/S2212017312004069 (cited on page 7).

[PS17] I. Pivotal Software. *Spring Framework*. https://spring.io/. Last retrieved 2018-09-18. 2017 (cited on page 12).

[Ret14] I. RetailMeNot. *Pairwise*. https://github.com/RetailMeNot/pairwise. Last retrieved 2018-09-18. 2014 (cited on page 15).

[SEI10] C. M. U. Software Engineering Institute. *What is Your Definition of Software Architecture*. Dec. 2010 (cited on page 10).

[Sim18] K. K.D. E. Simos. "An Efficient Design and Implementation of the In-Parameter-Order Algorithm". In: *Mathematics in Computer Science* 12.1 (Mar. 2018), pp. 51–67. ISSN: 1661-8289. DOI: 10.1007/s11786-017-0326-0. URL: https://doi.org/10.1007/s11786-017-0326-0 (cited on page 35).

[Sky18] Skymind. *Deeplearning4j*. https://deeplearning4j.org/. Last retrieved 2018-09-18. 2018 (cited on page 12).

[SNX05] L. Shi, C. Nie, and B. Xu. "A Software Debugging Method Based on Pairwise Testing". In: *Computational Science – ICCS 2005*. Ed. by V. S. Sunderam, van Geert Dick Albada, P. M. A. Sloot, and J. Dongarra. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 1088–1091. ISBN: 978-3-540-32118-7 (cited on pages 1, 16).

[SS18] SonarSource S.A. *Sonarqube 7.2*. https://www.sonarqube.org/. Last retrieved 2018-09-18. June 2018 (cited on page 47).

[Sta11] International Organization for Standardization. *ISO 25010*. https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en. Last retrieved 2018-09-18. 2011 (cited on page 43).

[Ste+04] J. M. Stecklein, J. Dabney, B. Dick, B. Haskins, R. Lovell, and G. Moroney. "Error Cost Escalation Through the Project Life Cycle". In: (Toulouse, France, June 19–24, 2004). INCOSE Foundation, 2004 (cited on page 3).

[TASF04] The Apache Software Foundation. *Apache Maven*. https://maven.apache.org/what-is-maven.html. Last retrieved 2018-09-18. 2004 (cited on page 12).

[TASF18] The Apache Software Foundation. *Maven - Introduction to the Build Lifecycle*. https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html. Last retrieved 2018-09-18. 2018 (cited on page 13).

[TMD09] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009. ISBN: 0470167742, 9780470167748 (cited on page 10).

[TSB18] TIOBE Software BV. *TIOBE Index for August 2018*. https://www.tiobe.com/tiobe-index/. Last retrieved 2018-09-18. Aug. 2018 (cited on page 12).

[Uka17] H. Ukai. *JCUnit*. https://github.com/dakusui/jcunit. Last retrieved 2018-09-18. 2017 (cited on pages 1, 15, 16).

[UPL12] M. Utting, A. Pretschner, and B. Legeard. "A Taxonomy of Model-based Testing Approaches". In: *Softw. Test. Verif. Reliab.* 22.5 (Aug. 2012), pp. 297–312. ISSN: 0960-0833. DOI: 10.1002/stvr.456. URL: http://dx.doi.org/10.1002/stvr.456 (cited on page 1).

[UQ17] H. Ukai and X. Qu. "Test Design as Code: JCUnit". In: *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. Mar. 2017, pp. 508–515. DOI: 10.1109/ICST.2017.58 (cited on page 15).

[VG12] P. Vavassori and A. Gargantini. "CITLAB: A Laboratory for Combinatorial Interaction Testing". In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation(ICST)*. Vol. 00. Apr. 2012, pp. 559–568. DOI: 10.1109/ICST.2012.141. URL: doi.ieeecomputersociety.org/10.1109/ICST.2012.141 (cited on page 47).

[VG13] P. Vavassori and A. GargantiniA. *CITLAB Constrained Benchmark models*. https://sourceforge.net/p/citlab/code/HEAD/tree/trunk/citlab.benchmarks/constrained/. Last retrieved 2018-09-18. 2013 (cited on page 47).

[Vig14] S. Vigna. *fastutil*. `http://fastutil.di.unimi.it/`. Last retrieved 2018-09-18. 2014 (cited on page 33).

[Wan+10] Z. Wang, B. Xu, L. Chen, and L. Xu. "Adaptive Interaction Fault Location Based on Combinatorial Testing". In: *2010 10th International Conference on Quality Software*. July 2010, pp. 495–502. DOI: `10.1109/QSIC.2010.36` (cited on pages 1, 16).

[YZ09] J. Yan and J. Zhang. "Combinatorial Testing: Principles and Methods". In: 20 (July 2009) (cited on page 7).

[Zhe+16] W. Zheng, X. Wu, D. Hu, and Q. Zhu. "Locating Minimal Fault Interaction in Combinatorial Testing". In: *Advances in Software Engineering* 2016 (2016). DOI: `https://doi.org/10.1155/2016/2409521` (cited on pages 9, 16).